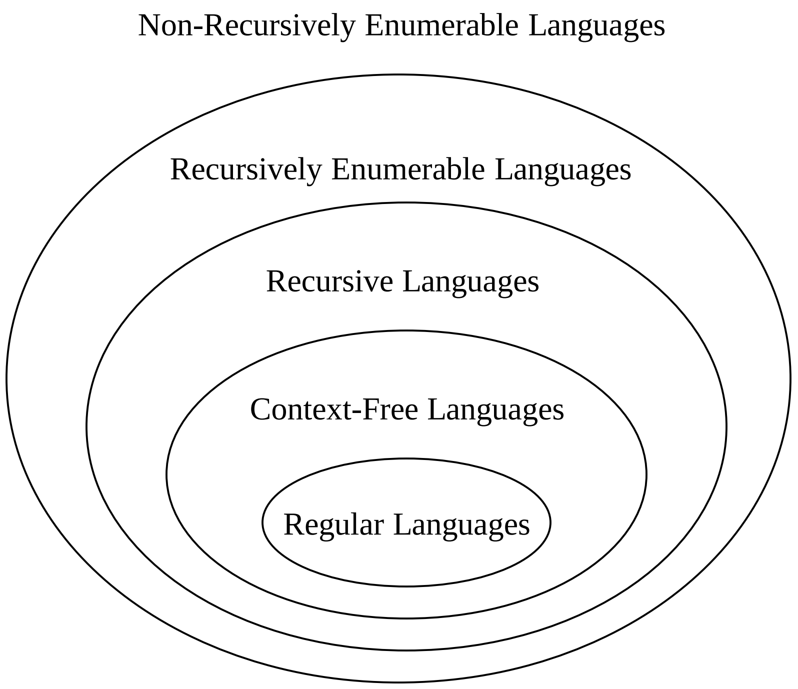
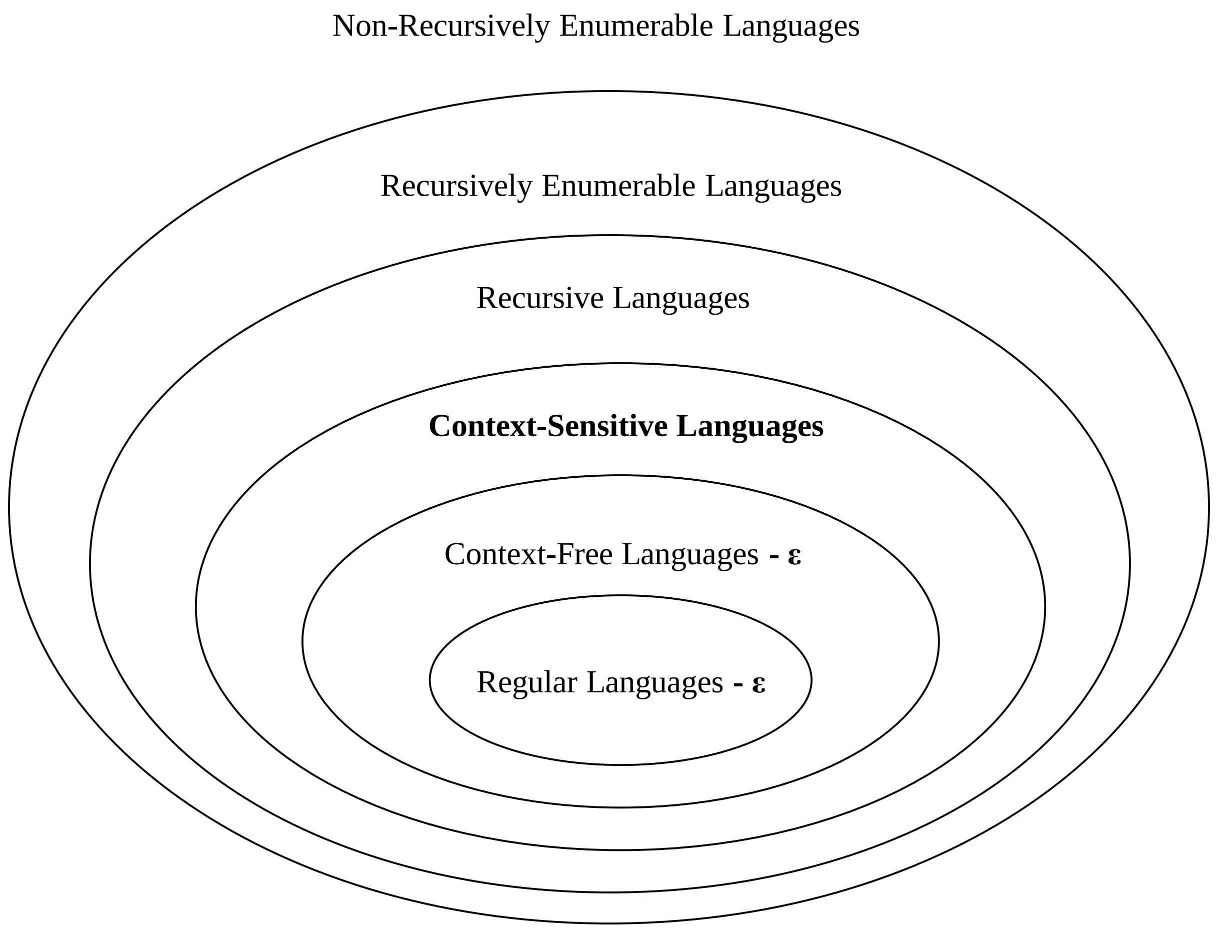
Turing Machines (TM)

- Generalize the class of CFLs:


Non-Recursively Enumerable Languages
Recursively Enumerable Languages
Recursive Languages
Context-Free Languages
Regular Languages

- **Another Part of the Hierarchy:**


Non-Recursively Enumerable Languages
Recursively Enumerable Languages
Recursive Languages
Context-Sensitive Languages
Context-Free Languages **- ε**
Regular Languages **- ε**

- Recursively enumerable languages are also known as *type 0* languages.
- Context-sensitive languages are also known as *type 1* languages.
- Context-free languages are also known as *type 2* languages.
- Regular languages are also known as *type 3* languages.

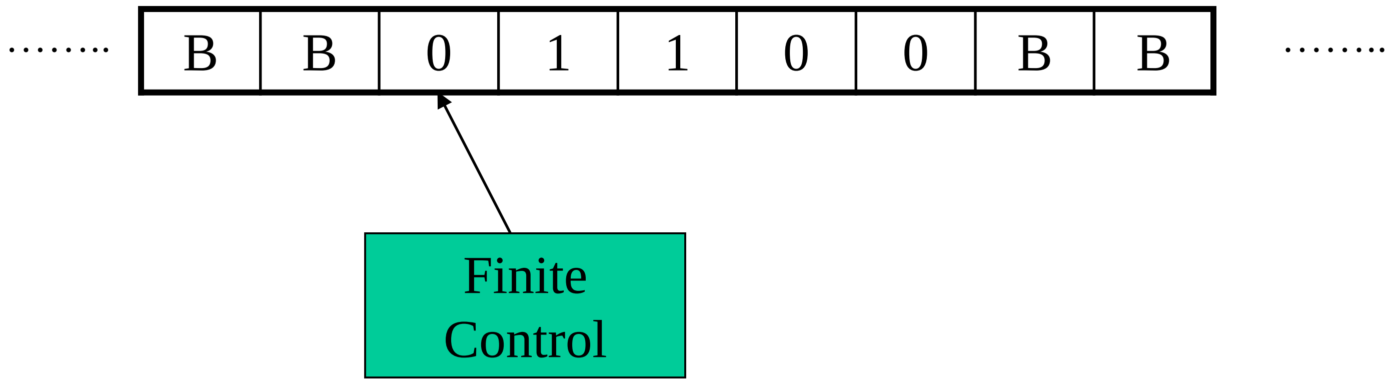
- TMs model the computing capability of a general purpose computer, which informally can be described as:
  - Effective procedure
    - Finitely describable
    - Well defined, discrete, “mechanical” steps
    - Always terminates
  - Computable function
    - A function computable by an effective procedure

- TMs formalize the above notion.

- **Church-Turing Thesis:** There is an effective procedure for solving a problem if and only if there is a TM that halts for all inputs and solves the problem.
  - There are many other computing models, but all are equivalent to or subsumed by TMs. *There is no more powerful machine* (Technically cannot be proved).

- DFAs and PDAs do not model all effective procedures or computable functions, but only a subset.

Deterministic Turing Machine (DTM)



- Two-way, infinite tape, broken into cells, each containing one symbol.
- Two-way, read/write tape head.
- An input string is placed on the tape, padded to the left and right infinitely with blanks, read/write head is positioned at the left end of input string.
- Finite control, i.e., a program, containing the position of the read head, current symbol being scanned, and the current state.
- In one move, depending on the current state and the current symbol being scanned, the TM 1) changes state, 2) prints a symbol over the cell being scanned, and 3) moves its' tape head one cell left or right.
- Many modifications possible, but Church-Turing declares equivalence of all.

Formal Definition of a DTM

- A DTM is a seven-tuple:

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

Q	A <u>finite</u> set of states
Σ	A <u>finite</u> input alphabet, which is a subset of $\Gamma - \{B\}$
Γ	A <u>finite</u> tape alphabet, which is a strict <u>superset</u> of Σ
B	A distinguished blank symbol, which is in Γ
q_0	The initial/starting state, q_0 is in Q
F	A set of final/accepting states, which is a subset of Q
δ	A next-move function, which is a *mapping* (i.e., may be undefined) from $Q \times \Gamma \to Q \times \Gamma \times \{L,R\}$

Intuitively, $\delta(q,s)$ specifies the next state, symbol to be written, and the direction of tape head movement by M after reading symbol s while in state q.

- **Example #1:** {w | w is in {0,1}* and w ends with a 0}

  0
  00
  10
  10110
  Not ε

  $Q = \{q_0, q_1, q_2\}$
  $\Gamma = \{0, 1, B\}$
  $\Sigma = \{0, 1\}$
  $F = \{q_2\}$
  δ:

	0	1	B
$\to q_0$	$(q_0, 0, R)$	$(q_0, 1, R)$	(q_1, B, L)
q_1	$(q_2, 0, R)$	-	-
q_2^*	-	-	-

  – q_0 is the start state and the "scan right" state, until hits B
  – q_1 is the verify 0 state
  – q_2 is the final state

- **Example #2:** $\{0^n1^n \mid n \geq 1\}$

	0	1	X	Y	B
$\to q_0$	(q_1, X, R)	-	-	(q_3, Y, R) ***0's finished***	-
q_1	$(q_1, 0, R)$ ***ignore1***	(q_2, Y, L)	-	(q_1, Y, R) ***ignore2***	- (more 0's)
q_2	$(q_2, 0, L)$ ***ignore2***	-	(q_0, X, R)	(q_2, Y, L) ***ignore1***	-
q_3	-	- (more 1's)	-	(q_3, Y, R) ***ignore***	(q_4, B, R)
q_4*	-	-	-	-	-

- **Sample Computation:** (on 0011), *presume state q looks rightward*

$q_0$0011BB.. $\vdash$ X$q_1$011
$\vdash$ X0$q_1$11
$\vdash$ X$q_2$0Y1
$\vdash$ q_2X0Y1
$\vdash$ X$q_0$0Y1
$\vdash$ XXq_1Y1
$\vdash$ XXY$q_1$1
$\vdash$ XXq_2YY
$\vdash$ Xq_2XYY
$\vdash$ XXq_0YY
$\vdash$ XXYq_3Y B…
$\vdash$ XXYYq_3 BB…
$\vdash$ XXYYBq_4

Making a TM for $\{0^n1^n \mid n >= 1\}$

Try n=2 or 3 first.

- q0 is on 0, replaces with the character to X, changes state to q1, moves right
- q1 sees next 0, ignores (both 0's and X's) and keeps moving right
- q1 hits a 1, replaces it with Y, state to q2, moves left
- q2 sees a Y or 0, ignores, continues left
- when q2 sees X, moves right, returns to q0 for looping step 1 through 5
- when finished, q0 sees Y (no more 0's), changes to pre-final state q3
- q3 scans over all Y's to ensure there is no extra 1 at the end (to crash on seeing any 0 or 1)
- when q3 sees B, all 0's matched 1's, done, changes to final state q4
- blank line for final state q4

Try n=1 next.

Make sure unbalanced 0's and 1's, or mixture of 0-1's,
"crashes" in a state not q4, as it should be

$$
\begin{aligned}
q_0 0011BB.. &\vdash Xq_1011 \\
&\vdash X0q_111 \\
&\vdash Xq_20Y1 \\
&\vdash q_2X0Y1 \\
&\vdash Xq_00Y1 \\
&\vdash XXq_1Y1 \\
&\vdash XXYq_11 \\
&\vdash XXq_2YY \\
&\vdash Xq_2XYY \\
&\vdash XXq_0YY \\
&\vdash XXYq_3Y\ B\ldots \\
&\vdash XXYYq_3\ BB\ldots \\
&\vdash XXYYBq_4
\end{aligned}
$$

- **Same Example #2:** $\{0^n1^n \mid n \geq 1\}$

	0	1	X	Y	B
q_0	(q_1, X, R)	-	-	(q_3, Y, R)	-
q_1	$(q_1, 0, R)$	(q_2, Y, L)	-	(q_1, Y, R)	-
q_2	$(q_2, 0, L)$	-	(q_0, X, R)	(q_2, Y, L)	-
q_3	-	-	-	(q_3, Y, R)	(q_4, B, R)
q_4	-	-	-	-	-

Logic: cross 0's with X's, scan right to look for corresponding 1, on finding it cross it with Y, and scan left to find next leftmost 0, keep iterating until no more 0's, then scan right looking for B.

- The TM matches up 0's and 1's
- q_1 is the "scan right" state, looking for 1
- q_2 is the "scan left" state, looking for X
- q_3 is "scan right", looking for B
- q_4 is the final state

Can you extend the machine to include n=0?
How does the input-tape look like for string epsilon?

- **Other Examples:**

000111	00
11	001
011	

- **Roger Ballard's TM for Example #2, without any extra Tape Symbol:** $\{0^n1^n \mid n \geq 0\}$

	0	1	B
q_0	(q_1, B, R)		(q_4, B, R)
q_1	$(q_1, 0, R)$	$(q_1, 1, R)$	(q_2, B, L)
q_2	-	(q_3, B, L)	-
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4^*	-	-	-

Logic: Keep deleting 0 and corresponding 1 from extreme ends, until none left.

- q_0 deletes a leftmost 0 and let q_1 scan through end of string, q_0 accepts on epsilon
- q_1 scans over the string and makes q_2 expecting 1 on the left
- q_2 deletes 1 and let q_3 "scan left" looking for the start of current string
- q_3 lets q_0 start the next iteration
- q_4 is the final state

Any bug?

Try on:

000111	00
11	001
011	

- **And his example of a correct TM for the language that goes on infinite loop outside language:** $\{0^n1^n \mid n \geq 0\}$

	0	1	B
q_0	(q_1, B, R)	$(q_3, 1, L)$	(q_4, B, R)
q_1	$(q_1, 0, R)$	$(q_1, 1, R)$	(q_2, B, L)
q_2	-	(q_3, B, L)	-
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4^*	-	-	-

Logic: This machine still works correctly for all strings in the language, but
start a string with 1 (not in the language),
and it loops on B1 for ever.

- **Exercises:** Construct a DTM for each of the following.
  - {w | w is in {0,1}* and w ends in 00}
  - {w | w is in {0,1}* and w contains at least two 0's}
  - {w | w is in {0,1}* and w contains at least one 0 and one 1}
  - Just about anything else (simple) you can think of

Formal Definitions for DTMs

- Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a TM.

- **Definition:** An *instantaneous description* (ID) is a triple $\alpha_1 q \alpha_2$, where:

  - q, the current state, is in Q
  - $\alpha_1\alpha_2$, is in Γ^*, and is the current tape contents up to the rightmost non-blank symbol, or the symbol to the left of the tape head, whichever is rightmost
  - The tape head is currently scanning the first symbol of α_2
  - At the start of a computation $\alpha_1 = \boldsymbol{\varepsilon}$
  - If $\alpha_2 = \boldsymbol{\varepsilon}$ then a blank is being scanned

- **Example:** (for TM #1)

$q_0 0011$	$Xq_1 011$	$X0q_1 11$	$Xq_2 0Y1$	$q_2 X0Y1$
$Xq_0 0Y1$	$XXq_1 Y1$	$XXYq_1 1$	$XXq_2 YY$	$Xq_2 XYY$
$XXq_0 YY$	$XXYq_3 Y$	$XXYYq_3$	$XXYYBq_4$	

- Suppose the following is the current ID of a DTM

$$x_1x_2\ldots x_{i-1}qx_ix_{i+1}\ldots x_n$$

Case 1) $\delta(q, x_i) = (p, y, L)$

(a) if $i = 1$ then $qx_1x_2\ldots x_{i-1}x_ix_{i+1}\ldots x_n \vdash pByx_2\ldots x_{i-1}x_ix_{i+1}\ldots x_n$

(b) else $x_1x_2\ldots x_{i-1}qx_ix_{i+1}\ldots x_n \vdash x_1x_2\ldots x_{i-2}px_{i-1}yx_{i+1}\ldots x_n$

  – If any suffix of $x_{i-1}yx_{i+1}\ldots x_n$ is blank then it is deleted.

Case 2) $\delta(q, x_i) = (p, y, R)$

$x_1x_2\ldots x_{i-1}qx_ix_{i+1}\ldots x_n \vdash x_1x_2\ldots x_{i-1}ypx_{i+1}\ldots x_n$

  – If $i>n$ then the ID increases in length by 1 symbol

$x_1x_2\ldots x_nq \vdash x_1x_2\ldots x_nyp$

- **Definition:** Let M = (Q, Σ, Γ, δ, q_0, B, F) be a TM, and let w be a string in Σ*. Then w is *accepted* by M iff

  $$q_0w \vdash^* \alpha_1 p \alpha_2$$

  where p is in F and α_1 and α_2 are in Γ*

- **Definition:** Let M = (Q, Σ, Γ, δ, q_0, B, F) be a TM. The *language accepted by M*, denoted L(M), is the set

  {w | w is in Σ* and w is accepted by M}

- **Notes:**
  – In contrast to FA and PDAs, if a TM simply *passes through* a final state then the string is accepted.
  – Given the above definition, no final state of a TM need to have any transitions. *Henceforth, this is our assumption*.
  – **If x is NOT in L(M) then M may enter an infinite loop, or halt in a non-final state**.
  – Some TMs halt on ALL inputs, while others may not. In either case the language defined by TM is still well defined.

- **Definition:** Let *L* be a language. Then *L* is *recursively enumerable* if <u>there exists</u> a TM *M* such that L = L(M).

  – If *L* is r.e. then L = L(M) for some TM *M*, and
    - If *x* is in *L* then *M* halts in a final (accepting) state.
    - If *x* is not in *L* then *M* may halt in a non-final (non-accepting) state or no transition is available, or loop forever.

- **Definition:** Let *L* be a language. Then *L* is *recursive* if there exists a TM *M* such that L = L(M) and M halts on all inputs.

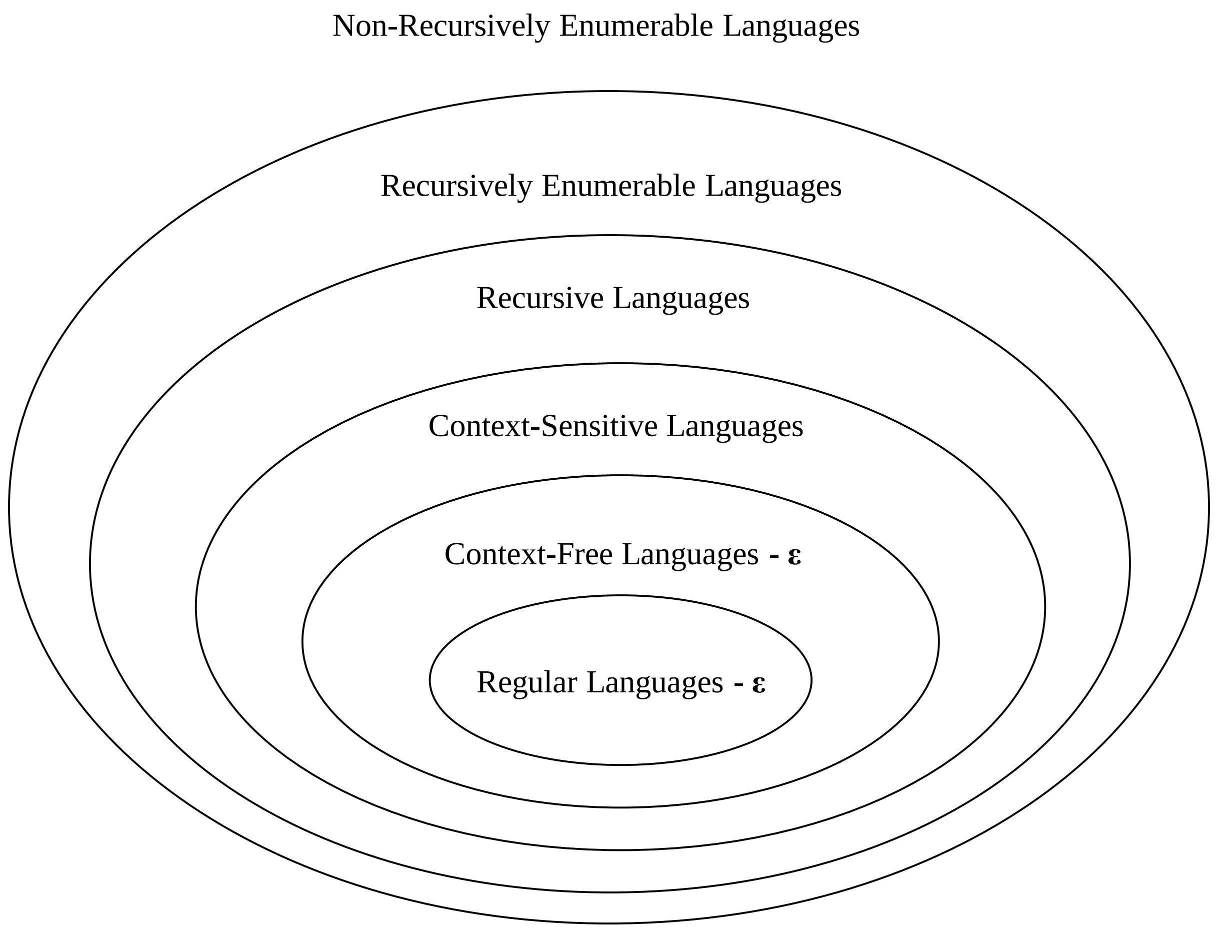
  – If *L* is recursive then L = L(M) for some TM *M*, and
    - If *x* is in *L* then *M* halts in a final (accepting) state.
    - If *x* is not in *L* then *M* halts in a non-final (non-accepting) state or no transition is available (does <u>not</u> go to infinite loop).

Notes:

– The set of all recursive languages is a subset of the set of all recursively enumerable languages

– Terminology is easy to confuse: A *TM* is not recursive or recursively enumerable, rather a *language* is recursive or recursively enumerable.

- **Recall the Hierarchy:**


Non-Recursively Enumerable Languages
Recursively Enumerable Languages
Recursive Languages
Context-Sensitive Languages
Context-Free Languages - **ε**
Regular Languages - **ε**

- **Observation:** Let L be an r.e. language. Then there is an infinite list M_0, M_1, … of TMs such that L = L(M_i).

- **Question:** Let L be a **recursive** language, and M_0, M_1, … a list of all TMs such that L = L(M_i), and choose any i>=0. Does M_i always halt?

- **Answer:** Maybe, maybe not, but *at least one in the list does*.

- **Question:** Let L be a **recursive enumerable** language, and M_0, M_1, … a list of all TMs such that L = L(M_i), and choose any i>=0. Does M_i always halt?

- **Answer:** Maybe, maybe not. Depending on L, none might halt or some may halt.

  – If L is also recursive then L is recursively enumerable, *recursive is subset of r.e.*

- **Question:** Let L be a r.e. language that is not recursive (L is in r.e. – r), and M_0, M_1, … a list of all TMs such that L = L(M_i), and choose any i>=0. Does M_i always halt?

- **Answer:** No! If it did, then L would not be in r.e. – r, it would be recursive.

L is Recursively enumerable:

TM exist: M_0, M_1, *…*

They accept string in L, and do not accept any string outside L

L is Recursive:

at least one TM halts on L and on $\sum$*-L, others may or may not*

L is Recursively enumerable but not Recursive:

TM exist: M_0, M_1, *…*

but <u>none</u> halts on <u>all</u> x in $\sum$*-L*

M_0 *goes on infinite loop on a string p in* $\sum$*-L, while* M_1 *on q in* $\sum$*-L*

However, each correct TM accepts each string in L, and none in $\sum$*-L*

L is not R.E*:*

no TM exists

- **Let *M* be a TM.**
  - Question: Is L(M) r.e.?
  - Answer: Yes! By definition it is!

  - Question: Is L(M) recursive?
  - Answer: Don’t know, we don’t have enough information.

  - Question: Is L(M) in r.e – r?
  - Answer: Don’t know, we don’t have enough information.

- **Let *M* be a TM that halts on all inputs:**
  - Question: Is L(M) recursively enumerable?
  - Answer: Yes! By definition it is!

  - Question: Is L(M) recursive?
  - Answer: Yes! By definition it is!

  - Question: Is L(M) in r.e – r?
  - Answer: No! It can’t be. Since *M* always halts, L(M) is recursive.

- **Let *M* be a TM.**
  - As noted previously, L(M) is recursively enumerable, but may or may not be recursive.
  - Question: Suppose, we know L(M) is recursive. Does that mean *M* always halts?
  - Answer: Not necessarily. However, some TM *M'* must exist such that L(M') = L(M) and *M'* always halts.
  - Question: Suppose that L(M) is in r.e. – r. Does *M* always halt?
  - Answer: No! If it did then L(M) would be recursive and therefore not in r.e. – r.

- **Let *M* be a TM, and suppose that *M* loops forever on some string *x*.**
  - Question: Is L(M) recursively enumerable?
  - Answer: Yes! By definition it is. But, obviously *x* is not in L(M).

  - Question: Is L(M) recursive?
  - Answer: Don't know. Although *M* doesn't always halt, some other TM *M'* may exist such that L(M') = L(M) and *M'* always halts.

  - Question: Is L(M) in r.e. – r?
  - Answer: Don't know.

  May be another *M'* will halt on *x*, and on all strings! May be no TM for this L(M) does halt on all strings! We just do not know!

Modifications of the Basic TM Model

- **Other (Extended) TM Models:**
  - One-way infinite tapes
  - Multiple tapes and tape heads
  - Non-Deterministic TMs
  - Multi-Dimensional TMs (n-dimensional tape)
  - Multi-Heads
  - Multiple tracks

All of these extensions are equivalent to the basic DTM model

Closure Properties for Recursive and Recursively Enumerable Languages

- **TMs model General Purpose (GP) Computers:**
  - If a TM can do it, so can a GP computer
  - If a GP computer can do it, then so can a TM

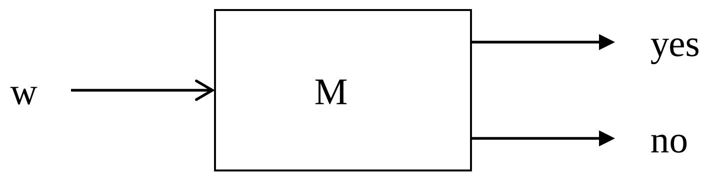
If you want to know if a TM can do X, then some equivalent question are:

- *Can a general purpose computer do X?*
- *Can a C/C++/Java/etc. program be written to do X?*

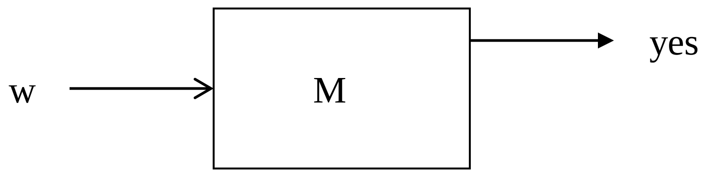
For example, is a language L recursive?

- *Can a C/C++/Java/etc. program be written that always halts and accepts L?*

- **TM Block Diagrams:**
  - If *L* is a recursive language, then a TM *M* that accepts *L* and always halts can be pictorially represented by a “chip” or “box” that has one input and two outputs.



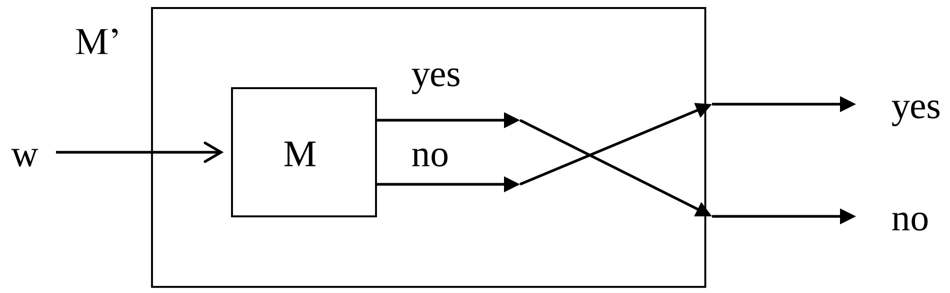
  - If *L* is a recursively enumerable language, then a TM *M* that accepts *L* can be pictorially represented by a “box” that has one output.



  - Conceivably, *M* could be provided with an output for “no,” but this output cannot be counted on. Consequently, we simply ignore it.

- **Theorem 1:** The recursive languages are closed with respect to complementation, i.e., if *L* is a recursive language, then so is $\overline{L} = \Sigma^* - L$

- **Proof:** Let *M* be a TM such that L = L(M) and *M* always halts. Construct TM *M'* as follows:



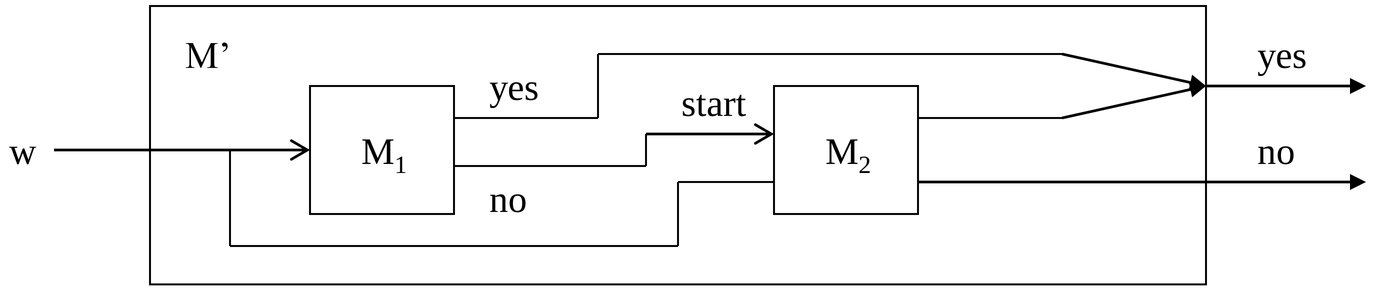
- **Note That:**
  - *M'* accepts iff *M* does not
  - *M'* always halts since *M* always halts

From this it follows that the complement of *L* is recursive. •

- **Question:** How is the construction achieved? Do we simply complement the final states in the TM? No! A string in *L* could end up in the complement of *L*.
  - Suppose q_5 is an accepting state in *M*, but q_0 is not.
  - If we simply complemented the final and non-final states, then q_0 would be an accepting state in *M'* but q_5 would not.
  - Since q_0 is an accepting state, by definition all strings are accepted by *M'*

- **Theorem 2:** The recursive languages are closed with respect to union, i.e., if L_1 and L_2 are recursive languages, then so is $L_3 = L_1 \cup L_2$

- **Proof:** Let M_1 and M_2 be TMs such that $L_1 = L(M_1)$ and $L_2 = L(M_2)$ and M_1 and M_2 always halts. Construct TM M' as follows:

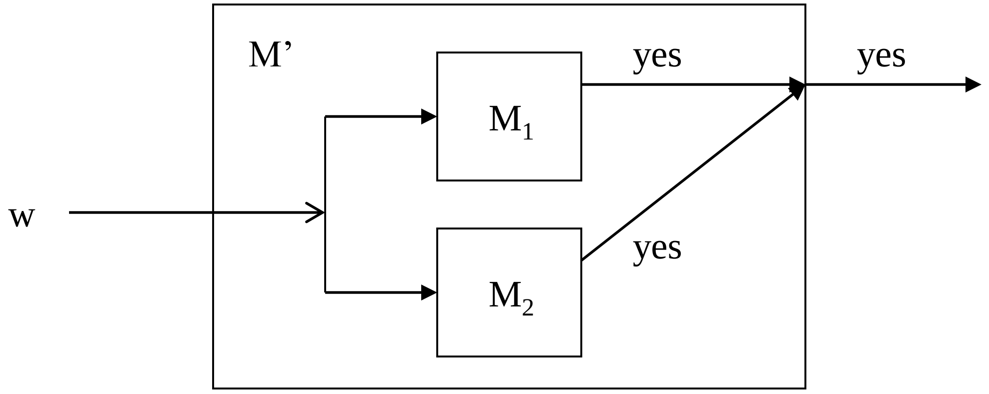


- **Note That:**
  - $L(M') = L(M_1) \cup L(M_2)$
    - $L(M')$ is a subset of $L(M_1) \cup L(M_2)$
    - $L(M_1) \cup L(M_2)$ is a subset of $L(M')$
  - M' always halts since M_1 and M_2 always halt

It follows from this that $L_3 = L_1 \cup L_2$ is recursive. •

- **Theorem 3:** The *recursive enumerable languages* are closed with respect to union, i.e., if L_1 and L_2 are recursively enumerable languages, then so is $L_3 = L_1 \cup L_2$

- **Proof:** Let M_1 and M_2 be TMs such that $L_1 = L(M_1)$ and $L_2 = L(M_2)$. Construct *M'* as follows:

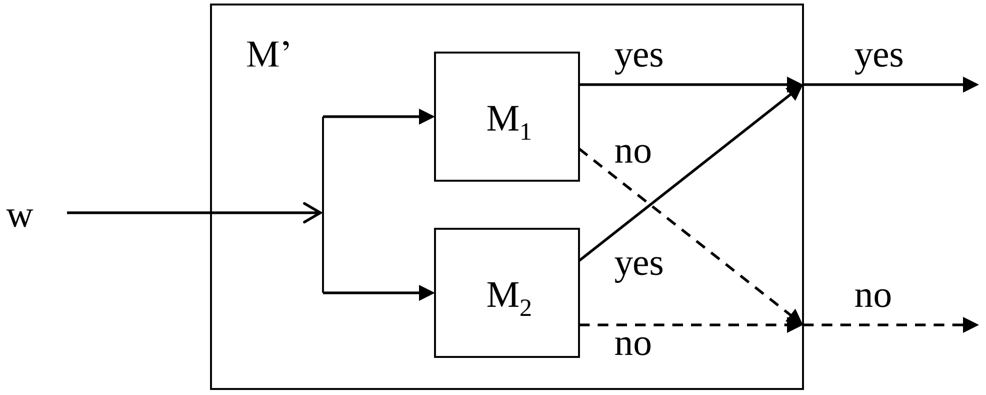


- **Note That:**
  - $L(M') = L(M_1) \cup L(M_2)$
    - $L(M')$ is a subset of $L(M_1) \cup L(M_2)$
    - $L(M_1) \cup L(M_2)$ is a subset of $L(M')$
  - *M'* halts and accepts iff M_1 or M_2 halts and accepts

It follows from this that $L_3 = L_1 \cup L_2$ is recursively enumerable. •

- ***Question:*** *How do you run two TMs in parallel?*

- Suppose, M_1 and M_2 had outputs for "no" in the previous construction, and these were transferred to the "no" output for *M'*

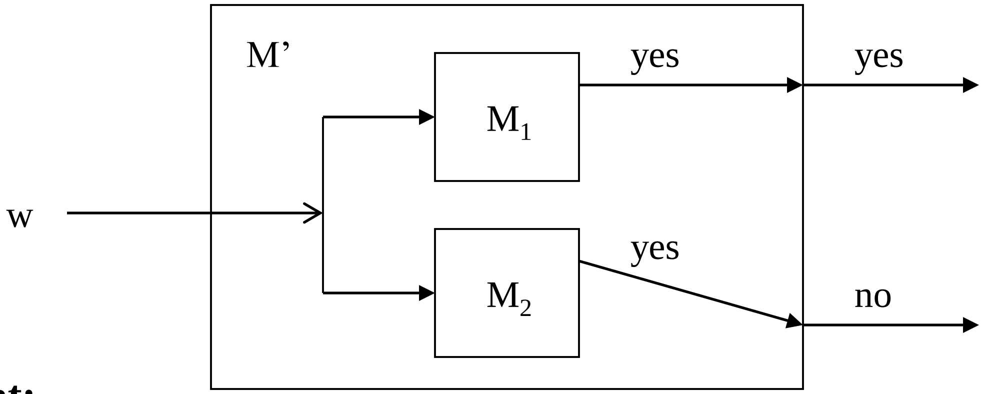


- **Question:** What would happen if *w* is in L(M_1) but not in L(M_2)?

- **Answer:** You could get two outputs – one "yes" and one "no."
  - At least M_1 will halt and answer accept, M_2 may or may not halt.
  - As before, for the sake of convenience the "no" output will be ignored.

- **Theorem 4:** If L and $\overline{L}$ are both recursively enumerable then L (and therefore $\overline{L}$) is recursive.

- **Proof:** Let M_1 and M_2 be TMs such that L = L(M_1) and $\overline{L}$ = L(M_2). Construct *M'* as follows:



- **Note That:**
  - L(M') = L
    - L(M') is a subset of *L*
    - *L* is a subset of L(M')
  - M' is TM for L
  - *M'* always halts since <u>either</u> M_1 <u>or</u> M_2 halts for any given string
  - M' shows that L is recursive

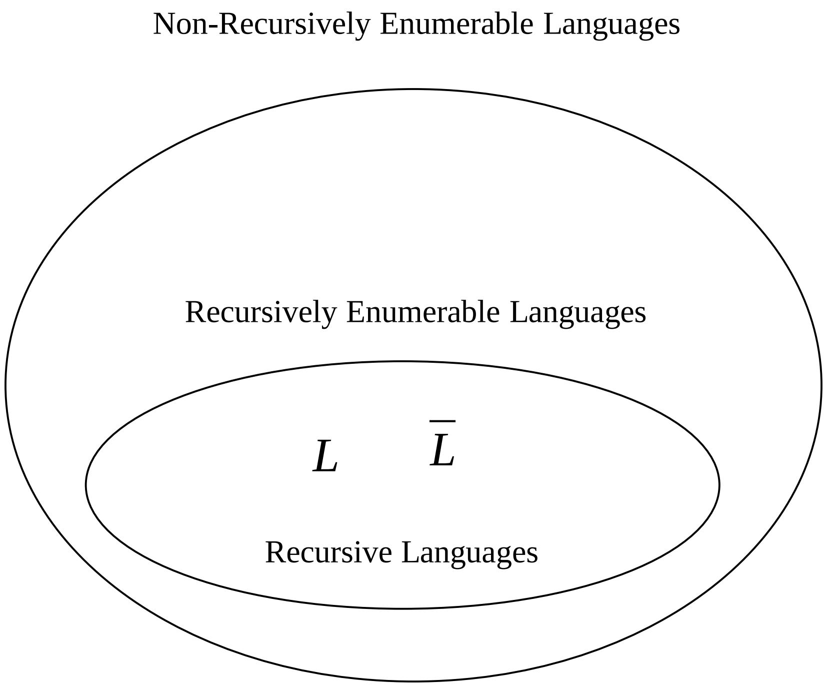
It follows from this that *L* (and therefore its' complement) is recursive.

So, $\overline{L}$ is also recursive (we proved it before). •

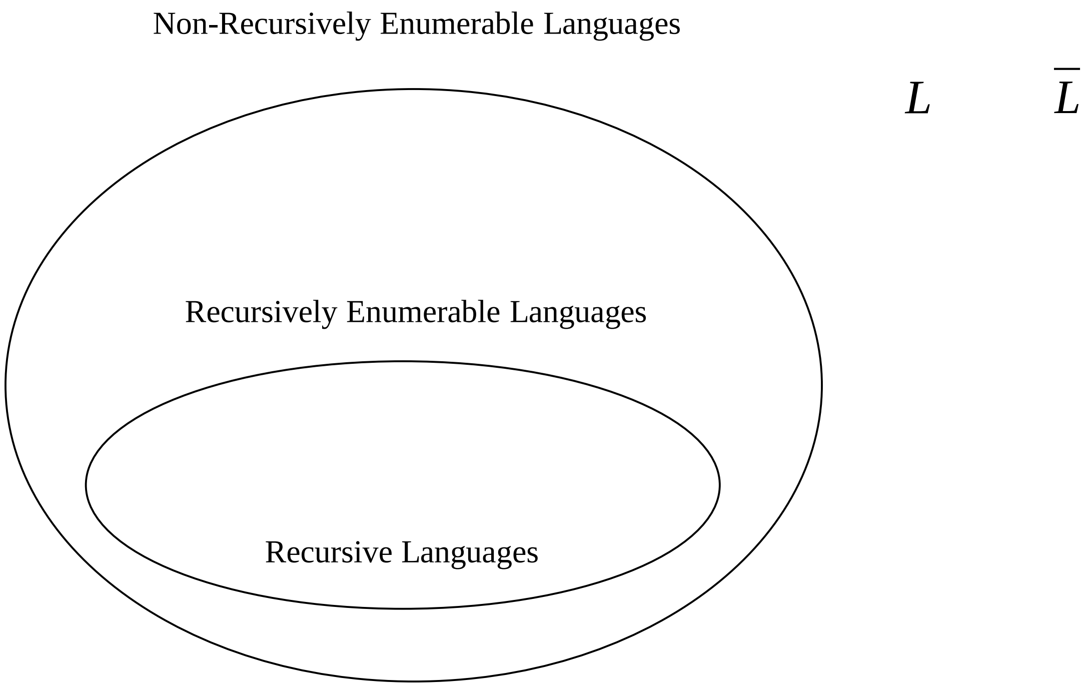
- **Corollary of Thm 4:** Let L be a subset of Σ^*. Then one of the following must be true:
  - Both L and $\overline{L}$ are recursive.
  - One of L and $\overline{L}$ is recursively enumerable but not recursive, and the other is not recursively enumerable, or
  - Neither L nor $\overline{L}$ is recursively enumerable

  - *In other words, it is impossible to have both L and* $\overline{L}$ *r.e. but not recursive*

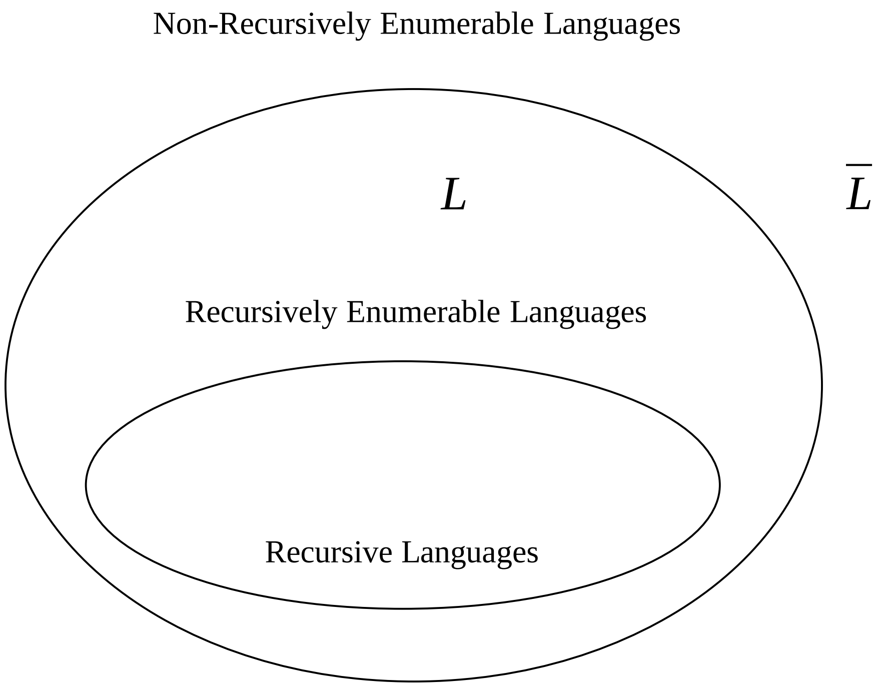
- **In terms of the hierarchy:** (possibility #1)


Non-Recursively Enumerable Languages
Recursively Enumerable Languages
L
$\overline{L}$
Recursive Languages

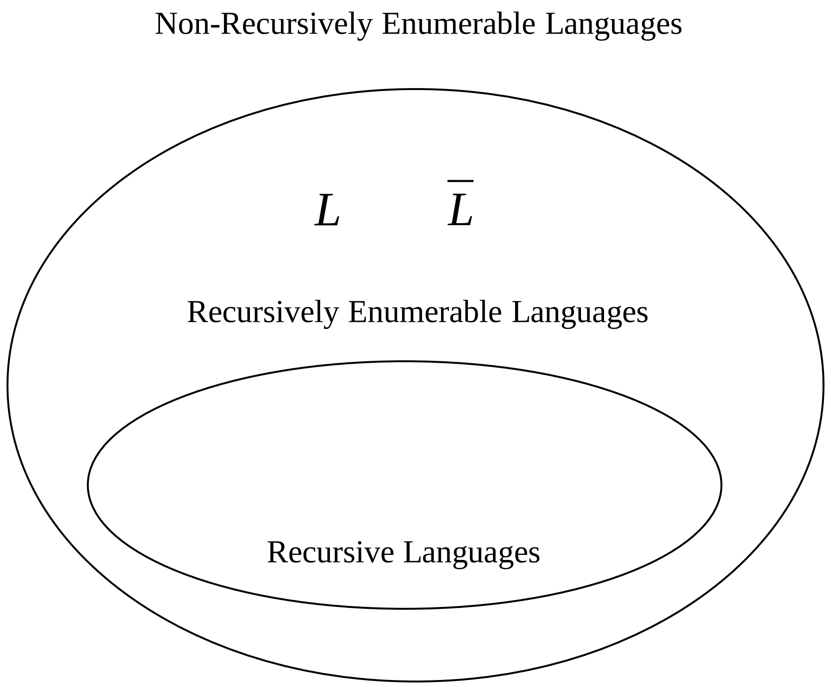
- **In terms of the hierarchy:** (possibility #2)


Non-Recursively Enumerable Languages
Recursively Enumerable Languages
Recursive Languages
L
$\overline{L}$

- **In terms of the hierarchy:** (possibility #3)

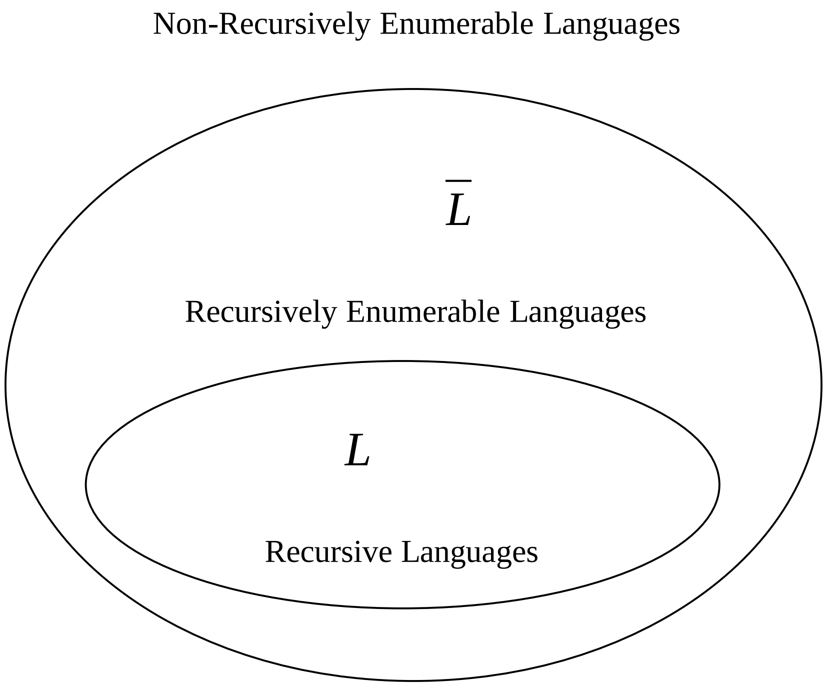

Non-Recursively Enumerable Languages
L
$\overline{L}$
Recursively Enumerable Languages
Recursive Languages

- **In terms of the hierarchy:** (Impossibility #1)

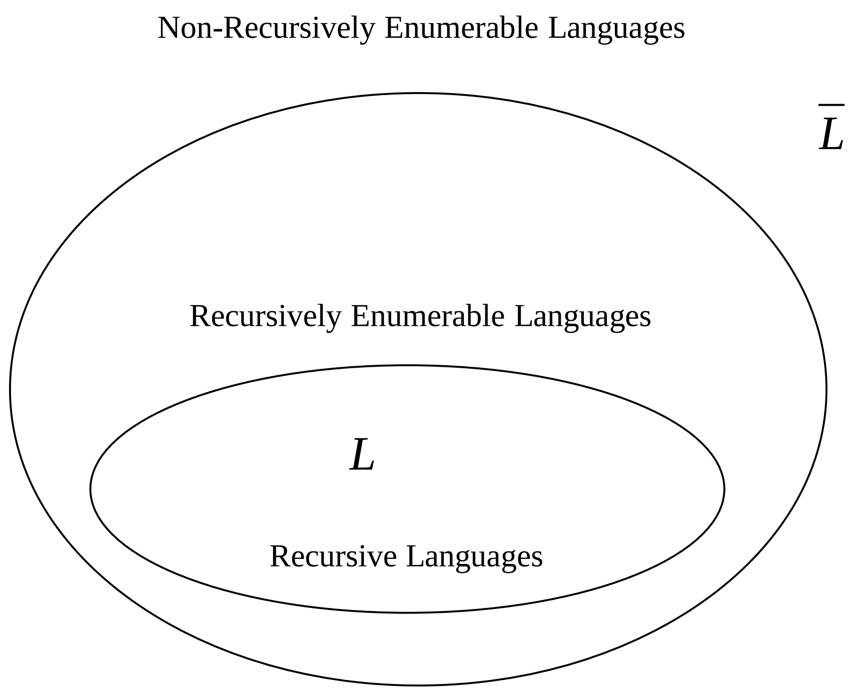


Non-Recursively Enumerable Languages
L
$\overline{L}$
Recursively Enumerable Languages
Recursive Languages

- **In terms of the hierarchy:** (Impossibility #2)


Non-Recursively Enumerable Languages
$\overline{L}$
Recursively Enumerable Languages
L
Recursive Languages

- **In terms of the hierarchy:** (Impossibility #3)


Non-Recursively Enumerable Languages
$\overline{L}$
Recursively Enumerable Languages
L
Recursive Languages

- **Note:** This gives/identifies three approaches to show that a language is not recursive.
  - Show that the language's <u>complement</u> is not recursive, in one of the two ways:
    - Show that the language's <u>complement</u> is recursively enumerable but not recursive
    - Show that the language's <u>complement</u> is not even recursively enumerable

The Halting Problem - Background

- **Definition:** A decision problem is a problem having a yes/no answer (that one presumably wants to solve with a computer). Typically, there is a list of parameters on which the problem is based.
  - Given a list of numbers, is that list sorted?
  - Given a number x, is x even?
  - Given a C program, does that C program contain any syntax errors?
  - Given a TM (or C program), does that TM contain an infinite loop?

  From a practical perspective, many decision problems do not seem all that interesting. However, from a theoretical perspective they are for the following two reasons:
  - Decision problems are more convenient/easier to work with when proving complexity results.
  - Non-decision *counter-parts* can always be created & are typically at least as difficult to solve.

- **Notes:**
  - The following terms and phrases are analogous:

Algorithm	-	A halting TM program
Decision Problem	-	A language *(will show shortly)*
(un)Decidable	**-**	**(non)Recursive**

Statement of the Halting Problem

- **Practical Form:** (P1)
  Input: Program P and input I.
  Question: Does P terminate on input I?

- **Theoretical Form:** (P2)
  Input: Turing machine M with input alphabet Σ and string w in Σ*.
  Question: Does M halt on w?

- **A Related Problem We Will Consider First:** (P3)
  Input: Turing machine M with input alphabet Σ and one final state, and string w in Σ*.
  Question: Is w in L(M)?

- **Analogy:**
  Input: DFA M with input alphabet Σ and string w in Σ*.
  Question: Is w in L(M)?
  Is this problem (*regular language*) decidable? Yes! DFA always accepts or rejects.

- **Over-All Approach:**
  - We will show that a language L_d is not recursively enumerable
  - From this it will follow that $\overline{L_d}$ is not recursive
  - Using this we will show that a language L_u is not recursive
  - From this it will follow that the halting problem is undecidable.
- **As We Will See:**
  - P3 will correspond to the language L_u
  - Proving P3 (un)decidable is equivalent to proving L_u (non)recursive

Converting the Problem to a Language

- Let M = (Q, Σ, Γ, δ, q_1, B, {q_n}) be a TM, where

  Q = {q_1, q_2, … , q_n}, order the states from 1 through n
  Σ = {x_1, x_2} = {0, 1}
  Γ = {x_1, x_2, x_3} = {0, 1, B}

- Encode each transition:

  $\delta(q_i, x_j) = (q_k, x_l, d_m)$ where q_i and q_k are in ordered Q
  x_j and x_l are in Σ,
  and d_m is in {L, R} = {d_1, d_2}

  as:

  $0^i10^j10^k10^l10^m$ where the number of 0's indicate the corresponding id, and single 1 acts as a barrier

- The TM *M* can then be encoded as:

  $111code_111code_211code_311 \ldots 11code_r111$

  where each $code_i$ is one transitions' encoding, and 11's are barriers between transitions from the table row-major. Let this encoding of *M* be denoted by <M>.

- Less Formally:

  - Every state, tape symbol, and movement symbol is encoded as a sequence of 0's:

q_1,	0
q_2,	00
q_3	000
:	
0	0
1	00
B	000
L	0
R	00

  - Note that 1's are not used to represent the above, since 1 is used as a special separator symbol.

  - Example:

$$\delta(q_2, 1) = (q_3, 0, R)$$

Is encoded as:

00100100010100

	0	1	B
q_1	(q_1, 0, R)	(q_1, 1, R)	(q_2, B, L)
q_2	(q_3, 0, R)	-	-
q_3	-	-	-

What is the L(M)?

Coding for the above table:

*111*0101010100**11**0100101001001**1**01000100100010**11**00101000101001*111*

Are the followings correct encoding of a TM?

01100001110001

111111

- **Definition:**

  L_t = {x | x is in {0, 1}* and x encodes a TM}

  – Question: Is L_t recursive?
  – Answer: Yes. [Check only for format, i.e. the order and number of 0's and 1's, syntax checking]

  – Question: Is L_t decidable:
  – Answer: Yes (same question).

The Universal Language

- Define the language L_u as follows:

  L_u = {x | x is in {0, 1}* and x = <M,w> where *M* is a TM encoding and *w* is in L(M)}

- Let *x* be in {0, 1}*. Then either:

  1. *x* doesn't have a TM prefix, in which case *x* is **not** in L_u

  2. *x* has a TM prefix, i.e., x = <M,w> and either:
     a) *w* is not in L(M), in which case *x* is **not** in L_u

     b) *w* is in L(M), in which case *x* is in L_u

- **Recall:**

	0	1	B
q_1	$(q_1, 0, R)$	$(q_1, 1, R)$	(q_2, B, L)
q_2	$(q_3, 0, R)$	-	-
q_3	-	-	-

- **Which of the following are in L_u?**

1110101010100110100101001001101000100100010110010100010100**111**

1110101010100110100101001001101000100100010110010100010100**111**01110

1110101010100110100101001001101000100100010110010100010100**111**00110111

01100001110001

111111

- **Compare P3 and L_u:**

  (P3):
  Input: Turing machine *M* with input alphabet Σ and one final state, and string *w* in Σ*.
  Question: Is *w* in L(M)?

  L_u = {x | x is in {0, 1}* and x = <M,w> where *M* is a TM encoding and *w* is in L(M)}

- Universal TM *(UTM)* is the machine for L_u
  - presuming it is r.e.! *Can you write a program to accept strings in Lu?*
- **Notes:**
  - L_u is P3 expressed as a language
  - *Asking if L_u is recursive is the same as asking if P3 is decidable.*
    - *Can you write a Halting program for accept/reject of strings in Sigma* ?*
  - We will show that L_u is *not recursive*, and from this it will follow that P3 is *un-decidable*.
  - From this we can further show that the *Halting problem is un-decidable*.

  => A general concept: *a decision problem ≡ a formal language*

- Define another language L_d as follows:

  L_d = {x | *x* is in {0, 1}* and (a) either *x* is **not** a TM,
  (b) or *x* is a TM, call it M, and *x* is **not** in L(M)}   (1)

  – *Note, there is only one string x*
  – *And, the question really is the complement of "does a TM accept its own encoding?" (Ld-bar's complement)*

- Let x be in {0, 1}*. Then either:

  1. *x* is **not** a TM, in which case *x* is *in* L_d
  2. *x* is a TM, call it M, and either:
     a) *x* is **not** in L(M), in which case *x* is *in* L_d
     b) *x* is in L(M), in which case *x* is **not** in L_d

- **Recall:**

	0	1	B
q_1	$(q_1, 0, R)$	$(q_1, 1, R)$	(q_2, B, L)
q_2	$(q_3, 0, R)$	-	-
q_3	-	-	-

- **Which of the following are in L_d?**

11101010101001101001010010011010001000100010110010100010100111

01100001110001

Change above machine to accept strings ending with 1: the encoding will not be in L_d

- **Lemma:** L_d *is not recursively enumerable.* *[No TM for L_d!!!]*

- **Proof:** (by contradiction)

  Suppose that L_d is recursively enumerable. In other words, there exists a TM *M* such that:

  $L_d = L(M)$ (2)

  Now suppose that *w* is a string encoding of *M*. (3)

  Case 1) ***w* is in $\mathbf{L_d}$** (4)

  By definition of L_d given in (1), either *w* does not encode a TM, or *w* does encode a TM, call it *M*, and *w* is not in L(M). But we know that *w* encodes a TM (3: that's where it came from). Therefore:

  w is not in L(M) (5)

  But then (2) and (5) imply that ***w* is not in $\mathbf{L_d}$** contradicting (4).

  Case 2) ***w* is not in $\mathbf{L_d}$** (6)

  By definition of L_d given in (1), *w* encodes a TM, call it *M*, and:

  w is in L(M) (7)

  But then (2) and (7) imply that ***w* is in $\mathbf{L_d}$** contradicting (6).

  Since both case 1) and case 2) lead to a contradiction, no TM *M* can exist such that $L_d = L(M)$. Therefore L_d is not recursively enumerable. •

- **Note:**

  $\overline{L_d}$ = {x | x is in {0, 1}*, x encodes a TM, call it M, and x is in L(M)}

- **Corollary:** $\overline{L_d}$ is not recursive.

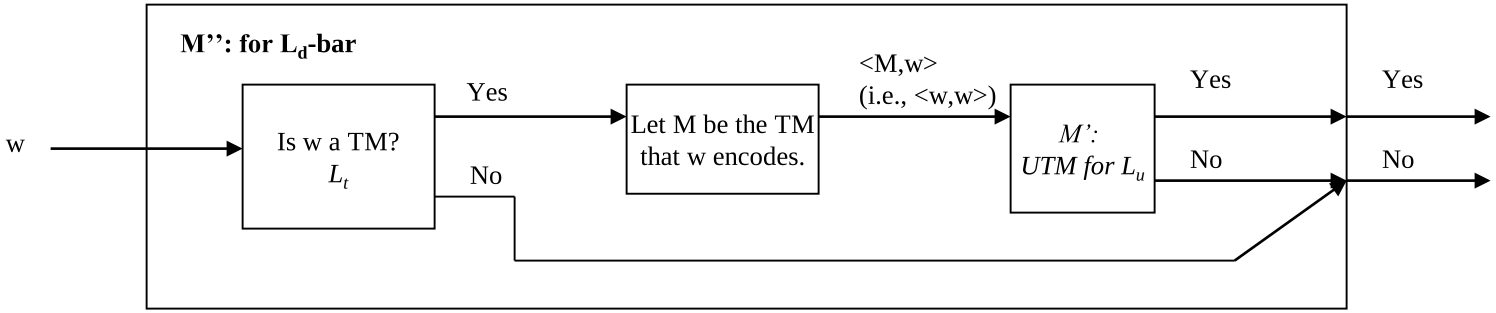
- **Proof:** If $\overline{L_d}$ were recursive, then L_d would be recursive, and therefore recursively enumerable, a contradiction. •

- **Theorem:** L_u *is not recursive*.

- **Proof:** (by contradiction)
  Suppose that L_u is recursive. Recall that:

  L_u = {x | x is in {0, 1}* and x = <M,w> where *M* is a TM encoding and *w* is in L(M)}

  Suppose that L_u = L(M') where *M'* is a TM that always halts. Construct an algorithm (i.e., a TM that always halts) for $\overline{L_d}$ as follows:



Suppose that *M'* always halts and L_u = L(M'). It follows that:

– M'' always halts
– L(M'') = $\overline{L_d}$

$\overline{L_d}$ would therefore be recursive, a contradiction. •

L_u is recursively enumerable (you may ignore this slide, for now)

```
Input the string
    Decode the TM prefix, if it doesn't have one then the string is not in Lu
    Otherwise, run/simulate the encoded TM on the suffix
    If it terminates and accepts then the original string is in Lu.
```

If a given string is in Lu, then the above algorithm will correctly determine that, halt and say *yes*.

If the given string is not in Lu, then there are three cases:
1) the string doesn't have a TM as a prefix. In this case the above algo correctly detects this fact, and reports the string is not in Lu.
2) the string has a TM prefix, and the TM halts and rejects on the suffix. In this case the above algo correctly reports the string is not in Lu.
3) the string has a TM prefix, but it goes into an infinite loop on the suffix. In this case the above algo also goes into an infinite loop, but that's ok since the string as a whole is not in Lu anyway, and we are just trying to show there exists a TM for only accepting strings in Lu.

From this proof note that if the prefix TM is a DFA or PDA, then our machine will also halt in the 3rd case above, no matter what the suffix is.

-- due to Dr. Bernhard (edited by me)

- **The over-all logic of the proof is as follows:**

1. If L_u were recursive, then so will be $\overline{L_d}$
2. $\overline{L_d}$ is not recursive, because L_d is not r.e.
3. It follows that L_u is not recursive.

The second point was established by the corollary.

The first point was established by the theorem on a preceding slide.

This type of proof is commonly referred to as a *reduction*. Specifically, the problem of recognizing $\overline{L_d}$ was *reduced* to the problem of recognizing L_u

- **Define another language L_h:**

  L_h = {x | x is in {0, 1}* and x = <M,w> where *M* is a TM encoding and *M* halts on *w*}

  Note that L_h is P2 expressed as a language:

  (P2):
  Input: Turing machine *M* with input alphabet Σ and string *w* in Σ*.
  Question: Does *M* halt on *w*?

- **Theorem:** L_h is not recursive.

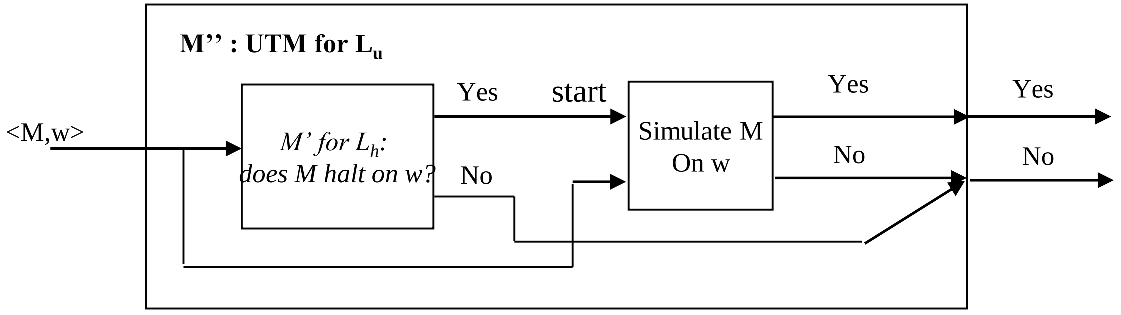
- **Proof:** (by contradiction)
  Suppose that L_h is recursive. Recall that:

  L_h = {x | x is in {0, 1}* and x = <M,w> where *M* is a TM encoding and *M* halts on *w*}

  and

  L_u = {x | x is in {0, 1}* and x = <M,w> where *M* is a TM encoding and *w* is in L(M)}

  Suppose that L_h = L(M') where M' is a TM that always halts. Construct an algorithm (i.e., a TM that always halts) for L_u as follows:



  Suppose that *M'* always halts and L_h = L(M'). It follows that:
  - M'' always halts
  - L(M'') = L_u

  L_u would therefore be recursive, a contradiction. •

- **The over-all logic of the proof is as follows:**

1. If L_h is recursive, then so is L_u

2. L_u is not recursive

3. It follows that L_h is not recursive.

The second point was established previously.

The first point was established by the theorem on the preceding slide.

This proof is also a reduction. Specifically, the problem of recognizing L_u was *reduced* to the problem of recognizing L_h.

[L_u and L_h both are recursively enumerable: for proof see Dr. Shoaff!]

- **Define another language L_q:**

  L_q = {x | x is in {0, 1}*, x encodes a TM M, and M does **not** contain an infinite loop}

  Or equivalently:

  L_q = {x | x is in {0, 1}*, x encodes a TM M, and there exists **no** string w in {0, 1}* such that M does **not** terminate on w}

  Note that:

  $\overline{L_q}$ = {x | x is in {0, 1}*, and either x does **not** encode a TM, or it does encode a TM, call it M, and there exists a string w in {0, 1}* such that M does **not** terminate on w}

  Note that the above languages correspond to the following problem:

  (P0):
  Input: Program P.
  Question: Does P contain an infinite loop?

  Using the techniques discussed, what can we prove about L_q or its' complement?

- **More examples of non-recursive languages:**

L_{ne} = {x | x is a TM M and L(M) is not empty} is r.e. but not recursive.

L_e = {x | x is a TM M and L(M) is empty} is not r.e.

L_r = {x | x is a TM M and L(M) is recursive} is not r.e.

Note that L_r is not the same as L_h = {x | x is a TM M that always halts} but L_h is in L_r.

L_{nr} = {x | x is a TM M and L(M) is not recursive} is not r.e.

Ignore this slide

Lemma: L_d is not recursively enumerable: *[No TM for L_d!!!]*

- **Proof:** (by contradiction)

Suppose that L_d were recursively enumerable. In other words, that there existed a TM M such that:

$$L_d = L(M) \qquad (2)$$

Now suppose that w_j is a string encoding of M. (3)

Case 1) **w_j is in L_d** (4)

By definition of L_d given in (1), either w_j does not encode a TM, or w_j does encode a TM, call it *M*, and w_j is not in L(M). But we know that w_j encodes a TM (3: that's where it came from). Therefore:

$$w_j \text{ is not in } L(M) \qquad (5)$$

But then (2) and (5) imply that **w_j is not in L_d** contradicting (4).

Case 2) **w_j is not in L_d** (6)

By definition of L_d given in (1), w_j encodes a TM, call it *M*, and:

$$w_j \text{ is in } L(M) \qquad (7)$$

But then (2) and (7) imply that **w_j is in L_d** contradicting (6).

Since both case 1) and case 2) lead to a contradiction, no TM *M* can exist such that L_d = L(M). Therefore L_d is not recursively enumerable. •

Roger's TM for balanced parenthesis:

	(	)	B
findPair	(findPair2, "(", R)	-	(final, B, R)
findPair2	(findPair2, "(", R)	(removePair, ")", L)	-
removePair	(fetch, "(", R)	(fetch, ")", R)	(goBack, B, L)
fetch	(retrieve, "(", R)	(retreive, ")", R)	(retreive, B, R)
retreive	(returnOpen, "(", L)	(returnClosed, ")", L)	(returnBlank, B, L)
returnOpen	(writeOpen, "(", L)	(writeOpen, ")", L)	(writeOpen, B, L)
returnClosed	(writeClosed, "(", L)	(writeClosed, ")", L)	(writeClosed, B, L)
returnBlank	(writeBlank "(", L)	(writeBlank, ")", L)	(writeBlank, B, L)
writeOpen	(removePair, "(", R)	(removePair, "(", R)	-
writeClosed	(removePair, ")", R)	(removePair, ")", R)	-
writeBlank	(removePair, B, R)	(removePair, B, R)	-
goBack	-	-	(backAgain, B, L)
backAgain	-	-	(seekFront, B, L)
seekFront	(seekFront, "(", L)	(seekFront, ")", L)	(findPair, B, R)
final*	-	-	-

On 111 111 as a TM encoding

<Quote> It was ambiguous, in my opinion, based on the definition in the Hopcroft book, i.e., the definition in the Hopcroft book was not clear/precise enought to

account this special case. I don't have the book in front of me right now, but I think this is the example I used in class: Consider the TM that has exactly one state, but no transitions. Perfectly valid TM, and it would give us this encoding (111111). In that case the encoded machine would accept sigma* because the highest numbered state would be q0, the only state, and that would be the final state under the Hopcroft encoding. Now consider the TM that has exactly two states, but no transitions. Also a perfectly valid TM, and it would give us the same encoding. In that case the encoded machine would not accept anything because the final state is q1 (highest numbered state), and there is no way to get to it. I used it only as a way to raise that issue in class, i.e., the the Hopcroft definition is a bit ambiguous in this case.

One way to resolve the ambiguity is to require the encoding to specifically specify the final state (at the end or something). In that case, 111111 isn't even a valid TM, since it doesn't specify the final state. Another related question is, does a TM even have to have any states at all to be a valid TM? The encoding would have to be able to isolate that as a unique string also. <End Quote>

Phil Bernhard