TURING MACHINES

\[ \text{REGULAR LANGUAGES} \sim \text{DFA}s \text{ NFA}s \text{ REG-EXP.} \]
\[ \text{CFLs} \sim \text{PDA}s \text{ CFGs} \]

"LIMITED" MODELS OF COMPUTATION

\[ \text{LET } \mathcal{A} = \{ L : L \subseteq \Sigma^* \} \]
\[ \exists \text{? MACHINE MODEL "CAPTURING" } \mathcal{A} ? \]
\[ (\text{i.e. so that } \forall L \in \mathcal{A} \exists \text{ MACHINE FOR } L ?) \]

\[ \text{NO RECALL COUNTING ARGUMENT FROM} \]
\[ 1^{\text{st}} \text{ OR } 2^{\text{nd}} \text{ LECTURE: UNCOUNTABLY MANY} \]
\[ \text{FUNCTIONS, BUT ONLY COUNTABLY MANY} \]
\[ \text{PROGRAMS.} \]

\[ \text{QUESTIONS} \]
\[ (1) \text{ WHAT IS "MOST GENERAL" TYPE OF MACHINE?} \]
\[ (2) \text{ WHAT LANGUAGES DO THEY ACCEPT?} \]
\[ (3) \text{ WHAT FUNCTIONS DO THEY COMPUTE?} \]

ALAN TURING (1936) CAPTURES NOTION OF
"ALGORITHM", SEE CHURCH'S THESIS
LATER.
The Turing Machine is defined as:

\[ M = (Q, \Sigma, \Gamma, \delta, q_0, B, F) \]

where:

- \( Q \): Finite set of states
- \( \Sigma \): Finite input alphabet
- \( \Gamma \): Finite tape alphabet (\( \Sigma \subseteq \Gamma \))
- \( q_0 \): Initial state in \( Q \)
- \( F \): Accepting states in \( Q \)
- \( B \): Blank symbol in \( \Gamma - \Sigma \)

The transition function \( \delta \) is defined as:

\[ \delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{ L, R, S \} \]

- \( \text{CURRENT STATE} \)
- \( \text{SYMBOL SCANNED} \)
- \( \text{NEW STATE} \)
- \( \text{NEW SYMBOL TO WRITE} \)
- \( \text{MOVE LEFT, RIGHT, OR STAY} \)
$S(q,a) = (p, b, L)$ means:

If in state $q$ and scanning symbol $a$, then go to state $p$, overwrite $a$ with $b$, and then move one cell to left.

**Picture notation**

![Diagram](image)

**Note:** We allow $S(p,a)$ to be undefined for some choices of $p, a$. (In which case, machine crashes if in state $p$ scanning symbol $a$.)
INSTANTANEOUS DESCRIPTION
(OR "CONFIGURATION")

CONTAINS ALL NECESSARY INFORMATION TO
EXACTLY CAPTURE "CURRENT STATE OF THE
COMPUTATION"

THUS:

- STATE q OF MACHINE
- LOCATION OF READ/WRITE HEAD
- CONTENTS OF ALL CELLS FROM
  LEFT EDGE TO RIGHTMOST NON-BLANK,
  (OR TO READ/WRITE HEAD — WHICHEVER
  IS RIGHTMOST)

ID: \(X_1X_2X_3 \ldots X_{i-1}qX_iX_{i+1} \ldots X_n\)

\(q \in \mathcal{Q}\)
\(X_i \in \Gamma\)

INDICATES

\[
\begin{array}{cccccccc}
X_1 & X_2 & \cdots & X_{i-1} & X_i & X_{i+1} & \cdots & X_n \\
\end{array}
\]

TAPE = \(X_1X_2\ldots X_n\), STATE = \(q\), SCANNING \(i^{th}\) CELL,
WHICH CONTAINS \(X_i\)
Relation $I^m$ for TM $M$ on IDs

If $S(q, x_i) = (p, y, l)$

Then

$x_1x_2...x_{i-1}qx_i x_{i+1}...x_n \xrightarrow{I^m} x_1x_2...x_{i-2}px_{i-1}yx_{i+1}...x_n$

\underline{Current ID} \hspace{1cm} \underline{New ID}

- Similarly for moving right, or staying in one spot.
- If $S(q, x_i)$ is undefined, then $\not\exists$ next ID
- If machine tries to move off left edge - $\not\exists$ next ID, (it crashes)

$I^*_m = \text{Reflexive} \& \text{Transitive Closure of } I^m$

Thus $ID_1 I^*_m ID_2$ iff the machine, when run starting from $ID_2$, ends up in $ID_2$ after some finite # of steps.
INITIAL ID: \( q_0 \) \\

ACCEPTING ID: \( \alpha_1, q_f \alpha_2 \) \( q_f \in F \) \( \alpha_1, \alpha_2 \in \Gamma^* \)

\[ M \text{ accepts } w \]

iff \( \exists \alpha_1, \alpha_2 \in \Gamma^* \) and \( \exists q_f \in F \)

such that \( q_0 w \xrightarrow{\ast} M \alpha_1 q_f \alpha_2 \)

Thus \( M \) accepts if at any time it enters an accepting state

[Note it may not have read all of \( w \), or may have passed back and forth over \( w \) any number of times, and \( w \) may have been completely written over or changed]

\[ L(M) = \{ w : M \text{ accepts } w \} \]

\[ = \{ w : \text{ for some } q_f \in F \text{ and } \alpha_1, \alpha_2 \in \Gamma^* \]

\[ q_0 w \xrightarrow{\ast} M \alpha_1 q_f \alpha_2 \} \]
OBSERVE:

- Why bother to have > 1 accept state?
- No apparent way for machine to indicate "reject".

NEW MODEL (equivalent to one just defined ... exercise)

\[ M = (Q, \Sigma, \Gamma, \delta, q_0, \emptyset, q_{\text{accept}}, q_{\text{reject}}) \]

WHERE \( q_{\text{accept}} \) AND \( q_{\text{reject}} \) ARE UNIQUE STATES, FROM WHICH NO TRANSITIONS ARE POSSIBLE.

\[ \rightarrow M \text{ accepts } w \text{ iff } q_0w \xrightarrow{\delta^*} q_1 \text{ for some } q_1 \in Q^* \]

\[ \rightarrow M \text{ rejects } w \text{ iff } \]

\[ q_0w \xrightarrow{\delta^*} q_1 \text{ for some } q_1 \in Q^* \] or

\[ M \text{ doesn't halt on } w \text{ (infinite loop)} \] or

\[ M \text{ moves off left edge} \] or

\[ \text{at some point, no next transition is applicable, from state other than } q_{\text{accept}}. \]
IF M ACCEPTS, WE CAN TELL (IT ENDS UP IN q_accept).

IF M DOESN'T ACCEPT:
WE MAY NOT BE ABLE TO DETERMINE THIS.

M MAY "REJECT" BY NOT HALTING... THIS MAY NOT BE DETERMINABLE BY INSPECTING M's DESCRIPTION, w, AND/OR BY SIMULATING M ON w. (PERHAPS IF WE RUN M A LITTLE BIT LONGER — IT MIGHT ACCEPT?...)

RECURSIVELY ENUMERABLE (r.e.) LANGUAGES

= \{ L : \exists TM M such that L = L(M) \}

RECURSIVE LANGUAGES

= \{ L : \exists TM M that halts \forall w \in \Sigma^* AND such that L = L(M) \}

RECURSIVE: NICE:
GIVEN w, WE CAN TELL IF w \in L
BY RUNNING TM, WHICH WILL HALT AND EITHER ACCEPT OR REJECT

R.E. NOT SO NICE:
WILL TELL US IF w \in L, BUT MAY NOT TELL US IF w \notin L.
FACTS

CFL ⊈ Recursive ⊈ Recursively Enumerable

≤ by definition
≤ shown later

- Can program a TM to implement CYK algorithm, thus a CFG ∈ TM accepting w ∈ CL(g).
  So CFL ⊆ Recursive.

  Also, \(a^n b^n c^n : n \geq 1\) ∈ Recursive - CFL.

- Alt: Nondet. TMs ∼ Det TMs (ignoring time)
  and a PDA can be simulated by a Nondet TM using tape as stack.

(Both of these args are forward references)
Example: \( L = \{a^n b^n c^n : n \geq 1\} \)

Not context-free, but is recursive.

TM for \( L \):

**Intuition** — Go back & forth

"Marking" \( a, b, c \) one character at a time.

| \( A \) | Represents "Marked" \( a \)
| \( B \) | "b"
| \( C \) | "c"

At any time, tape will look like \( AAAaaa...BBbb...CCccc... \)

\[
S(q_0, a) = (q_1, A, R) \quad \text{Mark single "a" by changing it to "A"}
\]

\[
\begin{align*}
S(q_1, a) &= (q_1, a, R) \\
S(q_1, b) &= (q_1, b, R)
\end{align*}
\]

\( \text{Find next (unmarked) "b" while moving right} \)

\[
\begin{align*}
S(q_1, b) &= (q_2, B, R) \\
S(q_2, C) &= (q_2, C, R)
\end{align*}
\]

\( \text{Mark the } \text{b} \text{ by changing it to B} \)

\[
\begin{align*}
S(q_2, B) &= (q_2, B, R) \\
S(q_2, C) &= (q_2, C, R)
\end{align*}
\]

\( \text{Find next "c" while moving right} \)

\[
\begin{align*}
S(q_2, C) &= (q_3, C, L) \\
S(q_3, X) &= (q_3, X, L) \quad \text{for } X \in \{C, B, b, a\} \\
S(q_3, A) &= (q_0, A, R)
\end{align*}
\]

\( \text{Go back to first unmarked } a \)

\( \text{We moved one cell to far to left. Move one cell to right, and start over at } q_0 \text{ for another iteration} \)
How does the computation end?

- Most errors are handled by lack of transition (e.g., if "c" is found in q, while looking for "b" — this means there are too few b's, and M rejects by having no applicable move).

- To make "recursive" add transitions to q_reject for those that are missing.

**To Accept:**

\[ \delta(q_0, B) \]

\[ \rightarrow \] \[ \delta(q_{\text{check}}, B, R) \]

\[ \leftarrow \] Last "a" has been marked so a "B" is found in state q_0

\[ = (q_{\text{check}}, B, R) \]

\[ \rightarrow \] q_check is a state responsible for verifying that all b's, c's have been marked also.

If not — go to q_reject
If so — go to q_accept.

Transitions for q_check left as exercise.
TURING MACHINE ACCEPTING \( \exists a^n b^n c^n : n \geq 1 \)

Diagram:

- States: 10, 11, 12, 13, 9 CHECK, , 9 CHECK, 9 CHECK, 9 accept
- Transitions:
  - \( a/A, R \) from 10 to 11
  - \( b/B, R \) from 11 to 12
  - \( c/C, L \) from 12 to 13
  - \( B/B, R \) from 13 to 2
  - \( C/C, R \) from 2 to 13
  - Blank Symbol from 13 to 9 accept

Transitions not shown: \( \exists a^n b^n c^n : n \geq 1 \)
Computing Functions with TMs

Ignore CPLXITY for now.
Use unary notation

- $i$ represented by $0^i = \underbrace{000\ldots00}_i$

[How to represent $0^i$?]

- $M(i) \overset{\Delta}{=} \text{output of } M \text{ on "input" } i \text{ (that is, } 0^i\text{)}$
  
  If $q_0 \overset{1^i}{\rightarrow}_m \text{ qHALT } 0^j$ then we say $M(i) = j$

- Every TM computes some fn: $\mathbb{N} \rightarrow \mathbb{N}$
  (which may not be defined on some (many) args)

- $M(i) \uparrow \text{ "diverges" if } \exists j \text{ such that } q_0 \overset{1^i}{\rightarrow} m ^j$
  ($M(i)$ could diverge by not halting, or by halting without proper conventions)

- Multiple arguments $\langle i_1, i_2, \ldots, i_n \rangle$ represented by
  $0^{i_1} 1 0^{i_2} 1 \ldots 10^{i_n}$

$M(i_1, i_2, \ldots, i_n) = \langle j_1, \ldots, j_m \rangle \text{ iff } q_0 \overset{1^i}{\rightarrow} m ^j \overset{1^*}{\rightarrow} m ^{j_1} 0^{i_1} 10^{i_2} \ldots 0^{i_n}$
Example: Unary Addition

Must return to left edge without falling off.

Should have marked leftmost symbol initially, or inserted "$" at left edge, and shift input to right (we do this in a few minutes).
OTHER EXAMPLES

. PROPER SUBTRACTION ("MONUS")

\[ m - n = \begin{cases} m - n & \text{if } m - n > 0 \\ 0 & \text{otherwise} \end{cases} \]

. \[ 2^n \]

. ANY FUNCTION THAT YOU HAVE AN "ALGORITHM" FOR.

"TM-COMPUTABLE FUNCTIONS" "PARTIAL RECURSIVE FUNCTIONS"

\[ = \{ f : \mathbb{N} \to \mathbb{N} \cup \{\text{undefined}\} : \exists M \text{ s.t. } \forall x \ f(x) = \text{M}(x) \} \]

TOTAL RECURSIVE FUNCTIONS

\[ = \{ f : \mathbb{N} \to \mathbb{N} : \exists M \text{ s.t. } \forall x \in \mathbb{N} \ f(x) = \text{M}(x) \} \]

\[ \text{M}(x) \text{ defined } \forall x, \text{ M never } \uparrow. \]
COMMENTS / CONCERNS

- Why consider only $f: \mathbb{N} \to \mathbb{N}$?
- What about negatives?
- Rationals?
- Reals?

**Answer**

Negatives, rationals can be encoded as finite sets of integers, or as strings.

E.g. $-5$ could be coded $100000$

\( \frac{p}{q} \) could be coded $0^{p}110^{q}$

And $\frac{p}{q} + \frac{a}{b}$ as $0^{p+b+a}110^{b}$

Reals... can be approximated to given precision, or we can use ASCII symbolic names.

Intuition: Any function can be represented as a function from $\mathbb{N} \to \mathbb{N}$ via coding.
GENERAL TM TRICKS

- SHIFTING OVER
- MULTIPLE TRACKS
- CHECKING OFF SYMBOLS
- SUBROUTINE CALLS
- USING FINITE CONTROL MEMORY

"EXTENSIONS" OF TMs

- 2-WAY ∞ TAPE
- MULTIPLE TAPE
- NONDETERMINISTIC TMs
- MULTI-DIMENSIONAL TMs
- MULTI-HEAD TMs

GOAL: SHOW HOW MUCH IS POSSIBLE WITH THE BASIC MODEL.
GENERAL SHIFT-BY-K CHARs
FOR ANY CONSTANT K

- USE STATES TO REMEMBER PREV \(| \Sigma |^k\) CHARs
- SIMILAR TO DFA FOR "AN-CHAR-FROM-END IS A "I"

\[ b_1, b_2, ..., b_k \xrightarrow{a/b_i, R} b_2, ..., b_k, a \]

+ STATES TO BEGIN, END THE PROCESS
USING FINITE STATE MEMORY

- just like DFAs.
- can use tuples to store different types of information

**Example**

**Design** $M$ to accept

\[ \{a^n b^n c^n : n \equiv 1 \mod 4 \text{ and } n \equiv 2 \mod 7\} \]

**States:** $\langle q, i, j \rangle$

**Initial state:** $\langle q_0, 0, 0 \rangle$

**Transition:** $\delta(\langle q, i, j \rangle, a) = \langle \delta(q, a), i + 1 \mod 4, j + 1 \mod 4 \rangle$

**With final states** $\{ \langle q_{\text{accept}}, 1, 2 \rangle \}$
**MULTIPLE TRACKS**

<table>
<thead>
<tr>
<th>Track</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>q</td>
<td>b</td>
<td>b</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>

In can address any particular track in the cell it is visiting.

Such a machine is easily simulated by a standard single-track TM as follows...
NEW ALPHABET CONTAINING FUNNY SYMBOLS
OF FORM \[
\begin{array}{c|c|c|c|c}
& X & Y & Z \\
\hline 
X & & & \\
Y & & & \\
Z & & & \\
\end{array}
\]
REPRESENTING ALL TRACK INFO

S CAN BE CONSTRUCTED SO AS TO "IGNORE" TRACKS
THAT WEREN'T ADDRESSED BY ORIGINAL MACHINE

E.G. TO MOVE LEFT WHILE CHANGING BOTTOM TRACK
FROM B TO C, WE'D HAVE TRANSITIONS:

\[ S(9, \begin{array}{c|c|c}
X & Y & Z \\
\hline 
X & & \\
Y & & \\
\end{array}) = (9, \begin{array}{c|c|c}
X & Y & Z \\
\hline 
& & \\
& & \\
\end{array}) \quad \forall x, y, z \in \Gamma \]
**Example: Adding Binary Numbers (of same length)**

**INPUT**

\[ a_1 a_2 a_3 \ldots a_n \# b_1 b_2 \ldots b_n \frac{3}{2} \]

\( a_i, b_i \in \{0, 1\} \)

**Idea:** Pre-process so input looks like

\[
\begin{array}{c|c|c|c|c}
& a_1 & a_2 & a_3 & \ldots & a_n \\
\hline
& b_1 & b_2 & b_3 & \ldots & b_n \\
\end{array}
\]

Then addition is easy

(Dea can do it if input in this form)

*left as exercise

**Start scanning** at right end

- **CARRY = 0**
  - \( 0/0, L \)
  - \( 0/1, L \)
  - \( 1/0, L \)
  - \( 1/1, L \)

- **CARRY = 1**
  - \( 0/0, L \)
  - \( 0/1, L \)
  - \( 1/0, L \)
  - \( 1/1, L \)
Example: Adding Binary Numbers (of same length)

Input: $a_1 a_2 a_3 \ldots a_n \# b_1 b_2 \ldots b_n \#$

$a_i, b_i \in \{0, 1\}$

Idea: Pre-process so input looks like

<table>
<thead>
<tr>
<th>$a_1$</th>
<th>$a_2$</th>
<th>$a_3$</th>
<th>$\ldots$</th>
<th>$a_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_1$</td>
<td>$b_2$</td>
<td>$b_3$</td>
<td>$\ldots$</td>
<td>$b_n$</td>
</tr>
</tbody>
</table>

Then addition is easy

(DFA can do it if input in this form)

*Left as exercise

Start scanning \[ \frac{a_n}{b_n} \] at right end

Carry = 0

\[ \begin{array}{c}
0/0, L \\
1/0, L \\
0/1, L \\
1/1, L \\
\end{array} \]

Elim leading 0's by shifting

\[ \begin{array}{c}
0/0, L \\
0/0, L \\
1/1, L \\
\end{array} \]

Carry = 1

\[ \begin{array}{c}
0/0, L \\
1/1, L \\
\end{array} \]

$\Rightarrow$ Halt
CHECKING OFF SYMBOLS

- ALREADY SEEN EXAMPLE: \( \Sigma a^n b^n c^n : n \geq 1 \)

  USED "A" TO REPRESENT "CHECKED" a.

- CAN MAKE IT MORE OBVIOUS BY USING TWO-TRACK TYPE OF MACHINE

  FIRST CHANGE INPUT TO

  \[
  \begin{array}{c}
  a \quad a \quad a \quad a \quad a \quad b \quad b \quad b \quad b \quad c \quad \ldots \quad c
  
  \end{array}
  \]

  THEN AT ANY POINT, COMPUTATION TAPE LOOKS LIKE:

  \[
  \begin{array}{c}
  a \quad a \quad a \quad a \quad a \quad \checkmark \quad \checkmark \quad \checkmark \quad \checkmark \quad \checkmark \quad \checkmark \quad \checkmark \quad \checkmark \quad \checkmark \quad \checkmark \quad \checkmark \quad \checkmark \quad \ldots \quad c \quad c
  
  \end{array}
  \]
EXTENSIONS OF TMs

2-WAY INFINITE TAPE

... -5 -4 -3 -2 -1 0 1 2 3 4 5 ...

REPRESENT ON 1-WAY INFINITE TAPE,
MODIFYING TRANSITIONS AS APPROPRIATE:

$ 0 1 -1 1 2 -2 2 3 -3 3 4 -4 4 ...

OR WITH 2-TRACKS:

$ O 1 2 3 4 5 -1 -2 -3 -4 -5 -5 ...
MULTI-TAPE TMs

- **K-TAPE TM**

  - K DIFFERENT (2-WAY ∞) TAPES
  - K DIFFERENT INDEPENDENTLY CONTROLLABLE READ/WRITE HEADS

- INPUT INITIALLY ON TAPE 1
- TAPES 2, 3, 4, ..., K INITIALLY BLANK

- SINGLE MOVE
  - READ SYMBOLS UNDER ALL HEADS
  - PRINT NEW SYMBOLS (POSSIBLY DIFFERENT)
  - MOVE ALL HEADS (POSSIBLY DIFFERENT DIRECTIONS)
  - GO TO NEW STATE

\[ S(q, a_1, a_2, ..., a_k) = (q', b_1, b_2, ..., b_k, D_1, D_2, ..., D_k) \]

- SYMBOL SCANNED ON TAPE 2
- NEW SYMBOL WRITTEN ON TAPE 2
- MOVE MADE ON TAPE 2
UTILITY OF MULTI-TAPE TM

$L = \{\overline{w#w} : w \in \{0,1\}^* \}$

1-TAPE \( \Omega(n^2) \) STEPS

$10010 \# 10010$
Utility of Multi-Tape TM

\[ L = \Sigma \{ w \# w : w \in \Sigma^* \} \]

\[ 2 = \text{Tape} \quad \Omega(n^2) \text{ Steps} \]

\[ \begin{array}{c}
\shline
1 & 0 & 0 & 1 & 0 & \# & 1 & 0 & 0 & 1 & 0 \\
\shline
\end{array} \]

\[ \begin{array}{c}
\shline
1 & 0 & 0 & 1 \\
\shline
\end{array} \]
THEOREM: \( L \) accepted by \( k \)-tape \( M \) \( \Rightarrow \) \( L \) accepted by \( 1 \)-tape \( M' \)

Proof idea: Construct \( M' \) from \( M \)

\( M \) has \( k \) tapes
\( M' \) has \( 2^k \) tracks to simulate \( k \) tapes

But \( M \) has \( k \) heads!

... and everyone knows "2 heads are better than 1"
TRACK $2i-1 \approx \text{TAPE } i$

TRACK $2i \approx \text{POSITION OF HEAD } i$

EXAMPLE: $k = 2$

\[ \begin{array}{c}
M \\
\text{HEAD}_1 \\
\text{HEAD}_2
\end{array} \quad \begin{array}{c}
01101 \\
\downarrow
\end{array} \quad \begin{array}{c}
\text{TAPE } 1 \\
11010
\end{array} \quad \begin{array}{c}
\text{TAPE } 2
\end{array} \]

\[ \begin{array}{c}
M' \\
\begin{array}{c}
01101 \\
\checkmark
\end{array}
\end{array} \]

\[ \begin{array}{c}
\underline{11010} \\
\checkmark
\end{array} \]

A SINGLE MOVE OF $M$ REQUIRES MANY MOVES OF $M'$...
To simulate one move of M

1) Sweep from leftmost √ on any track to rightmost √ on any track, noting symbols √'d, and what track they are on. Save this info in finite control.

Implementation:

To do a "state q" move of M, M' starts in state \( \langle q, ?, ?, ?, ?, \ldots \rangle \) moving right over unmarked symbols, until...

\[ S(\langle q, ?, ?, ?, \rangle, \text{Track}) = (\langle q, ?, 1, ?, \rangle, \text{Track}, R) \]

Until a state \( \langle q, 1, 0, 0, 1, \ldots \rangle \) reached no ? marks.
Now \( M' \) has all \( k \) tape symbols scanned by \( M \) in its finite control.

\( M' \) now sweeps left "making appropriate changes" and ends up in next state \( \langle q', ?, ?, \ldots, ? \rangle \) at left of all scanned symbols.
\( \delta(9, 1, 0) = (p, 0, 0, l, r) \)

\( \delta' \langle 9, 1, 0 \rangle \)

[Grid diagram with markings]

\( \delta'' \langle 9, 1, 0 \rangle, \langle 9, 1, 0 \rangle \)

[Another grid diagram with markings]
CONSTRUCTION: \( M_1 \) CALLS \( M_2 \)

- **RENAME STATES** so \( M_1, M_2 \) have no common states

- **GOAL:** \( M_1 \) CALLS \( M_2 \) FROM STATE \( q \) AND CONTROL IS RETURNED TO STATE \( q' \)

- **RENAME** initial state of \( M_2 \): \( q_{\text{CALL}} \), \( q_{\text{RETURN}} \)

- **AFTER** \( M_1 \) SETS UP INPUT \( w_{\text{INPUT}} \#q_0q_1\ldots q_n \) AND POSITIONS HEAD AT \( q_1 \), IT GOES TO STATE \( q \)

  **ADD TRANSITIONS:**

  \[
  \delta(q, 0) = (q_{\text{CALL}}, 0, S) \quad \text{TRANSFER, CONTROL}
  \]

  \[
  \delta(q, 1) = (q_{\text{CALL}}, 1, S)
  \]

- **NOW** \( M_2 \) runs on \( \#q_0q_1\ldots q_n \) and ends up with \( \$ \ldots \#b_1b_2\ldots b_m \) \( \uparrow \) \( q_{\text{RETURN}} \)

  **ADD TRANSITIONS**

  \[
  \delta(q_{\text{RETURN}}, 0) = (q', 0, S)
  \]

  \[
  \delta(q_{\text{RETURN}}, 1) = (q', 1, S)
  \]
CAN BE MORE ELABORATE:

TO RETURN CONTROL TO SOME STATE $q_i$, CAN LEAVE INDEX $i$ IN SPECIAL SPOT ON TAPE:

```
# 1101 # a.a2a3...
```

SAVED COMPUTATION

RETURN TO STATE $q_i$ WHEN DONE

ARGUMENT PASSED TO SUBROUTINE

THEN $q_i$ RETURN GOES TO SPECIAL SEQUENCE OF STATES DESIGNED TO READ 1101 (OR WHATEVER WAS WRITTEN) AND THEN TRANSFER CONTROL TO CORRESPONDING STATE. (THIS ACTUALLY CAN BE DONE WITH A DFA)
MULTIDIMENSIONAL TMs

- OPERATES ON WORK GRID OF K DIMENSIONS
- INPUT WRITTEN ALONG 1ST AXIS
- S MUST SPECIFY DIRECTIONS L, R, U, D
  "UP" "DOWN"

FOR K DIM, S SPECIFIES ONE OF 2K DIRECTIONS TO MOVE (+ OR - IN DIRECTION)

**Proof**

- At any point in time, the k-dim. TM has used only finitely many cells.
- In X moves, it could not have moved out of the k-dim. hypercube with side length X.
- "Linearize" the hypercube by computing offsets, exactly how arrays are stored in computer memory.

**Example for 2-dim:**

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 \\
13 & 14 & 15 & 16 \\
\end{array}
\]

\[A[i,j] = 4(i-1) + j\]

- If TM moves outside of current cube, must recompute everything using larger cube.
  E.g., double cube side length, recompute offsets, and lay out contents based on new offsets.
MULTIHEAD TMs

- NO ADDITIONAL POWER.
- SIMULATE WITH STANDARD TM, USING TRACKS, SIMILAR TO K-TAPE TM

OFFLINE TM

[RELEVANT WHEN INVESTIGATING SPACE COMPLEXITY < LINEAR]

- SPECIAL READ-ONLY INPUT $\omega$
  WITH END MARKERS

- SEPARATE WORKTAPES (INITIALLY BLANK)

- SPECIAL CASE OF MULTITAPE TM, SO NO MORE POWERFUL.
- SIMULATE BY COPYING INPUT TO WORKTAPE
"RESTRICTED" TM MODELS

- DETERMINISTIC 2-STACK MACHINE ≅ TM

 deductible by:

TM moves on tape → shuttle characters back & forth between stacks

- COUNTER MACHINES

simulate TMs by counters → by 4 counters → by 2 counters

- no stack symbols
- just can push/pop blanks and test for empty

- see text ... very clever.
**Nondeterministic TMs** (NTM)

**Deterministic:** \( |S(q,a)| \leq 1 \)

**Nondeterministic:** \( S(q,a) \) is finite set of possibilities.

The TM may take any transition in \( S(q,a) \).

**Example**

\[
S(q,a) = \{(q_1,a_1,x_1), (q_2,a_2,x_2), \ldots (q_n,a_n,x_n)\}
\]

where \( q_i \in Q, a_i \in \Gamma, x_i \in \{L,R,S\} \)

\( T_M \) defined as for det. TM:

\[
x_1x_2\ldots x_{i-1}qx_i\ldots x_n \xrightarrow{T_M} x_1x_2\ldots x_{i-1}yp x_i\ldots x_n
\]

Iff \( (p,y,r) \in S(q,x_i) \)

**NTM accepts** \( w \) iff \( \exists \) sequence of moves leading to accept state.

I.e., iff \( q_0w \xrightarrow{T_M}^* q_{\text{accept}}\beta \) for some \( \alpha, \beta \in \Gamma^* \)
EXERCISE: EXTRA TAPES, 2-WAY TAPES, ETC.
DO NOT INCREASE POWER OF NTMs

SAME PROOF USED IN DET. TM CASE WORKS

THEOREM

$L$ ACCEPTED BY SOME NTM $N$

$\Rightarrow$

$L$ ACCEPTED BY SOME DTM $M$

Thus: The class of languages accepted by NTMs = REC. ENUM. LANGS.
LET \( N \) BE NONDET. TM.
WE CONSTRUCT DET. TM \( M \) THAT SIMULATES \( N \)

- CAN'T KEEP TRACK OF STATES \( N \) COULD BE IN - SINCE DIFFERENT NONDET. CHOICES LEAD TO DIFFERENT TAPE CONTENTS, HEAD POS., ETC.

- IDEA - TRY ALL POSSIBLE COMPUTATION SEQUENCES OF \( N \)...

LET \( r = \max \{|s(q,a)| \text{ in machine } N|_{q,a} \}

THUS AT ANY POINT IN TIME, \( N \) HAS AT MOST \( r \) POSSIBLE CHOICES.

WE CAN VIEW THE POSSIBLE RUNS OF \( N \) ORGANIZED AS AN \( r \)-ARY TREE AS SHOWN ON NEXT PAGE
Possible computations of $N$ on input $w$.

$$ID_0 = q_0w$$

$r$ branches

- $ID_i$
- $ID_l$
- $ID_r$

Consequence of taking $i^{th}$ transition applicable from $ID_0$.
POSSIBLE COMPUTATIONS OF N ON INPUT W

ID₀ = 9₀W

... R BRANCHES...

IDᵢ

... IDᵢᵣ

IDᵢᵣ

... IDᵢᵣᵣ

IDᵢᵣᵣ

... IDᵢᵣᵣᵣ...

IDᵢᵣᵣᵣ

... IDᵢᵣᵣᵣᵣ...

IDᵢᵣᵣᵣᵣ

... IDᵢᵣᵣᵣᵣᵣ...

IDᵢᵣᵣᵣᵣᵣ

... IDᵢᵣᵣᵣᵣᵣᵣ...

IDᵢᵣᵣᵣᵣᵣᵣ

CONSEQUENCE OF TAKING iᵗʰ TRANSITION APPLICABLE FROM ID₀

ID₅₈₃₁₉

ETC.
Possible Computations of $N$ on input $w$

$ID_0 = 90w$

$r$ branches

$ID_i$, $ID_j$, $ID_k$, ...

Consequence of taking $i^{th}$ transition applicable from $ID_0$

$ID_{i_1}$, $ID_{i_2}$, $ID_{i_3}$, ...

State of computation if $N$ takes 5th choice, then 8th, then 3rd, then 1st, then 9th, nondet choice.
If $N$ accepts $w$, it does so within a finite # of steps, say $S$.

(The value $S$ may be different for different accepted strings $w$)

There is a sequence of $S$ nondet choices which cause $N$ to accept... we can represent the sequence as an $r$-ary number of $S$ places: $a_1 a_2 a_3 \ldots a_S$, $a_i \in \{1, 2, \ldots, r\}$

- The construction of $M$ is to search for the sequence $a_1 a_2 \ldots a_S$ of choices, execute them, and accept $w$.

- $M$ must try all possible sequences, of all possible lengths.

- $M$ will do a breath-first search of $N$'s computation tree.

(Why not depth-first search?)
M uses 3 tapes to simulate N

Tape 1

Holds input w (is never changed)

Tape 2

- Used to simulate some possible computation of N on w.
- The particular non-deterministic choices made are dictated by contents of tape 3

Tape 3

Holds r-ary sequence a_1, a_2, ..., a_n to be used as "choices".

M uses subroutine, as needed, to generate in lexicographic order on tape 3, all possible r-ary sequences.

Example - if r = 2, then lex. ordering is

1, 2, 11, 12, 21, 22, 111, 112, 121, ...

So if subroutine were called with input 112, then it would output 121.
**MAIN PROGRAM**

- **INITIALIZE**: WRITE FIRST SEQ "1" ON TAPE 3

- COPY CONTENTS OF TAPE 1 (= INPUT W) TO TAPE 2

- SIMULATE N ON TAPE 2 AS FOLLOWS:
  (N USES ONLY ONE TAPE)
  
  - AT NEXT STEP OF N, IF STATE = q, SYMBOL = a
  
  THEN EXECUTE i^TH POSSIBILITY FOR S(q,a) IN N,
  WHERE i = CHARACTER READ ON TAPE 3.

  DETAIL: IN M, S(q,a,i) = i^TH ELEMENT OF S(q,a) IN N

  - AFTER EXECUTING TRANSITION, UPDATING TAPE 2
    AND STATE AS APPROPRIATE, MOVE TAPE 3 HEAD
    ONE CELL TO RIGHT. IF NEW SYMBOL ON
    TAPE 3 IS BLANK, GO TO UPDATE

  - IF N ACCEPTS NOW, THEN M DOES
    ELSE CONTINUE SIMULATION

**UPDATE**: REPLACE SEQ ON TAPE 3 WITH LEX. NEXT SEQ.
ERASE TAPE 2, AND START OVER.
WHY IT WORKS

M runs N (over and over and over......)
for longer and longer sequences of
non-deterministic choices.

If \( N \) accepts \( W \) then

3 seq of non-deterministic choices which cause
\( N \) to accept \( W \).
Eventually \( M \) will produce that sequence
on tape 3, and simulate the accepting computation.

If \( N \) does not accept \( W \) then

\( \exists \) seq of choices for which \( N(w) \) accepts.
\( M \) will run forever, hence not accept \( W \)

Note: \( M \) does not halt if on some seq of choices, \( N(w) \) halts & rejects.
If it did, the construction would fail
(why?)
"Anything algorithmic ("effectively computable") can be done on a Turing machine."

Not a formal stmt. or mathematical proposition. Cannot be "proved" or disproved.

Says that our intuitive notion of "algorithm" and "computation" is captured (in power) by TMs.

Evidence:

People have proposed "different" models of comp. "Different" ways to define functions algorithmically various extensions of TMs (as we've seen).

But in all cases they turn out to be no more powerful than TM.

λ-calculus, random access machines, marion algorithms, semi-Thue systems, general recursive functions, phrase-structure grammars...
**Example 1**

Random Access Machines (RAMs)

- Finite # of arithmetic registers
- Infinite # of available memory locations
- Instruction set (see below)

Initially, program instructions are written in contiguous block of memory, starting at location 1. All registers set to 0.

---

**Instruction Set**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD X, Y</td>
<td>Add contents of REG X and REG Y, putting result in REG X</td>
</tr>
<tr>
<td>LOAD C X, NUM</td>
<td>Place constant NUM in REG X</td>
</tr>
<tr>
<td>LOAD X, M</td>
<td>Put contents of mem loc. M into REG X</td>
</tr>
<tr>
<td>LOADI X, M</td>
<td>Indirect addressing - put value(value(M)) into REG X</td>
</tr>
<tr>
<td>STORE X, M</td>
<td>Copy contents of REG X into mem loc. M</td>
</tr>
<tr>
<td>JUMP X, M</td>
<td>IF contents(X) = 0, jump to instruction at memory location M; else next instruction is one following current</td>
</tr>
<tr>
<td>HALT</td>
<td>HALT.</td>
</tr>
</tbody>
</table>

(There are other versions besides one presented here)
A TM can simulate a RAM (sketch!)

TM maintains:

**Instruction-location tape**
- Stores memory location where next instruction to be executed is stored.
- Initially this tape contains "1", since program's first step is at memory loc. 1.

**Register tape**
- Stores register numbers and their contents, as follows: \# <reg. num>, <reg contents> \# ... etc.

Example: Suppose REG 1 contains 11 (=3)
REG 4 " 101 (=5)
All others empty.

Then REG tape has:

$\#1, 11\#100, 101\#$
MEMORY TAPE IS SIMILAR TO REGISTER TAPE

- To hold a number

  #<MEM LOCATION>, <MEM CONTENTS>#

  Binary String  Binary String

- To hold an instruction

  EXAMPLE: location 101 holds ADD 3, 6

  Then on memory tape we have

  ... #101, 30, 11, 11, 0 #

  Single symbol.

  Similarly for other instructions

WORK TAPES

Several (≤5) should suffice for carrying out computations during simulation.
TM BEGINS WITH BLANK REG. TAPE,
WITH "1" WRITTEN ON INSTR-LOC TAPE,
WITH PROGRAM STORED IN MEMORY
ON MEMORY TAPE AS ABOVE.

TM EXECUTES MANY TM STEPS FOR EACH
STEP OF RAM.

FOR EACH RAM STEP, IT:

- READS INSTR-LOC TAPE
- SEARCHES MEMORY TAPE FOR NEXT INSTR
  TO BE EXECUTED
- EXECUTES THE INSTRUCTION, CHANGING
  REGISTER, MEMORY TAPE AS NEEDED
- UPDATES INSTRUCTION LOCATION TAPE.

EXAMPLE .....

Suppose instruction location tape holds $101$ blank →

TM scans across memory tape, looking for pattern #101,

Indicates contents of mem loc 101 follow the comma

Suppose it finds #101, ADD, 111, 110#

- It sees ADD following comma, and
- Switches to a special state $q_{add}$ whose job is to handle the addition operation.
ADD, 11, 110 #

- Must first fetch arguments, which are in registers 11 and 110.

- Searches register tape for pattern #11, indicating that what follows is contents of reg 11, suppose it is 10110

- Write 10110 on work tape

- Now search for #110, on reg. tape, and copy second argument (say 11111) to work tape

- Add contents of worktape, obtaining 110101

- Go back to reg tape, look for reg #11, and replace string 10110 with new result 110101, shifting over if necessary, or to delete blank regions.

- Now add 1 to instr-loc. tape, and start all over.
MORE EVIDENCE FOR CHURCH'S THESIS:

DEFINITE "COMPUTABLE" FUNCTIONS MATHEMATICALLY.

THEY TURN OUT TO BE CAPTURED BY
TM-COMPUTABLE FUNCTIONS (PARTIAL REC. FNS)

WE'LL DESCRIBE THE "GENERAL RECURSIVE FNS"
AND SKETCH HOW THEY'RE EQUIV TO
THE PARTIAL REC FNS.

- FIRST WE START WITH A SIMPLER
  CLASS OF COMPUTABLE FNS, AND SEE
  WHY THEY FAIL TO CAPTURE ALL INTUITIVELY
  COMPUTABLE FNS,

- THEN WE EXTEND TO GEN. REC. FNS.
**PRIMITIVE RECURSIVE FUNS**

Defined inductively

\[ \text{FNS} : \mathbb{N} \rightarrow \mathbb{N} \]

1. **CONSTANT FUNCTIONS**
2. **SUCCESSOR FUNCTION**
3. **PROJECTION**
4. **COMPOSITION**
5. "**PRIMITIVE RECURSION**"

\[ \text{PRIM REC FUNS} = \text{ALL FNS OBTAINED BY FINITE & OF APPLICATIONS OF 1 - 5} \]

- **CAPTURE LARGE CLASS OF FUNCTIONS**, all of which are computable.
- **DO NOT CAPTURE ALL "COMPUTABLE" FUNCTIONS**
1. **CONSTANT FUNCTIONS**

   \[ \forall n \text{ the function } f(x_1, x_2, \ldots, x_n) = 0 \]
   (constant 0 function of n arguments)
   is primitive recursive

2. **SUCCESSOR FUNCTION** "S( )"

   \[ S(x) \] is a one-argument prim.rec. fn, where \( \forall x \in \mathbb{N}, \quad S(x) = x + 1 \)

3. **PROJECTION FUNCTIONS** "P_i^n"

   \( \forall i \leq i < n \) the function \( P_i^n \) is an n-argument primitive-recursive function defined by
   \[ P_i^n(x_1, x_2, \ldots, x_n) = x_i \]
   - picks the \( i^{th} \) argument out of \( n \)
   - \( P_i^1(x) = x \) identity function
4. Composition

\[ \forall m \forall n \]

**IF** \[ g(x_1, x_2, \ldots, x_m) \] **IS A PRIM. REC. FUNCTION OF** \( m \) **ARGS**

**AND**

\[ h_1(x_1, x_2, \ldots, x_n), h_2(x_1, x_2, \ldots, x_n), \ldots, h_m(x_1, x_2, \ldots, x_n) \]

**ARE PRIM. REC. FNS OF** \( n \) **ARGS**

**THEN** \[ g(h_1(x_1, x_2, \ldots, x_n), h_2(x_1, x_2, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n)) \]

**IS A PRIM. REC. FUNCTION OF**

\( n \) **ARGUMENTS**
\[ f(x_1, x_2, \ldots, x_{n-1}, x_n) = \begin{cases} 
  g(x_1, x_2, \ldots, x_{n-1}) & \text{if } x_n = 0 \\
  h(x_1, x_2, \ldots, x_{n-1}, x_n - 1, f(x_1, \ldots, x_{n-1}, x_n - 1)) & \text{otherwise} 
\end{cases} \]
5 "Primitive Recursion"

\[ \forall n \text{ if } g(x_1, x_2, \ldots, x_{n-1}) \text{ a prim rec fn of } n-1 \text{ args} \]

\[ h(x_1, \ldots, x_{n+1}) \text{ a prim rec fn of } n+1 \text{ args} \]

Then \( f \) is a prim rec. fn of \( n \) args defined by:

\[
f(x_1, x_2, \ldots, x_{n-1}, x_n) = \begin{cases} 
g(x_1, x_2, \ldots, x_{n-1}) & \text{if } x_n = 0 \\
h(x_1, x_2, \ldots, x_{n-1}, x_n - 1, f(x_1, x_2, \ldots, x_n - 1)) & \text{otherwise} \end{cases}
\]

**Example:** Let \( n = 1 \). Then \( g \) is constant \((0-\text{arg})\), \( h \) is 2-arg.

\[
f(x) = \begin{cases} 
g & \text{if } x = 0 \\
h(x-1, f(x-1)) & \text{otherwise} \end{cases}
\]

\( g \) is base case
\( h \) describes how \( f \) depends on \( f \).
Recursion on Last Argument $x_n$

Base case (where recursion bottoms out; $x_n = 0$)

Recursion: * value of $f$ based on value of $f$
on smaller arg $x_{n-1}$
plus other args $x_1, x_2, \ldots, x_{n-1}$

$h$ describes how $f$ depends on $f$
Recursion Examples.

Factorial \( (n) \)

\[
= \begin{cases} 
1 & \text{if } n = 1 \\
 n \cdot f(n-1) & \text{otherwise}
\end{cases}
\]

\[
F(n) = \begin{cases} 
g() & \text{if } n = 0 \\
h\left(n, f(n-1)\right) & \text{one less argument than } F
\end{cases}
\]
Example

\[ + (x_1, x_2) ::= x_1 + x_2 \]

is a primitive recursive function

Idea - use rule (5): Primitive recursion

Intuitively

\[ + (x_1, x_2) = 1 + + (x_1, x_2 - 1) \]

\[ = S (+ (x_1, x_2 - 1)) \]

But must follow format correctly:

\[ + (x_1, x_2) = \begin{cases} 
  g(x_1) & \text{if } x_2 = 0 \\
  h(x_1, x_2 - 1, + (x_1, x_2 - 1)) & \text{otherwise}
\end{cases} \]

Clearly this should be \( x_1 \), so \( g \) is identity, or \( p_1 \)

This should be \( 1 + \) third arg, or successor applied to third arg.

So \( h \) is \( S(p_3^3()) \)

\[ + (x_1, x_2) = \begin{cases} 
  p_1'(x_1) & \text{if } x_2 = 0 \\
  S(p_3^3(x_1, x_2 - 1, + (x_1, x_2 - 1))) & \text{otherwise}
\end{cases} \]
**Example:** \( \text{DOUBLE}(x) = 2x = x + x \) is primitive recursive.

**Idea:** \( \text{DOUBLE}(x) = \text{+(}(x, x)) \), so are we done?

What rule was used to do this?

There is no "copy argument" rule.

**Solution:** use \( P_1'(x) \) to provide "copies" of \( x \)

**Correct Definition:**

\[ \text{DOUBLE}(x) = \text{+(}(P_1'(x), P_1'(x))) \]

**Proof:** \( \text{+(}, \text{)} \) is prim rec fn of 2 args

\( P_1'(\cdot) \) \( P_1'(\cdot) \) are 2 prim rec funs of 1 arg

So composition gives prim rec fn of 1 arg

\[ \text{+(}(P_1'(\cdot), P_1'(\cdot)) \]
FACT 1

ALL PRIM. REC. FNS. ARE TOTAL
(i.e., DEFINED & ARGUMENTS)

PROOF

INDUCTION ON DEE OF PRIM. REC. FN.

BASE FUNCTIONS \{ constants are total \}
\{ projection are total \}
\{ successor is total \}

* composition of total fns yields total fns

* prim. recursion applied to total fns yields total fns

FACT 2

CAN "CODE" PRIM. REC. FNS IN A SYSTEMATIC WAY AS STRINGS
\( w_1, w_2, w_3, \ldots \) SUCH THAT:

* each prim. rec fn is some \( w_i \)
* \exists algorithm that on input \( i \) outputs \( w_i \)
* \exists algorithm that on input \( w_i, x \) outputs \( f(x) \) where \( f \) is function represented by \( w_i \)
Illustration of how we might encode Prim. Rec. Fns.

Write down, line-by-line, how the base Fns and composition, Prim-Rec operators, are applied in the def. of the function to be encoded.

Example: \( \text{double}(x) = +(p_1'(x), p_1'(x)) \)

where

\[+(x_1, x_2) = \begin{cases} 
p_1'(x_1) & \text{if } x_2 = 0 \\
S(p_3^g(x_1, x_2-1, +(x_1, x_2-1))) & \text{otherwise}
\end{cases}\]

<table>
<thead>
<tr>
<th>STRING</th>
<th>FUNCTION DEFINED</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. #1 #1</td>
<td>( p_1'(x) )</td>
</tr>
<tr>
<td>2. #3 #3</td>
<td>( p_3^g(x) )</td>
</tr>
<tr>
<td>3. S</td>
<td>( S(x) )</td>
</tr>
<tr>
<td>4. COMP # line 3 # line 2</td>
<td>( S(p_3^g( )) )</td>
</tr>
<tr>
<td>5. \text{PrimRec} #2 # line 1 # line 4</td>
<td>( +(x_1, x_2) )</td>
</tr>
<tr>
<td>6. COMP # line 5 # line 1 # line 1</td>
<td>( \text{double}(x) )</td>
</tr>
</tbody>
</table>
ILLUSTRATION OF HOW WE MIGHT ENCODE PRIM. REC. FNS.

WRITE DOWN, LINE-BY-LINE, HOW THE BASE FNS AND COMPOSITION, PRIM-REC OPERATORS, ARE APPLIED IN THE DEF. OF THE FUNCTION TO BE ENCODED.

EXAMPLE: \( \text{DOUBLE}(x) = + (p'_1(x), p'_4(x)) \)

where \( + (x, x_2) = \begin{cases} 
    p'_1(x) & \text{if } x_2 = 0 \\
    S(p_3^3(x, x_2-1, + (x, x_2-1))) & \text{otherwise}
\end{cases} \)

**STRING** | **FUNCTION DEFINED**
---|---
1. P#1#1 | \( p'_1(x) \)
2. P#3#3 | \( p'_3(x) \)
3. S | \( S(x) \)
4. COMP # line 3 # line 2 | \( S(p_3^3(\text{COMP} \uparrow \text{line 3} \uparrow \text{line 2})) \)
5. PRIMREC # line 2 # line 1 # line 4 | \( + (x, x_2) \)
6. COMP # line 5 # line 1 # line 1 | \( \text{DOUBLE}(x) \)

1. P#1#1, 2. P#3#3, 3. S, 4. COMP # line 3 # line 2, 5. PRIMREC # line 2 # line 4, 6. COMP # line 5 # line 1 # line 1

THIS STRING REPRESENTS "DOUBLE(\(x\))"

ARGUE:
- EACH PRIM REC FN CAN BE REP AS SOME STRING \(w_i\);
- \(\exists\) ALG THAT ON INPUT \(i\) OUTPUTS \(w_i\);
- \(\exists\) ALG THAT ON INPUT \(w_i \downarrow x\) OUTPUTS \(f(x)\) WHERE \(w_i \uparrow f\)
FACT 1 (ALL PRIM REC FNS TOTAL)

FACT 2 (CAN ON INPUT i, x OUTPUT THE VALUE OF fi(x) WHERE fi IS THE "ith" PRIM REC FN)

PRIM REC FNS DO NOT CAPTURE ALL "COMPUTABLE" FNS

PROOF: DIAGONALIZATION

ALGORITHM A

INPUT (i)
GENERATE W_i = DESCRIPTION OF i-th PRIM. REC FUNCTION f_i
FROM W_i, COMPUTE f_i(i) (DEFINED, MUST HALT)
OUTPUT f_i(i) + 1

- A IS AN ALGORITHM COMPUTING SOME TOTAL FUNCTION
- A(i) ≠ f_i(i) FOR ANY PRIM REC FN f_i, SINCE
  A(i) ≠ f_i(i) [A DIFFERS FROM i-th PRIM REC FN ON ARGUMENT i]
THE BAD NEWS APPLIES TO ANY COLLECTION OF FUNCTIONS $F$ SUCH THAT

1. EVERY $f \in F$ IS TOTAL
2. 3 FINITE LENGTH DESCRIPTION OF EACH $f \in F$ USING SOME FINITE ALPHABET,
3. WE CAN DISTINGUISH BETWEEN VALID DESCRIPTIONS AND INVALID ONES
   [HENCE CAN FIND $i^{th}$ DESCRIPTION BY ENUMERATING IN LEX. ORDER AND DROPPING OUT NONSENSE DESCRIPTIONS]
4. GIVEN DESCRIPTION FOR $f$, AND GIVEN $x$, CAN COMPUTE $f(x)$.

I.E., THE DESCRIPTIONS OF THE FUNCTIONS TELL US HOW TO COMPUTE THEM.

- ANY COLLECTION OF FUNCTIONS SATISFYING 1-4 DON'T CAPTURE ALL "COMPUTABLE" FUNCTIONS, VIA DIAGONALIZATION ARGUMENT WE JUST SAW.
- TO CAPTURE ALL "COMPUTABLE" FUNCTIONS, WE CAN'T INSIST ON 1-4.
"GENERAL RECURSIVE FUNCTIONS"

1. Constant Functions
2. Successor Function
3. Projection Functions
4. Composition
5. Primitive Recursion

PRIM REC FNS

6. Minimization

[Search for least value...
  Allows potentially nonterminating search]

If \( f(x_1, \ldots, x_n) \) is a gen rec fn of \( n \) args

then \( \mu f(x_1, \ldots, x_{n-1}) \) is a gen rec fn of \( n-1 \) args

defined by

\[
\mu f(x_1, \ldots, x_{n-1}) = \min \left\{ x_n : f(x_1, \ldots, x_n) = 0 \text{ and } \forall x < x_n \quad f(x_1, \ldots, x_{n-1}, x) \text{ defined } \neq 0 \right\}
\]

(IF NO SUCH \( x_n \) EXISTS, THEN \( \mu f(x_1, \ldots, x_{n-1}) \) IS UNDEFINED)

Gen. rec. FNS. ARE ALL ONLY THOSE DEFINED
BY FinitE & APPLICATIONS OF 1-6
"General Recursive Functions"

1. Constant Functions
2. Successor Function
3. Projection Functions
4. Composition \( y \left( h(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n) \right) \)
5. Primitive Recursion

Prim rec. Fns. Undefined if any of \( h_i(x_1, \ldots, x_n) \) are undefined.

6. Minimization

[Search for least value. In def. not def. allows potentially nonterminating search.]

If \( f(x_1, \ldots, x_n) \) is a gen. rec. fn. of \( n \) args,
then \( \mu f(x_1, \ldots, x_{n-1}) \) is a gen. rec. fn. of \( n-1 \) args
defined by

\[
\mu f(x_1, \ldots, x_{n-1}) = \min \left\{ x_n : f(x_1, \ldots, x_n) = 0 \text{ and } \forall x < x_n f(x_1, \ldots, x_{n-1}, x) \text{ defined } \neq 0 \right\}
\]

(If no such \( x_n \) exists, then \( \mu f(x_1, \ldots, x_{n-1}) \) is undefined.)

Gen. rec. Fns. are all / only those defined by finite \& applications of 1–6.
THEOREM \:$\{\text{partial recursive \ fns}\}^3 = \{\text{general rec. \ fns}\}^3$

Thus \: \text{general rec. \ fns} = \text{tm- \ computable \ fns}.

\begin{proof}
\begin{enumerate}
\item Sketch (Sketch (Sketch (Sketch (Sketch (Proof)))))
\item Must show every function built from finite # applications of \(1\) - \(6\) is computable by a TM.
\end{enumerate}
\end{proof}

\textbf{Induction on # rules used in definition}

- \textbf{Base case}
  - Constant
  - Successor \(\Rightarrow\) Clearly
  - Projection \(\Rightarrow\) TM-computable

- \textbf{If} \(g(\ )\) TM-computable and \(h_1(\), \ldots, \(h_m(\)
  - TM-computable, \textbf{then} \(g(h_1(\), \ldots, \(h_m(\))
  - TM-computable via subroutine calls

- Similarly for primitive recursion (\(\Rightarrow\) )

\(\mu f(\) is compatible with TM \textbf{if} \(f\) is,
  \text{by simply doing search} \(f(4), f(2), \ldots, f(x), \ldots\)
  \text{for least} \(x\) s.t. \(f(x) = 0, f(i) \neq 0 \ \forall i < x\).
Let \( M \) be a TM. We argue that the function \( M() \) computed by \( M \) can be defined using (1)-(6) as a Gen.Rec.Fn.

\[ \rightarrow \text{Hold on to your hats...} \]

1. W.L.O.G., modify \( M \) so even after it halts it has a "next configuration", which is the same as its halting config.

   Thus, if \( M(q_i) = q_j \), then

   \[ q_0q_i \xrightarrow{\star} q_{halt}q_i 1_{halt}q_i 1_{halt}q_i 1_{halt} \]

2. Each ID is a number in base \( |Q| + |Z U \{0,1\}| \)

3. If \( X \) is an ID, it is possible to "pick out" the state of \( M \) that appears in ID \( X \) by appropriate use of \( \text{div}, \text{mod} \) functions

   \[ X = X_1 X_2 X_3 \ldots X_i q X_{i+1} \ldots X_n \]

   Each "div" by base chops off a trailing symbol - do this until "mod" reveals last symbol is a state symbol.
4. CLAIM (BY PREV COMMENT)

3. GENERAL REC. FN

\[ \text{FINAL}(X) = \begin{cases} 1 & \text{if } X \text{ is of form } q_{\text{HALT}}0^* \\ 0 & \text{otherwise} \end{cases} \]

5. 3 GEN REC FN GIVING NEXT ID OF M:

\[ N_M(X) = Y \quad \text{WHERE} \quad X \rightarrow_m Y \]

INTUITION ... X, Y ARE ALMOST IDENTICAL EXCEPT FOR LOCAL DIFFERENCES NEAR READ/ WRITE HEAD.

A GEN. REC. FN CAN FIND THE "q" SYMBOL, THEN MAKE LOCAL CHANGES AS DICTATED BY M, THEN OUTPUT THE NEW ID.

ALL OF THIS IS DONE WITH DIV, MOD, AND SIMPLE COMPARISONS.

6. GEN REC FN RETURNING THE lTH ID AFTER 10 X:

\[ f_M(x,i) = \begin{cases} p_i(x) & \text{if } i = 0 \\ N_M(p^3_3(x,i-1, f_M(x,i-1))) & \text{otherwise} \end{cases} \]
7. \exists \text{ GEN REC FUNCTION } g(x, i) \text{ THAT EVALUATES TO } 0 \text{ IFF THE } i^{\text{th}} \text{ ID AFTER } x \text{ IS A HALTING-DONE-Computing ID ... \THUS}

\[ g(x, i) = 0 \text{ IFF: } \]

\[ \text{FINAL}(f_{m}(x, i)) \]

8. \text{ GEN. REC. FN TO DETERMINE } \# \text{ OF STEPS FOR } M \text{ TO DO ENTIRE COMPUTATION:}

\[ h(x) = \mu g(x, i) = \text{LEAST } i \text{ (# STEPS) SUCH THAT } g(x, i) = 0, \text{ OR ... SUCH THAT } M \text{ REACHES HALTING ID.} \]

\text{\THUS}

\[ h(q_00^n) \]

\text{GIVES THE } \# \text{ OF STEPS THAT } M \text{ TAKES TO COMPUTE ON INPUT } 0^n \text{ (IF } M(0^n) \uparrow \text{ THEN } h(q_00^n) \text{ UNDEFINED)
9. The final ID of \( M \) is thus:

\[
\text{\#}_{m}(90^0, h(90^0))
\]

\[
\text{\#}_{m}(x, i) = \text{\#}^t \text{ ID after } 10^x
\]

This gives the \( i \) such that
\( i^t \text{ ID after } 90^0 \) is
A "Done Computing" ID

10. \( \exists \text{ Gen Rec Fn (using Div, Mod Functions)} \)

\( \text{called Val such that} \)

\[
\text{Val}(9^{\text{halt}} i^j) = 0^i
\]

Thus Val erases the "9halt"

11. \( M(0^0) = \)

\[
\text{Val}(\text{\#}_{m}(90^0, h(90^0)))
\]

Pick Value From Final ID

Thus \( M(\cdot) \) can be expressed as a general rec function

Partial Rec Fns = TM-Computable Fns < Gen Rec Fns
TMs AS GENERATORS

\[ M: \]

- Starts with all tapes blank
- From time to time writes some word \( w_i \) on output tape (followed by \#)

\[ M \text{ generates } w \text{ iff at some time } "\#w\#" \text{ appears on output tape} \]

\[ (M \text{ need not generate words in order, } M \text{ might generate some words more than once}) \]

\[ G(M) = \{ w : M \text{ generates } w^2 \} \]
**Theorem**

$L$ is recursively enumerable (accepted by some TM)

$\iff$

$\exists M \ L = G(M)$

**Proof**

$(\Leftarrow)$

Given generator $M$ for $L$, construct acceptor $M'$ for $L$

If $M$ ever writes $\#w\#$ on output tape, then $M'$ halts & accepts $w$
(⇒) GIVEN ACCEPTOR M CONSTRUCT GENERATOR M'

PRINT #

FOR w = ε, 0, 1, 00, 01, 10, 11, ... (LEXICOGRAPHIC ORDER OF ALL WORDS IN Z*)

DO BEGIN

FEED w TO M

IF M ACCEPTS w THEN PRINT w ON OUTPUT TAPE

END FOR
(⇒) Given acceptor M, construct generator M'.

Wrong

\[\text{For } w = \epsilon, 0, 1, 00, 01, 10, 11, \ldots \] (lexicographic order of all words in \(\Sigma^*\))

Do Begin

\[\text{Feed } w \text{ to } M\]

If M accepts w then print w on output tape

End

End For

If M doesn't accept by getting in \(\epsilon\) loop, we never generate any other w's.
A CORRECT PROOF REQUIRES "DOVETAILING"

- A METHOD FOR CARRYING OUT AN \( \infty \) # OF COMPUTATIONS "IN PARALLEL".
  (ACTUALLY A SCHEDULING TRICK)

METHOD #1

\[ \text{Compute } M(w_1), M(w_2), M(w_3) \ldots \text{ in parallel.} \]

1. RUN \( M(w_1) \) FOR 1 STEP
2. RUN \( M(w_1) \) FOR 1 STEP MORE
   RUN \( M(w_2) \) FOR 1 STEP
3. RUN \( M(w_1) \) FOR 1 STEP MORE
   RUN \( M(w_2) \) FOR 1 STEP MORE
   RUN \( M(w_3) \) FOR 1 STEP
   
   \vdots

\begin{align*}
\text{If } & \text{ } M(w_i) \text{ accepts at some point, then output } \# w_i \# \\
\end{align*}

\[ \text{This schedule allocates } \infty \text{ steps to } M(w_i) \text{ for each } w_i. \]
PAIRS OF NATURAL #3's CAN BE ENUMERATED IN SOME ORDER, SINCE $\mathbb{N} \times \mathbb{N} \approx \mathbb{N}$

\[
\langle c_{ij} \rangle = \frac{(i+j-1)(i+j-2)}{2} + i
\]

MAPS $\mathbb{N} \times \mathbb{N} \leftrightarrow \mathbb{N}$

(WHAT IS INVERSE MAP?)

\[ f_i(k) = i \text{ s.t. } \exists j : \langle c_{ij} \rangle = k \]

**GENERATOR M!**

```
FOR k = 1 TO 00 DO
    FIND \langle c_{ij} \rangle s.t. \langle c_{ij} \rangle \leftrightarrow k
    RUN M(w_i) FOR j STEPS
    IF M(w_i) HALTS, ACCEPTS IN = j STEPS
    THEN OUTPUT #w_i#
    ELSE NEXT k
```
THEOREM

\[ L \text{ recursive } \iff \exists \text{ TM that generates } L \text{ in canonical order} \]

PROOF

(\(\Rightarrow\)) Let \(M\) be TM that accepts \(L\) and \(M\) halts on all inputs.

Construct generator \(M'\):

\[
M' \quad \text{for } i := 1 \text{ to } \infty \text{ do}
\text{begin}
\text{run } M(w_i)
\text{if } M\text{ accepts } w_i \text{ then output } \#w_i
\text{end}
\]

Since \(M\) always halts, \(M'\) never gets stuck. It tests each string \(w_1, w_2, w_3, \ldots\) in order and outputs only those that \(M\) accepts.
Suppose \( M' \) generates \( L \) infinite in canonical order.

Construct \( M \) accepting \( L \).

\( M \) must halt for all inputs.

Run \( M' \) until it either generates \( w \) (in which case, accept) or until it generates some \( w' > w \) without having generated \( w \) (in which case, reject).

Construction only works if \( L \) is infinite - else we might never see some \( w' > w \), hence not know that we should reject.

Suppose \( M' \) generates \( L \) finite in canonical order.

Then clearly \( \exists \) halting \( M \) for \( L \) : \( L \) is regular!

Thus, regardless if \( L \) finite or infinite, \( \exists \) \( M \) halting.

Note: proof is nonconstructive.
RECURSIVE

"DECIDABLE"
"TM-DECIDABLE"

\[ W \rightarrow M \]

YES if \( w \in L(M) \)
NO if \( w \notin L(M) \)

RECURSIVELY ENUMERABLE (R.E.)

"TM-ACCEPTABLE"
"SEMI-DECIDABLE"

\[ W \rightarrow M \]

YES if \( w \in L(M) \)

M need not tell us "NO" if \( w \notin L(M) \)
WLOG, assume it never says "NO"
THEOREM

RECURSIVE LANGS CLOSED UNDER COMPLEMENT

PROOF: GIVEN $M$ ACCEPTING $L$ (ALWAYS HALTING)

CONSTRUCT $\overline{M}$ ACCEPTING $\overline{L}$ (ALWAYS HALTING)
THEOREM

RECURSIVE LANGS CLOSED UNDER COMPLEMENT

PROOF: GIVEN $M$ ACCEPTING $L$ (ALWAYS HALTING)
CONSTRUCT $\overline{M}$ ACCEPTING $\overline{L}$ (ALWAYS HALTING)
THEOREM

RECURSIVE LANGUAGES CLOSED
UNDER UNION AND INTERSECTION

GIVEN

$M_1 \xrightarrow{w} \begin{cases} \text{YES if } w \in L_1 \\ \text{NO if } w \notin L_1 \end{cases}$

$M_2 \xrightarrow{w} \begin{cases} \text{YES if } w \in L_2 \\ \text{NO if } w \notin L_2 \end{cases}$

CONSTRUCT

$M$ ACCEPTING $L_1 \cup L_2$

\[ \begin{array}{c}
\text{YES} \\
\text{w} \in L_1 \cup L_2 \\
\text{NO} \\
\text{w} \notin L_1 \cup L_2 \\
\end{array} \]
THEOREM

RECURSIVE LANGUAGES CLOSED

UNDER UNION AND INTERSECTION

GIVEN

\[ M_1 \]

YES if \( w \in L_1 \)

NO if \( w \notin L_1 \)

\[ M_2 \]

YES if \( w \in L_2 \)

NO if \( w \notin L_2 \)

CONSTRUCT

\( M \) ACCEPTING \( L \cup U L_2 \)

[Diagram showing the construction scheme]
THEOREM

RECURSIVE LANGUAGES CLOSED
UNDER UNION AND INTERSECTION

Given:

- $M_1$: Yes if $w \in L_1$, No if $w \notin L_1$.
- $M_2$: Yes if $w \in L_2$, No if $w \notin L_2$.

Construct $M$ accepting $L_1 \cup L_2$ and $L_1 \cap L_2$.

Alternatively, since rec langs closed under $\cup$ and complement, DeMorgan $\Rightarrow$ closed under $\cap$. 
THEOREM

RECURSIVELY ENUMERABLE LANGUAGES ARE CLOSED UNDER UNION INTERSECTION

Q: DOES PROOF SHOWING RECURSIVE CASE HOLD HERE?

A: NO

\[
\text{RUN } M_1 \text{ ON } W. \text{ IF IT ACCEPTS, THEN}
\]
\[
\text{RUN } M_2 \text{ ON } W. \text{ IF IT ACCEPTS, THEN ACCEPT}
\]
\[
\text{THIS ACCEPTS } L(M_1) \cup L(M_2)
\]

\[
\text{RUN } M_1 \text{ ON } W. \text{ IF IT ACCEPTS THEN ACCEPT}
\]
\[
\text{IF IT REJECTS THEN RUN } M_2 \text{ ON } W
\]
\[
\text{IF IT DO LOOP... ??}
\]

\[
\text{WON'T ACCEPT } W \in L(M_2) - L(M_1)
\]
\[
\text{WHEN } M_1(W) \uparrow
\]

SOLUTION FOR UNION:

\[
\begin{array}{ccc}
M_1 & \rightarrow & YES \\
\downarrow & & \\
M_2 & & \rightarrow \text{YES}
\end{array}
\]

\[
\text{YES \ IF EITHER}
\]
\[
W \in L(M_1) \text{ OR } W \in L(M_2) \text{ (OR BOTH)}
\]

DEVETAIL (ALTERNATE STEPS IN BOTH COMPUTATIONS) OR NONDETERMINISM (FINITE & CHOICES)
THEOREM

L RECURSIVE ⇔ L AND L BOTH R.E.

PROOF

(⇒)

L RECURSIVE

L R.E.

(⇐)

GIVEN M ACCEPTING L

M ACCEPTING L

M

w \rightarrow YES w \in L

M

w \rightarrow YES w \in \overline{L}
Comments:

THM showed $L \text{ REC } \iff L \subseteq \text{ R.E.}$

Thus

- $L \text{ not REC. ENUM } \iff L \text{ not REC.}$
- $L \text{ not REC. ENUM } \iff \overline{L} \text{ not REC.}$
- If $L$ is REC ENUM but not REC. then $\overline{L}$ is not REC ENUM.

[So to show R.E. not closed under complement, sufficient to exhibit $L \subseteq \text{ R.E.} - \text{ REC.}$]
DECISION PROBLEMS
AND DECIDABILITY

DECISION PROBLEM: A QUESTION (OVER MANY INSTANCES) WITH A YES-NO ANSWER

- GIVEN GRAMMAR G, IS G AMBIGUOUS?
- GIVEN A TM M, DOES L(M) = Ε*?
- GIVEN DFAs M₁, M₂, IS L(M₁) = L(M₂)?
- GIVEN GRAPH G, IS G CONNECTED?

CAN BE REPRESENTED AS A SET (OR LANGUAGE)

- \langle \langle G \rangle \rangle: \text{"\langle G \rangle" IS A STRING Encoding A Grammar G S.T. G IS AMBIGUOUS.}
- \langle \langle M \rangle \rangle: \text{"\langle M \rangle" Encodes A TM M SUCH THAT L(M) = Ε*}
- \langle \langle M₁, M₂ \rangle \rangle: \text{M₁, M₂ ARE DFAs, WITH L(M₁) = L(M₂)}
- \langle \langle G \rangle \rangle: \text{"\langle G \rangle" IS A STRING Encoding A CONNECTED GRAPH.}

WE SEEK AN ALGORITHM TO SOLVE THE PROBLEM FOR ANY POSSIBLE INPUT. ALWAYS HALTS
A (decision) problem is **decidable**

if ∃ algorithm that ∀ instances (inputs)
says either "yes" or "no" (and is correct)

thus a problem is **decidable**

associated language is **recursive**

**Note**

since finite languages are all recursive,
any decision problem with only a finite number of instances is **decidable**.

Examples...
Example 1

Problem: Does a substring of exactly 83 consecutive "7"s in $n$?

"Decidable"

One of these input-less algorithms outputs the correct answer.

Too bad we don't know which algorithm is the correct one.

Moral: This is nonsense. The original problem has no "instances," it only asks a single yes/no question. We are considering problems that are questions ranging over many possible inputs.

Not even clear what language corresponds to the above "problem."
**Example 2**  (let's try again)

**Problem:** Given $n$, find a substring of exactly $n$ consecutive "7"'s in decimal expansion of $\pi$?

**Corresponding Language:**

\[ \exists n \in \mathbb{N} : \text{decimal expansion of } \pi \text{ contains the substring } a_{7777...76}^n \]

where $a, b \neq 7$

**Is this language recursive?**

**Can we decide the above problem?**
EXAMPLE 3

Q: GIVEN \( n \), \( \exists \) SUBSTRING OF \( \geq n \) CONSECUTIVE "7"'S IN \( \pi \)?

LANGUAGE:

\[ L = \{ n \in \mathbb{N} : \text{DECIMAL EXP OF } \pi \text{ CONTAINS} \underbrace{7777\ldots}_n \} \]

Q: IS THIS DECIDABLE (RECURSIVE)?

A: IT IS REGULAR !!

\( L \) IS EITHER ALL OF \( \mathbb{N} \), OR ELSE \( \exists \) \( n_0 \) \( L = \{ 1, 2, 3, \ldots, n_0 \} \)
CODING OF TMs

- Show how to associate each TM with a number in a nice way. Then can treat programs (TMs) as data (numbers).

**Lemma [Hu Thm 7.10]**

If \( L \subseteq \Sigma^* \) is accepted by some TM, then \( \exists \) TM with \( \Sigma = \{0,1\}, \Gamma = \{0,1,\#\} \) that accepts \( L \).

Now code any TM with \( \Sigma = \{0,1\}, \Gamma = \{0,1,\#\} \) as binary string.

WLOG assume:

- States numbered \( 1, 2, 3, \ldots, k \) for some \( k \)
- 91. Unique initial state
- 92. Unique halt/accept state
- 93. Unique halt/reject state

To code a TM, we list its transitions (its states are implicitly listed, and we need not specify initial state, halt states, by our convention).
Transitions are listed in the following order, omitting any that are undefined:

$\delta(q_0, 0), \delta(q_1, 1), \delta(q_1, B), \delta(q_2, 0), \delta(q_2, 1), \delta(q_2, B), \ldots$

$\ldots \delta(q_k, 0), \delta(q_k, 1), \delta(q_k, B)$

The encoding:

```
111 111 111 111 -- 111 111
1st transition 2nd transition
```

Where each transition

$\delta(q_i, \{0, 1\}) = (q_j, \{0, 1\}, \{L, R\})$

is encoded by

$0^i \{\{00\} \{00\}\} 0^j \{\{00\} \{00\}\} \{00\} \{00\}$

Thus $\delta(q_3, 1) = (q_2, 0, 5)$ becomes:

$000100100101000$
Clearly every TM \( \rightarrow \) some \( n \in \mathbb{N} \)

View it as one-to-one mapping:
If \( n \in \mathbb{N} \) doesn't syntactically correspond to any TM description,
we'll let \( n \) represent the "null TM" that accepts \( \emptyset \).
By looping on all inputs.

\( \mathbb{N} = \) Turing machine descriptions

Let \( \langle M \rangle = n \) such that \( n \) encodes \( M \).

We write \( M_i \) to indicate \( i^{th} \) TM, that is, \( M_i \) is the \( M \) such that \( \langle M \rangle = i \).
A non recursively-enumerable language

\[ L_d = \{ \text{"DIAGONAL LANGUAGE"} \} = \{ i : i \notin L(M_i) \} \]

<table>
<thead>
<tr>
<th>Inputs</th>
<th>( 1 )</th>
<th>( 2 )</th>
<th>( 3 )</th>
<th>( 4 )</th>
<th>( 5 )</th>
<th>( 6 )</th>
<th>( 7 )</th>
<th>( \ldots )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>2</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>3</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>4</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Turing Machines

(Strings corr to binary #'s)

\( 7 \notin L(M_2) \)
A non recursively-enumerable language

\[ L_d = \text{"DIAGONAL LANGUAGE"} \]
\[ = \{i \in \mathbb{N} : i \notin L(M_i)\} \]

<table>
<thead>
<tr>
<th>INPUTS</th>
<th>(STRINGS CORR TO BINARY #'S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 ...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Turing Machines</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Y Y Y Y N N Y Y</td>
</tr>
<tr>
<td>2 N N Y Y N Y Y N</td>
</tr>
<tr>
<td>3 N Y N N Y N Y Y</td>
</tr>
<tr>
<td>4 Y N N N Y Y Y N</td>
</tr>
<tr>
<td>5 Y Y Y Y Y Y N N</td>
</tr>
<tr>
<td>6 ; ; ; ; ; ; ; ;</td>
</tr>
</tbody>
</table>

\[ L_d \text{ is not rec. enum.} \]
\[ \text{If it were, then } L_d = L(M_j) \text{ for some } j \in \mathbb{N} \]
\[ \text{but } j \in L_d \iff j \notin L(M_j) \]
But doesn't following accept $L_d$?

$M_{ld}:

ON INPUT $i$
GENERATE TM $M_i$ DESCRIPTION
RUN $M_i$ ON INPUT $i$
OUTPUT YES IF $M_i(i)$ OUTPUTS NO
OUTPUT NO IF $M_{ld}(i)$ OUTPUTS YES

Does $M_{ld}$ accept $L_d$?

Why not?
\[ L_u = \{ (M)\#w : M \text{ accepts } w \} \text{ is } R.E. \]

We describe \( M_u \) that on input \( (M)\#w \)

- Simulates \( M(w) \) and accepts if \( M \) does

* Similar to \texttt{SINGLE TM} that simulates an arbitrary RAM program

* \exists \texttt{SINGLE TM} \( M_u \) that functions as stored PGM

  \( M_u \) is \textit{the embodiment of "computer"}
HOW M U WORKS (3 TAPES)

| $ | t_1 | t_2 | t_3 | ... | $ | \# | W |

\( \langle m \rangle \) = CODE FOR M
\( t_i \) = CODE FOR TRANSITION i

TAPE 1 NEVER CHANGES

1. USING TAPE 3 AS WORKSPACE, M U CHECKS THAT \( \langle m \rangle \) IS VALID TM CODE.

   THUS IT CHECKS, FOR EXAMPLE, THAT

   • SUBSTRING 110^i 10^i DOESN'T APPEAR TWICE
     (INDICATING NONDETERMINISM)

   • NO FOUR 1's

   • THREE 1's AT BEGINNING, END ONLY

   • APPROPRIATE # OF 0's BETWEEN 1's IN TRANSITION CODES

   11000010100000100001...-

   \( S(q_4, 0) = (q_5, \uparrow \text{NOT A VALID CHAR. TO WRITE} \)
2. Assuming $<M>$ is OK, $M_U$ copies $W$ to Tape 2.

3. $M_U$ writes "0" on Tape 3. This indicates $M$ is in initial state $q_i$. (Tape 3 holds $0^i$ indicating state is $q_i$.) If ever Tape 3 holds $0^i = q_2 = \text{HALT/ACCEPT}$ then $M_U$ halts and accepts.

**Tape 1**
$$\begin{array}{c}
\$ \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \\
\end{array}$$

**Tape 2**
$$\begin{array}{c}
\$ \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \\
\end{array}$$

**Tape 3**
$$\begin{array}{c}
\$ \ 0 \\
\end{array}$$

4. If Tape 3 holds $0^i$ and head on Tape 2 is scanning symbol "1" then $M_U$ looks for $10^i100\ldots$ on Tape 1. If not found, then halt/reject. Else $M_U$ finds $10^i10010100101001$ (for example) so $M_U$ writes $0^i$ on Tape 3, replaces 1 with 0 on Tape 2, and moves Tape 2 head to right.
1. Assuming \( M \) is OK, \( M_0 \) copies \( W \) to tape 2.

2. \( M_0 \) writes "0" on tape 3. This indicates \( M \) is in initial state \( q_1 \). (Tape 3 holds \( q_0 \) indicating state is \( q_j \)). If ever tape 3 holds \( 00 = q_2 = \text{Halt/Accept} \) then \( M_0 \) halts and accepts.

3. **Example:** \( M_0 \) looks for \( 110|001 \) on tape 1.

   - **Tape 1:**
     
     \[
     \$ 1110|010|000|0|00|000|0|001|\ldots
     \]

   - **Tape 2:**
     
     \[
     \$ 1011|10|001|11101
     \]

   - **Tape 3:**
     
     \[
     \$ 1
     \]

   - **Current state of \( M \):**

   - **Current contents of \( M \)'s tape:**
2. Assuming \( <M> \) is OK, \( M_0 \) copies \( W \) to Tape 2.

3. \( M_0 \) writes "0" on Tape 3. This indicates \( M \) is in initial state \( q_1 \). (Tape 3 holds \( Q \) indicating state is \( q_1 \)). If ever Tape 3 holds \( 00 = q_2 = \text{Halt/Accept} \), then \( M_0 \) halts and accepts.

Example: \( M_0 \) looks for \( 110100 \) on Tape 1.

Now copy next state to Tape 3.

Write new symbol on Tape 2.

Move Tape 2 head to right one cell.

5. Move Tape 1, 3 heads to $$. Check for Accept. Go to 4.
THEOREM \( L_u \) IS NOT RECURSIVE

Thus it is undecidable given a TM \( M \) and input \( w \) to determine whether or not \( M \) accepts \( w \).

**Proof**

Assume to contrary that \( M_u \) decides \( L_u \).

Thus \( \langle M \rangle \# w \rightarrow M_u \rightarrow \text{YES} \iff w \in L(M) \rightarrow \text{NO} \iff w \notin L(M) \).

Consider \( M_u \) exists, always halts, because \( M_u \) does.
"REDUCTIONS"

If we can use an algorithm for deciding $L_2$ to obtain an alg. for deciding $L_1$, then we say: $L_1$ reduces to $L_2$

Write: $L_1 \leq L_2$

If $L_1 \leq L_2$ and $L_2$ is recursive, then $L_1$ is recursive.

(So $L_1$ is "no harder" than $L_2$, hence the "$\leq\"")

If $L_1 \leq L_2$ and $L_1$ is not recursive, then $L_2$ cannot be recursive.

"$\leq$" has a well defined technical meaning. We use reductions informally here. Later we carefully define "$\leq\" in context of NP-completeness.
ANOTHER NON-RECURSIVE LANGUAGE:

THE HALTING PROBLEM IS UNDECIDABLE

PRIM: GIVEN M, W, DECIDE IF M HALTS ON INPUT W

LANGUAGE: \[ L_{\text{HALT}} = \{ \langle M \rangle \# w : M \text{ halts on input } w \} \]

ALTERNATIVE \[ K = \{ i : M_i(i) \text{ halts} \} \]

K IS ALSO CALLED THE HALTING PROBLEM

THEOREM: \[ L_{\text{HALT}} \] IS NOT RECURSIVE

THUS \( \exists \) ALGORITHM, GIVEN A PROGRAM (TM) AND INPUT (W), THAT DECIDES WHETHER OR NOT THE PROGRAM WILL GET INTO AN \( \infty \) LOOP WHEN RUN ON INPUT W
Proof: Show \( L_u = L_{\text{HALT}} \)

And since \( \not\exists \text{alg for } L_u \), \( \not\exists \text{alg for } L_{\text{HALT}} \)

Assume \( L_{\text{HALT}} \) is recursive, witnessed by

\[
M_u \text{ (always halt)}
\]

\[
\begin{array}{c}
\langle M \rangle \neq w \\
\rightarrow \ M_{\text{HALT}} \\
\begin{array}{c}
\text{YES} \\
M(w) \downarrow
\end{array} \\
\begin{array}{c}
\text{NO} \\
M(w) \uparrow
\end{array}
\end{array}
\]

Simulate \( M(w) \)

\[
\begin{array}{c}
\text{YES} \\
\rightarrow \text{YES}
\end{array}
\]

\[
\begin{array}{c}
\text{NO} \\
\rightarrow \text{NO}
\end{array}
\]

\[
M_u \text{ (always halt)} \text{ determines } \forall \langle M \rangle, w \text{ whether or not } M \text{ accepts } w
\]

Always halts because it only simulates \( M(w) \) if the computation is guaranteed to halt (by assumption that \( M_{\text{HALT}} \) is correct)

Contradicts that \( L_u \) is not recursive
Another example

\[ L_{\text{NONEMPTY}} = \exists i : L(M_i) \neq \emptyset \]

is R.E. but not recursive.

\[ L_{\text{NONEMPTY}} \text{ is R.E.} \]

\[ L_{\text{NONEMPTY}} \text{ accepts } i \text{ iff } \exists w \text{ that } M_i \text{ accepts } i \text{ iff } L(M_i) \neq \emptyset \]
**Theorem**

L NONEMPTY IS NOT RECURSIVE

- Can't determine, given a program, whether or not the program accepts any input

**Proof Technique**

Show that $L_u \subseteq L$ NONEMPTY, thus if we had an algorithm to decide given $i$ whether or not $L(M_i) = \emptyset$, then we could use it to obtain an algorithm to decide given $\langle M \rangle \# w$ whether or not $w \in L(M)$.

Assume (to the contrary) that there exists

```
\[ i \rightarrow M \quad \text{NONEMPTY} \]
```

Yes if $L(M_i) \neq \emptyset$

No if $L(M_i) = \emptyset$
IDEA: CREATE A TRANSFORMATION (AN ALGORITHM) A

Such that

\[ \langle M \rangle \# w \rightarrow A \rightarrow \langle T_{M,w} \rangle \]

Given \( \langle M \rangle \# w \) A outputs a coding of A

\[ T_M, \langle T_{M,w} \rangle \] (that depends on \( M, w \))

Such that \( L(T_{M,w}) \neq \emptyset \iff M \text{ accepts } w \)

I.e., \( \langle T_{M,w} \rangle \in L_{\text{NonEmpty}} \iff \langle M \rangle \# w \in L_U \)

\[ M_U \text{ decides } L_U \]

\[ \langle M \rangle \# w \rightarrow A \rightarrow \langle T_{M,w} \rangle \rightarrow M_{\text{NonEmpty}} \]

\[ M_U \text{ accepts } \langle M \rangle \# w \iff M_{\text{NonEmpty}} \text{ accepts } \langle T_{M,w} \rangle \iff \langle M \rangle \# w \in L_U \]

\[ M_U \text{ always halts, since by assumption } M_{\text{NonEmpty}} \text{ does, and by construction } A \text{ does.} \]
DESCRIPTION OF TRANSFORMATION ALGORITHM A

INPUT TO A: \( \langle M \rangle \neq w \)

OUTPUT OF A: \( \langle T_{M,w} \rangle \)

WHERE \( T_{M,w} \) IS THE FOLLOWING TM:

\[ T_{M,w} \]

\[ X \rightarrow \]

RUN \( M \) ON \( w \)
IF \( M(w) \) ACCEPTS
THEN ACCEPT \( X \)

\( T_{M,w} \) IGNORES ITS INPUT AND ACCEPTS ONLY IF \( M \) ACCEPTS \( w \)
- IF \( M \) ACCEPTS \( w \), \( L(T_{M,w}) = \Sigma^* \)
- IF \( M \) DOESN'T ACCEPT \( w \), \( L(T_{M,w}) = \emptyset \)

THUS \( L(T_{M,w}) \neq \emptyset \iff M \) ACCEPTS \( w \) AS DESIRED

... But what does A look like? ??
RECALL \( M_u \) (UNIVERSAL TM) THAT RECEIVES \( \langle M \rangle \# w \) AND SIMULATES \( M \) ON \( w \)

A OUTPUTS DESCRIPTION OF \( T_{M_u} w \) THAT RUNS \( M_u \) ON INPUT \( \langle M \rangle \# w \)

\( T_{M_u} w \) HAS SUBROUTINE TO WRITE \( \langle M \rangle \# w \) ON TAPE (WRITING OVER INPUT)

\( T_{M_u} w \) CALLS \( M_u \)

GIVEN \( \langle M \rangle \# w \), THE TM CODE NEEDED TO WRITE \( \langle M \rangle \# w \) IS OBVIOUS

\( A \) CAN CONSTRUCT THIS PORTION OF CODE OF \( T_{M_u} w \) EASILY

THE TM CODE FOR \( M_u \) IS SOME NUMBER \( i \)

\( A \) HAS \( i \) STORED AS PART OF ITS DATA (IN A BIG PRINT \"START\")

Thus on input \( \langle M \rangle \# w \), \( A \) PRINTS:

\[ \text{III} \quad \text{TM transitions to print out } \langle M \# w \rangle \text{ on tape} \quad \text{II} \quad i \equiv \langle M \rangle \quad \text{III} \]
Thus problem of determining, given $M$, whether or not $L(M) \neq \emptyset$ is undecidable.

**Corollary 1**  
Can't decide if $L(M) = \emptyset$ either. Thus  
$\text{COMMAND} = \{ i : L(M_i) = \emptyset \}$  
is not recursive.

**Corollary 2**  
$\text{COMMAND}$ is not recursively enumerable either.

**Proof:**  
If it were, then both  
$\text{COMMAND}$, $\text{LANKDOM}$ would be r.e.  
and thus both would be recursive.
Show $L_{\text{EMPTY}}$ is not REC ENUM directly.

We show $L_d \leq L_{\text{EMPTY}}$, thus a TM accepting $L_{\text{EMPTY}}$ can be used to obtain a TM accepting $L_d$. (Neither TM need halt on rejecting)
But $L_d$ is known to be non REC ENUM.

Recall $L_d = \exists i : i \not\in L(M_{i'})$

We construct algorithm A (always halts) such that

\[ i \rightarrow A \rightarrow i' \]

where $i \not\in L(M_i) \iff L(M_{i'}) = \emptyset$

i.e., $i \in L_d \iff i' \in L_{\text{EMPTY}}$

Thus if $M_{\text{EMPTY}}$ accepts $L_{\text{EMPTY}}$, we have

\[ i \rightarrow A \rightarrow M_{\text{EMPTY}} \rightarrow \text{YES} \]

$M_d$ accepts $L_d$, a contradiction! $M_{\text{EMPTY}}$ can't exist!
WE NEED: $i \rightarrow A \rightarrow i'$

\[i \notin L(M_i) \iff L(M_{i'}) = \emptyset\]

- RATHER THAN EXPLICITLY CONSTRUCTING $A$
  WE JUST SHOW WHAT $M_{i'}$ IS IN TERMS OF $M_i$

- BECAUSE $M_{i'}$ WILL DEPEND ON $M_i$ "IN THE SAME WAY"

  REGARDLESS OF $i$, AND THE WAY IN WHICH IT DEPENDS
  IS RATHER STRAIGHTFORWARD, IT IS POSSIBLE TO CONSTRUCT
  $A$ THAT REALIZES THE TRANSFORMATION

- LEFT AS EXERCISE.
Proof that $i \notin L(M_i) \iff L(M_{i'}) = \emptyset$

- If $i \notin L(M_i)$ then no matter how many steps we run $M_i(i)$, it will never accept. So every $x$ is rejected and $L(M_{i'}) = \emptyset$.

- If $i \in L(M_i)$ then $\exists n$ $M_i(i)$ accepts in $n$ moves. Thus $\forall x | x > n$ $M_{i'}(x)$ will run $M_i(i)$ for enough steps to see that $M_i(i)$ accepts, so $M_{i'}$ accepts $x$ and thus $L(M_{i'}) \neq \emptyset$. 

"Hardcoded Data": value $i$

- Run $M_i(i)$ for $|x|$ steps
- IF $M_i(i)$ halts and accepts in $|x|$ steps
  THEN accept $x$
  ELSE reject $x$
ONE MORE EXAMPLE REDUCTION

Show that the problem of determining whether or not, given \( M \), \( |L(M)| = 17 \), is undecidable.

I.e., show

\[ L = \Sigma^* \text{ s.t. } |L(M_i)| = 17 \] is not recursive.

METHOD: \( L_U \leq L \)

(Show how an alg for \( L \) gives an alg for \( L_U \))

Devise transformation \( A \) such that

\[ \langle M \rangle \# W \rightarrow A \rightarrow \text{ i } \rightarrow \]

\[ |L(M_i)| = 17 \text{ if } M \text{ accepts } W \]

\[ |L(M_i)| \neq 17 \text{ if } M \text{ doesn't accept } W. \]

Then:

\[ \langle M \rangle \# W \rightarrow A \rightarrow M_L \]

YES

\[ \text{ iff } L \leq M_L \]

\[ \text{ iiff } |L(M_i)| = 17 \]

\[ \text{ iiff } M_i \text{ accepts } W \]

\[ \text{ iiff } \langle M \rangle \# W \leq L_u \]

\[ \text{ iiff } i \notin M_L \]

\[ \text{ iiff } M \text{ doesn't accept } W. \]
$\langle M, w \rangle \notin W \rightarrow A \rightarrow i$

\[ \text{M ACCEPTS } w \iff |L(M_i)| = 17 \]

Q: What should $M_i$ look like w.r.t. $M, w$?

A: If $M(w)$ accepts, $L(M_i)$ should be any language of 17 words.

If $M(w)$ doesn't accept, $L(M_i)$ should be any language of $\neq 17$ words.

If $M$ accepts $w$, $L(M_i) = \{0, 0, 0, 0, \ldots, 0\}^{17}$, cardinality 17.

If $M$ doesn't accept $w$, $L(M_i) = \emptyset$, cardinality 0.
Properties of Algorithms (TMs)

(|M|) given i, does L(Mi) contain i?

is L(Mi) empty?

" " nonempty?

given i, is L(Mi):

infinite?

finite?

recursive?

co-finite?

= Σ*

is |L(Mi)| even?

does L(Mi) contain "01"?

etc.

which properties are decidable?

Rice's Theorem: None!
A PROPERTY OF R.E. LANGUAGES

IS A SET $\exists i : L(M_i) \text{satisfies } P$ $\exists$

WHERE $P$ IS A PREDICATE $P : \exists L : L \text{ is R.E.}$ $\rightarrow \exists 0, 1, 3$

EXAMPLES $\exists i : L(M_i) \text{ is infinite}$ $\exists$

$\exists i : L(M_i) \text{ contains no primes}$ $\exists$

NONEXAMPLE $\exists i : M_i \text{ has 15 states}$ $\exists$

A PROPERTY IS TRIVIAL IF EITHER ALL OR NONE OF THE R.E. LANGUAGES SATISFY IT. I.E., IF THE SET IS EITHER IN OR $\emptyset$ $\emptyset$

EXAMPLES: $\exists i : L(M_i) \text{ is recursively enumerable} = \text{IN}$ $\exists$

$\exists i : L(M_i) \text{ is not R.E.} = \emptyset$ $\emptyset$

$\exists i : L(M_i) = \emptyset \text{ or } L(M_i) \neq \emptyset = \text{IN}$ $\exists$

CERTAINLY TRIVIAL PROPERTIES ARE DECIDABLE
RICE'S THEOREM

EVERY NONTRIVIAL PROPERTY OF THE R.E. LANGS IS UNDECIDABLE

Thus \( \{ i : L(M_i) \text{ satisfies } P \} \) is not recursive
(unless all or no langs satisfy P)

(SEVERELY LIMITS OUR ABILITY TO DETERMINE ALGORITHMICALLY A PROGRAM'S BEHAVIOR GIVEN ITS DESCRIPTION.

(WE CAN'T DETERMINE SQUAT)
NOTE

PROPERTIES ABOUT TMs (as opposed to
the languages they accept or functions they
compute) may or may not be decidable

EXAMPLES

\{i: M_i \text{ has 193 states}\} \quad \text{DECIDABLE}

\{i: M_i \text{ uses at most 32 tape cells on blank input}\}

\{i: M_i \text{ halts on blank input}\} \quad \text{UNDecidedable}

\{i: \text{on input "00110" M_i at some point writes the symbol "2" on its tape}\}

\bin^2\cdot 0 \cdot 32
**Proof of Rice's Theorem**

- Let $L = \exists i: L(M_i)$ satisfies $P$.
  
  Where not all, and at least one, RE. lang satisfies $P$.

- We show $L$ not recursive.

- W.l.o.g, assume $\emptyset = \text{code for empty language}$ does not satisfy property $P$.

  [If it did, we would consider the complementary language $\bar{L} = \exists i: L(M_i)$ does not satisfy $P$ and we would show $\bar{L}$ not recursive.]

- Let $i_0 \in L$ (at least one RE. lang satisfies $P$.
  
  So $i_0$ is its code.

  Thus $L(M_{i_0})$ satisfies $P$.

- This space for rent.
Assume to contrary that $\exists M_L$ deciding $L$.

We describe transformation $A$ such that

\[
\begin{array}{c}
\langle M \rangle \# w \\
\xrightarrow{A} \\
\langle M' \rangle
\end{array}
\]

where

- If $M$ accepts $w$, $M'$ accepts same lang. as $M_{\emptyset}$
- If $M$ doesn't accept $w$, $M'$ accepts $\emptyset$

I.E.

\[
\begin{align*}
\langle M \rangle \# w \in L_U & \iff \langle M' \rangle \in L \quad (\text{since } i_0 \in L) \\
\langle M \rangle \# w \in L_U & \iff \langle M' \rangle \in L \quad (\text{since } \emptyset \notin L)
\end{align*}
\]

If $A$ can be constructed, then:

\[
\begin{array}{c}
\langle M \rangle \# w \\
\xrightarrow{A} \\
\langle M' \rangle \\
\xrightarrow{M_L}
\end{array}
\]

$M_L$ decides $L_U$, a contradiction.
To complete proof, must describe a transforming \(<m \neq w>\) into \(<m'>\) such that:

- \(M(w)\) accepts \(\iff L(m') = L(M_{i_0})\)
- \(M(w)\) doesn't accept \(\iff L(m') = \emptyset\)

As before, we only describe \(M'\) in terms of \(M, w,\) leave as exercise the argument that \(E A\) producing \(<m'>\) given \(<m \neq w>\)

- Ignores input and runs \(M(w).\) If \(M(w)\) accepts, then runs \(M_{i_0}(x = \text{input}).\)
- If \(M(w)\) accepts, \(L(m') = L(M_{i_0})\)
- If \(M(w)\) doesn't, \(L(m') = \emptyset\)

QED
<table>
<thead>
<tr>
<th>LANGUAGE CLASS</th>
<th>MACHINE</th>
<th>GRAMMAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGULAR</td>
<td>DFA</td>
<td>&quot;REGULAR GRAMMAR&quot;</td>
</tr>
<tr>
<td></td>
<td>NFA</td>
<td>RIGHT-LINEAR GRAMMAR</td>
</tr>
<tr>
<td></td>
<td>2DFA</td>
<td>LEFT-LINEAR GRAMMAR</td>
</tr>
<tr>
<td></td>
<td>2NFA</td>
<td></td>
</tr>
<tr>
<td>CFL</td>
<td>PDA</td>
<td>CFG</td>
</tr>
<tr>
<td>CONTEXT-SENSITIVE LANGUAGES</td>
<td>LBA</td>
<td>CSG</td>
</tr>
<tr>
<td>RECURSIVE</td>
<td>HALTING TMs</td>
<td>*</td>
</tr>
<tr>
<td>RECURSIVELY ENUMERABLE</td>
<td>TMs</td>
<td>UNRESTRICTED &quot;PHRASE-STRUCTURE&quot; GRAMMARS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BEYOND &quot;COMPUTABLE&quot;</td>
</tr>
</tbody>
</table>
REGULAR GRAMMARS

RIGHT-LINEAR

ALL PROD. OF FORM
\[ \begin{cases} \lambda \to wB \\ A \to w \end{cases} \]
\[ w \in T^* \quad B \in V \]

LEFT-LINEAR

ALL PROD. OF FORM
\[ \begin{cases} A \to Bw \\ A \to w \end{cases} \]
\[ w \in T^* \quad B \in V \]

REGULAR GRAMMAR DEF:  \quad \text{RIGHT LINEAR}
\text{OR} \quad \text{LEFT LINEAR}

NOTE: NOT \[ \{ \begin{cases} A \to Bw \\ A \to wB \\ A \to w \end{cases} \} = \text{LINEAR GRAMMARS} \]}
THEOREM

\[ \text{REGULAR LANGS} = \text{LANGS GEN. BY RIGHT LIN. GRAMMARS} = \text{LANGS GEN BY LEFT LIN GRAMMARS} \]

(\( \neq \) \text{LANGS GEN BY LINEAR GRAMMARS})

PROOF (3 steps)

1. **RIGHT LINEAR GRAMMAR \( \rightarrow \) NFA

2. **NFA \( \rightarrow \) RIGHT LINEAR GRAMMAR

3. **\( \text{LANGS} (\text{LEFT LINEAR}) = \text{REVERSALS} (\text{LANGS} (\text{RIGHT LINEAR})) \)

= **\( \text{REVERSALS} (\text{REGULAR LANGS}) \)

= **\( \text{REGULAR LANGUAGES} \)
1. **Right Linear Grammar** \( \rightarrow \) **NFA**

   - **WAS NOT** a homework problem
   - WLOG Normal Form: \( A \rightarrow aB \), \( A \rightarrow a \)

   **Create NFA**:

   \[ \text{States} = \text{Nonterms} \cup \{ \text{Accept}\} \]

   \[ A \xrightarrow{a} B \iff A \rightarrow aB \]

   \[ A \xrightarrow{a, \text{Accept}} \iff A \rightarrow a \]

**Example**

```
S → aB | bA
A → bB | aC | a
B → aC
C → bS | b
```

- **Prove by induction** \( \forall A, w, B \)
  \[
  A \Rightarrow^* wB \iff \delta(A, w) = B
  \]

- **Will follow that**
  \[
  S \Rightarrow^* w \iff \delta(S, w) = \text{Accept}
  \]
NFA → RIGHT LINEAR GRAMMAR

SAME AS (1), BUT BACKWARDS

VARIABLES OF G = STATES OF M

\[ p \rightarrow aq \text{ A production if } s(p, a) = q \]

\[ p \rightarrow a \text{ also if } q \text{ a final state} \]

EXAMPLE

\[\text{DFA:}\]

\[\text{Grammar:}\]

\[A \rightarrow OB \mid 1D \mid 1\]

\[B \rightarrow OB \mid 1C\]

\[C \rightarrow OD \mid 0 \mid 1E\]

\[D \rightarrow 1B \mid 0E\]

\[E \rightarrow OD \mid 0 \mid 1E\]

CORRECTNESS: EASY INDUCTION SHOWING

\[s(p, w) = q \iff p \Rightarrow^* wq\]

AND IF q final state

\[s(p, w) = q \iff p \Rightarrow^* w\] also
UNRESTRICTED GRAMMARS

PHRASE-STRUCTURE GRAMMARS
TYPE O GRAMMARS
SEMI-THEU SYSTEMS

ALL PRODUCTIONS OF FORM

\[ \alpha \rightarrow \beta \]

\[ \alpha \in (VUT)^+ \]

\[ \beta \in (VUT)^* \]
EXAMPLE GRAMMAR FOR $a^n$: $n$ A POWER OF 2

1. $S \rightarrow \$ Ca \$

   Initialize with single "$a"$
   $\$ and $\$ are end markers

2. $Ca \rightarrow aaC$

   C moves to right, doubling each $a$

3. $C \# \rightarrow D \# \| E$

   When C reaches right end,
   either it turns into D
   (which means get ready for
   another doubling pass)

   or it turns into E
   (which will end the
   derivation after tidying up)

4. $aD \rightarrow Da$

   D moves left across $a$'s....

5. $\#D \rightarrow \$ C$

   ... until it reaches left marker,
   when it turns into C and
   then production (2) begins
   another doubling pass

6. $aE \rightarrow Ea$

   E moves left across $a$'s....

7. $\#E \rightarrow E$

   ... until it reaches $\$", then
   it and $\$ both vanish
   leaving only $a$'s.
THEOREM

\[ L \text{ is recursively enumerable} \iff L \text{ has an unrestricted grammar} \]

- Thus unrestricted grammars correspond exactly to languages accepted by Turing machines.
  (Further evidence of Church's thesis)

- \( \exists \) algorithm to determine, given unrestricted \( G \) and word \( w \), whether or not \( w \in L(G) \)
Proof ($\leq$)

- Let $G$ be an unrestricted grammar for $L$
- We describe a nondet. TM $M$ for $L$

Idea: $M$ simulates $G$
Given $M$ accepting $L$, show

$\exists$ grammar $G$ generating $L$

Main idea: Derivation simulates a "computation history" by manipulating IDs

1. $G$ has productions such that $\forall w, \forall k \in \mathbb{N}$
   \[ S \rightarrow^* w \# q_0 w b^k \]

2. $G$ simulates $M(w)$ with production rules that step-by-step manipulate
   \[ ID_0 = q_0 w b^k \] into $ID$'s of $M(w)$

   After $k^{th}$ production, the sentential form will be
   \[ w \# ID_k \]

3. $G$ has productions that allow any $ID_{accepting}$ to disappear, with preceding "#",
   leaving terminal string $w$ only if $M(w)$ accepts
0. $G$ has productions such that $\forall w \in \Sigma^* \forall k \in \mathbb{N}$

$$S \Rightarrow^* w \# q_w B_k$$

0. Exercise: Create grammar generating $\{ w \# w : w \in \Sigma^* \}$

0. Then trivially modify to generate $w \# q_w B_k$ for $k \in \mathbb{N}$

Meaning of $B_k$:

To right of $\#$, $G$ will simulate behavior of $M(w)$ by keeping track of successive IDs. Initially, we have init. ID of $M(w)$. However, sometime later $M(w)$ might use lots of extra tape. $B_k$ is a string of blanks as long as $M(w)$ will ever need.
Productions of $G \cong$ transitions of $M$ allowing $1D_4 \Rightarrow 1D_2$

Right move:

If $S(q_1, 0) = (q_2, 1, R)$ in machine $M$

Then $G$ has production $q_1, 0 \Rightarrow 1q_2$

This allows

$I0i = \cdots 9, 0 \cdots \Rightarrow \cdots 1q_2 \cdots = 1D_{i+1}$

Left move more involved:

If $S(q_1, 0) = (q_2, 1, L)$

Then $G$ has productions

\[ \begin{align*}
0q_0 & \Rightarrow q_2 01 \\
1q_0 & \Rightarrow q_2 11 \\
Bq_0 & \Rightarrow q_2 B1
\end{align*} \]

One prod. for each symbol of $\Gamma$ that might precede scanned cell

Thus $1D_0 = q_0 w \overset{+}{\Rightarrow} 1D_{\text{accept}}$ \(\iff S \Rightarrow^* \# w_0 w B^k \Rightarrow^* \# w \# \text{ accept} \) (for $k$ large enough)
WE HAVE

\[ S \Rightarrow^* w \neq q_0 w B^k \Rightarrow^* w \# \text{10Accepting} \]

IFF \( M \) accepts \( w \)

NOW ADD PRODUCTIONS TO ELIMINATE \#10Accepting.

CONSIDER: \( \text{10Accepting} = \sum x_1 x_2 x_3 \ldots x_n q_{\text{Accept}} x_{n+1} \ldots x_m \)

\[ \text{\# has productions} \]

\[ q_{\text{Accept}} x \Rightarrow q_{\text{Accept}} \]

\[ x_{q_{\text{Accept}}} \Rightarrow q_{\text{Accept}} \]

\[ \text{\#} q_{\text{Accept}} \Rightarrow \varepsilon \]

THESE IF \( q_{\text{Accept}} \) APPEARS ON RIGHT, CAN USE THESE PRODUCTIONS TO ERASE ENTIRE RIGHT SIDE, LEAVING ONLY \( w \):

\[ w \neq x_1 x_2 \ldots x_n q_{\text{Accept}} x_{n+1} \ldots x_m \Rightarrow^* w \neq q_{\text{Accept}} \Rightarrow w \]

THIS CAN ONLY HAPPEN IF \( M(w) \) ACCEPTS. IF \( M(w) \) REJECTS OR LOOPS, NO STRING OF \#10Accepting CAN BE GENERATED!!
SLIGHT ERROR WITH PREV. CONSTRUCTION —

accept might "erase" "#" before

it erases all symbols to the
right of the #, leaving wx
(all terminals) where x = garbage terminals

Solution: instead of writing w to right
of # initially, write W, a string
of nonterminals, one for each terminal
d w. Then entire "computation"
is done using the NONTERMINALS to
the right of #.

- to obtain a string of terminals,
  these must be erased.
CONTEXT-SENSITIVE LANGUAGES

"MOST LANGS ARE CSLs"

CONTEXT SENSITIVE GRAMMAR

ALL PRODUCTIONS OF FORM

\[ \alpha \rightarrow \beta \]

\[ \alpha \in (VUT)^+ \]

\[ \beta \in (VUT)^+ \]

\[ |\beta| \geq |\alpha| \]

"NON ERASING"

SENTENTIAL FORMS CANNOT SHRINK

NORMAL FORM: ALL PROD. OF FORM

\[ \alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2 \]

\[ A e V \]

\[ \beta \in (VUT)^+ \]

"A \rightarrow \beta" PRODUCTION IS APPLICABLE

ONLY IN THE CONTEXT OF \( \alpha_1 \), \( \alpha_2 \)
LINEAR BOUNDED AUTOMATA (LBA)

- Restricted space TM.
- Only allow TM to use \( C \cdot |w| \) space when run on \( w \). Later theorem (beyond scope of this course) shows this is \( \leq \) using \( |w| \) space.

A LBA is a Nondet TM with one input/work tape, and special characters $, \$.

\[
\begin{array}{c|c|c}
$ & w & \$ \\
\end{array}
\]

Computation occurs in place.

The LBA cannot move past $ or \$ symbols.
CSGs \equiv \text{LBA's}

CSGs CANNOT GENERATE \epsilon, \text{ BUT IGNORING THIS,}

CSGs GENERATE SAME LANGUAGES AS LBA's CAN ACCEPT.

**THEOREM**

1. \forall CSG G \exists LBA M
   such that \( L(M) = L(G) \)

2. \forall LBA M \exists CSG G
   such that \( L(G) = L(M) - \{\epsilon\} \)

**PROOF IDEA** (SAME AS UNRESTRICTED GRAMMARS \& TMs)

1. CREATE M THAT SIMULATES A DERIVATION IN PLACE

2. CREATE G THAT SIMULATES A COMPUTATION HISTORY WITHOUT ERASING
0. **GIVEN** CSG $G$ **CONSTRUCT** LBA $M$. 

\[
\begin{align*}
&M \text{ SAVES INPUT } w \text{ ON UPPER TRACK} \\
&M \text{ TRIES (NONDETERMINISTICALLY) TO GENERATE } w \text{ ON LOWER TRACK}
\end{align*}
\]

\[\text{SAME AS PROOF THAT A TM CAN SIMULATE AN UNRESTRICTED GRAMMAR, BUT WE USE EXTRA TRACK INSTEAD OF EXTRA TAPE.}\]

\[\begin{array}{c}
\$ \quad \text{W} \quad \text{EX} \quad \$ \\
\text{SENTENTIAL FORM IN DERIVATION OF } w
\end{array}\]

**BUT!** **HOW DO WE GUARANTEE THAT DERIVATION SIMULATION STAYS WITHIN \$,$ \$ MARKERS?**

- **SINCE** $G$ **IS A CSG (NONRECURSING), NO SENTENTIAL FORM IN DERIVATION OF $w$ CAN HAVE LENGTH $> |w|$.
- **THUS** $M$ **NEED ONLY CONSIDER SENTENTIAL FORMS THAT FIT ON TAPE.** IF $M$ **NONDET GENERATES A LONGER $\alpha$, THEN THIS CORRESPONDS TO A REJECTING COMPUTATION PATH.**
2. Given LBA M construct CSG G

- Recall simulation of TM M by unrestricted grammar G

1. For every possible word w, number k
   \[ G \text{ generates } w \# q_0 w B^k \]

2. G uses productions to simulate computation of M by manipulating ID's to right of #.
   - Top "track" saves \( w = q_1 \ldots q_n \).
   - Bottom "track" is just an ID of TM computation.
   - The state (\( q_0 \) initially) is stored with symbol being scanned.
   - G productions manipulate these symbols just as in

3. If simulation of TM by unrestricted grammar accepts is generated, additional productions allow erasing of everything to right of #, leaving terminal string w
   If \[ q_1 \ldots q_i \ldots q_n \] is generated, G turns this into terminals \( q_{a_0} \ldots q_{a_n} = w \).
CSLs $\subsetneq$ RECURSIVE LANGUAGES

(\text{\textbf{\textit{E}}}) \quad \text{IF } L \text{ IS A CSL, \exists CSG } G \text{ FOR } L

\text{G IS NONERASING, SO IF } w \in L(G),
\exists \text{ DERIVATION INVOLVING ONLY SENTENTIAL FORMS OF LENGTH } \leq |w|$

\text{TO DETERMINE GIVEN } G, w \text{ WHETHER OR NOT } w \in L(G)$

\begin{itemize}
  \item MAKE GRAPH WITH VERTEX SET \( \{v \in (V \cup T)^* : |v| \leq |w| \} \)

  \text{AND DIRECTED EDGE } \delta_i \rightarrow \delta_2 \text{ IFF }
  \delta_i \Rightarrow_G \delta_2 \text{ VIA A SINGLE PRODUCTION}

  \item \( w \in L(G) \) IFF \exists PATH FROM \( S \) TO \( w \)
\end{itemize}

\text{THUS } \{<G, w : w \in L(G)> \} \text{ IS RECURSIVE (ABOVE NOT)}

\text{ACCEPTED BY SOME TM } M\text{. }

\text{FURTHER } L(G) = \{w : w \in L(G)\} \text{ IS ACCEPTED BY TM } M_G,
\text{ WHICH IS } M \text{ BUT WITH } G\text{'S DESCRIPTION HARDCODED, INSTEAD OF RECEIVING } G \text{ AS INPUT.}
Recursive - CSL \neq \emptyset

- Proved as corollary of much more general theorem.

Theorem

Let \( L_1, L_2, L_3, \ldots \) be a recursively enumerable sequence of TM codes, all of which halt within

Then \( \exists \) recursive \( L \) such that \( \forall j \ L \neq L(M_j) \)

Theorem says there is no way to enumerate all \((\forall)\) only the TMs that halt. (Note \( M = 0, 1, 2, 3, \ldots \) is an enumeration of all TMs. Unfortunately we cannot weed out nonhalting ones without losing some halting ones.)

Proof... by diagonalization...

(WLOG assume the sequence \( L_1, L_2, \ldots \) is infinite, since if it were finite clearly it cannot contain a TM for every rec. lang.)
Let $G$ generate $i_1, i_2, i_3, \ldots$ codes for TMs that always halt.

Now consider:

- $M$ halts on every input (why?)
- Thus $L(M)$ is recursive
- $L(M) \neq L(M_{i,j})$ for any $j$ (why?)

$\exists i : L(M_i)$ is recursive $\exists i$ is not r.e. Q.E.D.
COROLLARY 1

3. RECURSIVE LANGUAGES THAT ARE NOT CONTEXT-SENSITIVE

PROOF

BY SHOWING THAT WE CAN ENUMERATE TMs THAT HALT ∀ INPUTS, AND THAT THE LIST CONTAINS A TM FOR EVERY CSL

(A) CAN ENUMERATE ALL CONTEXT-SENSITIVE GRAMMARS OVER TERMINAL ALPHABET \( \{0, 1\}^* \)

[SKETCH] WLOG ONLY NEED, SAY, 3 NONTERM SYMBOLS, SINCE MORE NONTERMS COULD BE SIMULATED BY SEQUENCES OF NONTERMS (NOTE THIS DOESN'T WORK FOR CFG)

THEN A CSG CAN BE ENCODED AS A FINITE LIST OF SYMBOLS OVER FINITE ALPHABET

\[ \{S, A, B, 0, 1, \rightarrow, \} \]

CAN ENUMERATE THESE CSGs LExicographically

LET \( M \) BE ENUMERATOR OF CSGs:

\[ M \rightarrow \langle G_0 \rangle, \langle G_1 \rangle, \langle G_2 \rangle, \ldots \]
For each (encoding of A) CSL \( G \), we can (algorithmically) construct a TM \( M_G \) that halts on all inputs and such that \( L(G) = L(M_G) \).

**Sketch**

\( M_G \) was described earlier. On input \( w \), it constructs a graph of sentential forms of length \( \leq |w| \), with edges \( \delta_1 \rightarrow \delta_2 \) if in \( G \) we have \( \delta_1 \rightarrow \delta_2 \). Then it checks if \( \exists \) directed path \( S \rightarrow \ldots \rightarrow w \).

Let \( A \) realize this construction of \( M_G \) from \( G \).

Thus:

\[ \langle G \rangle \xrightarrow{A} \langle M_G \rangle \]
A + B \text{ gives:}

\[ M \rightarrow \langle G_0 \rangle, \langle G_1 \rangle, \ldots \rightarrow A \rightarrow \langle M_{G_0} \rangle, \langle M_{G_1} \rangle, \langle M_{G_2} \rangle, \ldots \]

- A \text{ machine that generates TM codes}
- All of which \text{ halt on inputs,}
- Every CSL has some machine on list.

\text{By Theorem, } \exists \text{ recursive } L \neq L(M_{G_i}) \text{ for any } i

\text{Thus } \exists \text{ recursive } L, L \text{ not a CSL.}

\text{QED}

(of Corollary 1)
COROLLARY 2

F GRAMMARS "CAPTURING" RECURSIVE LANGS ONLY

DON'T BE SILLY... SURE THERE ARE... HOW ABOUT THOSE UNRESTRICTED GRAMMARS THAT CORRESPOND TO RECURSIVE LANGS (I.E. THROW OUT UNRESTRICTED GRAMMARS CORRESPONDING TO R.E. - REC. LANGS)

YES, ... BUT WE CAN'T IDENTIFY WHICH THESE ARE (JUST LIKE WE CAN'T IDENTIFY WHICH TM'S HALT)

IF A CLASS OF GRAMMARS \( G \) SATISFIES

(A) CAN TELL IF \( G \) IS A VALID \( G \) GRAMMAR

[[THUS CAN ENUMERATE ALL \( G \) GRAMMARS, ASSUMING THEY HAVE FINITE REPRESENTATIONS]]

(B) GIVEN A \( G \)-GRAMMAR \( G \) AND WORD \( w \)

CAN TELL IF \( w \in L(G) \) OR IF \( w \notin L(G) \)

[[CERTAINLY WOULD WANT THIS FOR GRAMMARS CAPTURING [RECURSIVE] LANGUAGES]]

THEN PROOF IDENTICAL TO THAT OF COROLLARY 1 SHOWS \( \exists \) RECURSIVE \( L \) THAT HAS NO \( G \)-GRAMMAR
THE BIG PICTURE (IGNORING "E")

HOMSKY HIERARCHY

REGULAR ⊆ CFL ⊆ CSL ⊆ REC. ENUM.

MORE CLASSES... ALL "NONCOMPUTABLE"