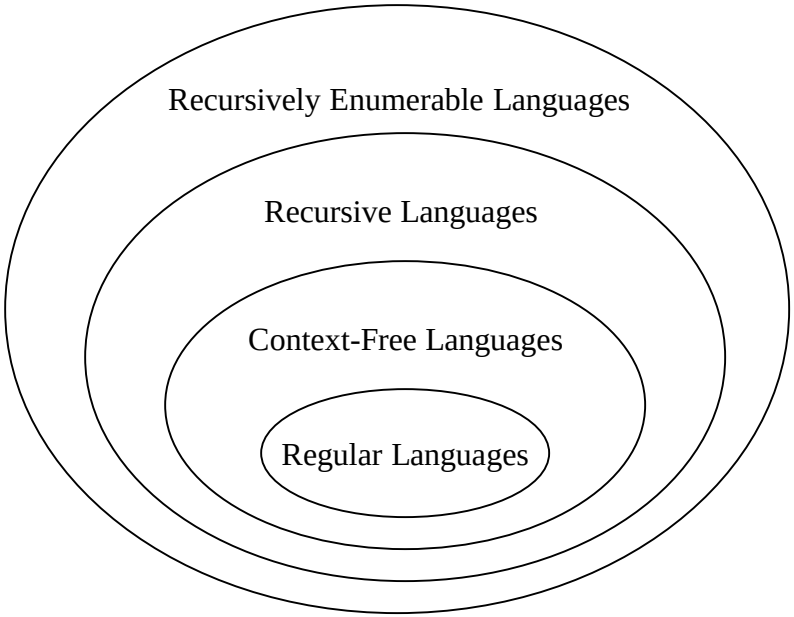
Introduction to Turing Machines

Reading: Chapters 8 & 9

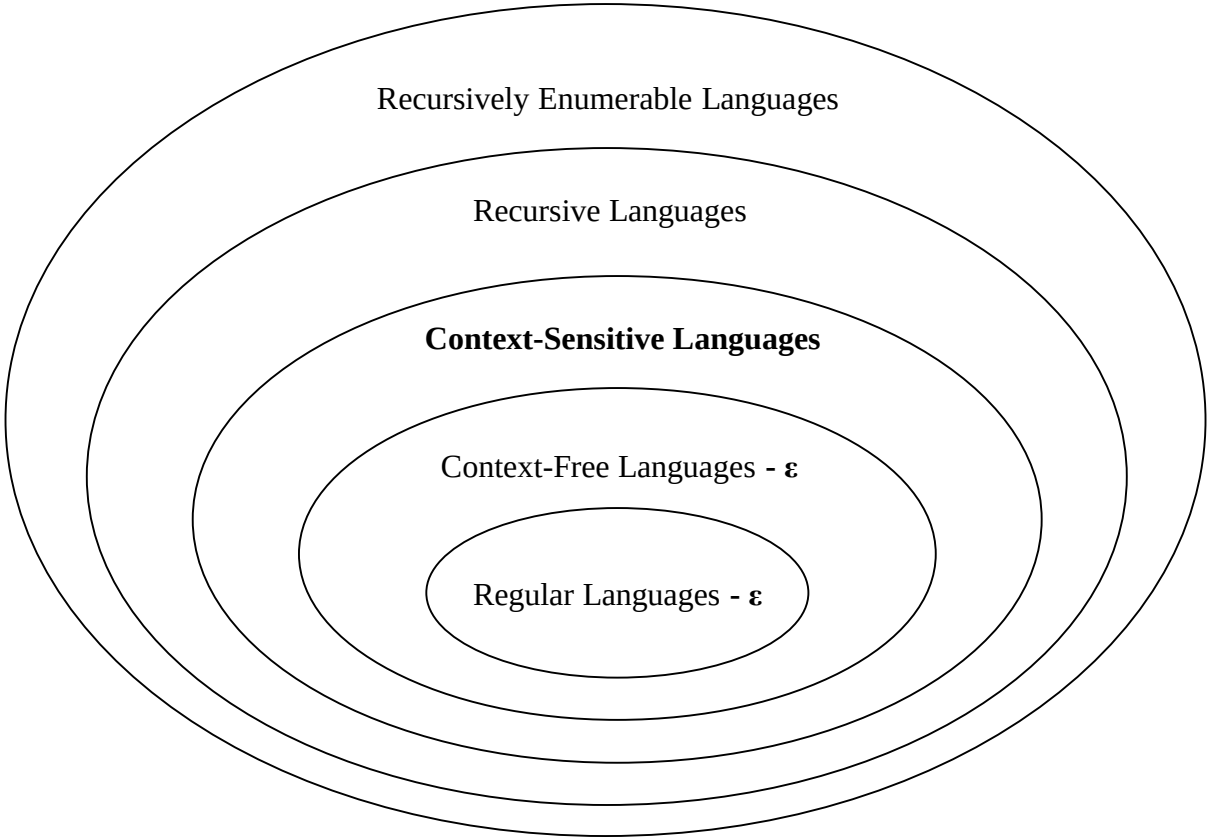
Turing Machines (TM)

- Generalize the class of CFLs:

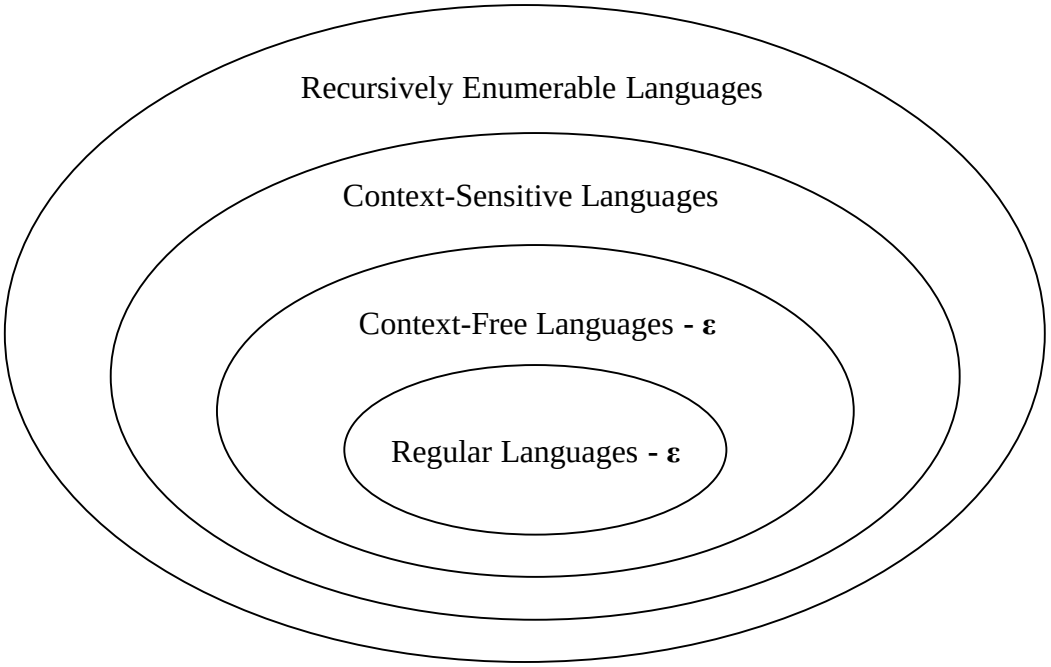


- Note that there are non-recursively enumerable languages.

- **Another Part of the Hierarchy:**



- Chomsky Hierarchy:
  - 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.



- **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.

- Informally, the notion of an "effective procedure" is intended to represent a program that a real, general purpose computer could execute:
  - Finitely describable
  - Well defined, discrete, "mechanical" steps
  - Always terminates

- TMs formalize the above notion, i.e., the computing capability of a general purpose computer.

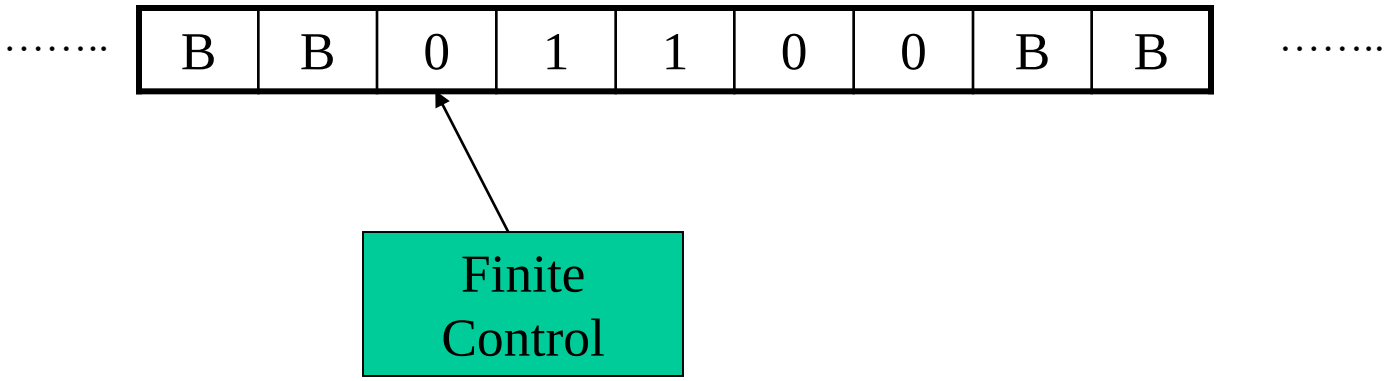
- Stated another way, *if something can be computed in any reasonable sense at all, then it can be computed by a TM.*

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

- There are many other computing models, but all are equivalent to or subsumed by TMs; for many models, this has been proven.

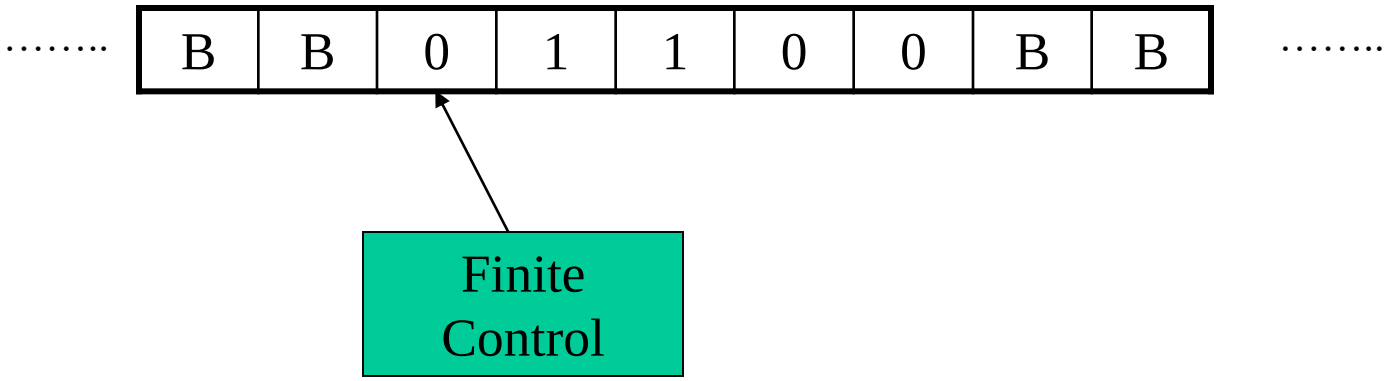
- The Church-Turing Thesis asserts that there is no more powerful machine (technically uproven).

Deterministic Turing Machine (DTM)



- Two-way, infinite tape, broken into cells, each containing one symbol.
- Two-way, read/write tape head.
- Finite control, i.e., a program, containing:
  - position of the tape head
  - current symbol being scanned
  - current state

Deterministic Turing Machine (DTM)



Execution:

- An input string is placed on the tape, padded to the left and right infinitely with blanks
- Tape head is positioned at the left end of input string
- In one move, depending on the current state and symbol being scanned, the TM:
  - changes state
  - prints a symbol over the cell being scanned, and
  - moves its' tape head one cell left or right.

- Many modifications possible (to be discussed later).

Formal Definition of a DTM

- A DTM is a seven-tuple:

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

Q	A finite set of states
Γ	A finite tape alphabet
B	A distinguished blank symbol, which is in Γ
Σ	A finite input alphabet, which is a subset of $\Gamma - \{B\}$
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 $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,R\}$

  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
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 "scan right" state
  – q_1 is the verify 0 state

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

	0	1	B
q_0	$(q_0, 0, R)$	$(q_0, 1, R)$	(q_1, B, L)
q_1	$(q_2, 0, R)$	-	-
q_2	-	-	-

- **Sample Computation:** (on 0010)

$q_0 0010 \vdash 0q_0 010$
$\vdash 00q_0 10$
$\vdash 001q_0 0$
$\vdash 0010q_0$
$\vdash 001q_1 0$
$\vdash 0010q_2$

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

	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	-	-	-	-	-

- **Sample Computation:** (on 0011)

$q_0 0011 \vdash X q_1 011$
$\vdash X 0 q_1 11$
$\vdash X q_2 0 Y 1$
$\vdash q_2 X 0 Y 1$
$\vdash X q_0 0 Y 1$
$\vdash X X q_1 Y 1$
$\vdash X X Y q_1 1$
$\vdash X X q_2 Y Y$
$\vdash X q_2 X Y Y$
$\vdash X X q_0 Y Y$
$\vdash X X Y q_3 Y$
$\vdash X X Y Y q_3$
$\vdash X X Y Y B q_4$

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

	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	-	-	-	-	-

  - The TM basically matches up 0's and 1's
  - q_0 is the *"check off a zero"* state
  - q_1 is the *"scan right, find and check off a matching 1"* state
  - q_2 is the *"scan left"* state
  - q_3 is the "*verify no more ones"* state
  - q_4 is the final state

- **Other Examples:**

000111	00	
11	001	
011	010	110

- What kinds of things can TMs do?
  - Accept languages (as above)
  - Simulate DFAs, NFAs and PDAs
  - Compute functions and operators (+, *, etc)
  - Have subroutines, pass parameters
  - Simulate arbitrary computer programs

- **Exercises (some from the class website):** 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 2 0's}
  - {w | w is in {0,1}* and w contains at least one 0 and one 1}
  - {w | w is in {0,1}* and w contains at least one occurrence of the substring 00}
  - A TM that adds 1 to a given binary number
  - Suppose the input to a Turing machine consists of two n-bit binary numbers separated by a # character. Give a deterministic Turing machine that will determine if the first binary number is larger than the second. Note that the turning machine should output 1 to the right of the second number if the answer is yes, and a 0 if the answer is no.

Be Careful!

- Note that, just like a computer program, a TM can contain an infinite loop:

	0	1	B
q_0	$(q_0, 0, R)$	$(q_0, 1, R)$	(q_0, B, R)

- This is somewhat equivalent to:

```
while (true) ;
```

- As with more general programs, more complicated TM infinite loops can occur.

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
  - α_1, which is in Γ^*, is the tape contents up to the left of the tape head, starting at the leftmost non-blank symbol
  - α_2, which is in Γ^*, is the tape contents from, and including, the current position of the tape head, up to the rightmost non-blank symbol
  - The tape head is currently scanning the first symbol of α_2
  - At the start of a computation $\alpha_1 = \varepsilon$
  - If $\alpha_2 = \varepsilon$ then a blank is being scanned

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

$q_0 0011$ $Xq_1 011$ $X0q_1 11$ $Xq_2 0Y1$ $q_2 X0Y1$

$Xq_0 0Y1$ $XXq_1 Y1$ $XXYq_1 1$ $XXq_2 YY$ $Xq_2 XYY$

$XXq_0 YY$ $XXYq_3 Y$ $XXYYq_3$ $XXYYBq_4$

Next-Move Function (Informal Definition)

- **Definition:** Let I and J be instantaneous descriptions. If J follows from I by exactly one transition, then I |— J.

  $q_0 0011 \vdash X q_1 011$ (see TM example #2)
  $X q_2 0 Y 1 \vdash q_2 X 0 Y 1$
  $XXYY q_3 \vdash XXYYB q_4$

- **Definition:** |—* is the reflexive and transitive closure of |—.
  - I |—* I for each instantaneous description I
  - If I |— J and J |—* K then I |—* K

- Intuitively, if I and J are instantaneous descriptions, then I |—* J means that J follows from I by zero or more transitions.

  $q_0 0011 \vdash^* XXYYB q_4$
  $XX q_1 Y 1 \vdash^* X q_2 XYY$

- **Definition:** Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a TM, and let w be a string in Σ^*. Then w is *accepted* by M iff

  $$q_0 w \vdash^* \alpha_1 p \alpha_2$$

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

- **Definition:** Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a TM. The *language accepted by M*, denoted L(M), is the set

  $$\{w \mid w \text{ is in } \Sigma^* \text{ and } w \text{ is accepted by } M\}$$

Accepting Computations & Halting

- Note that much like with a DFA or a PDA, a TM might pass through a final state many times during a given computation.

- On the other hand, unlike with a DFA or a PDA, in the previous definitions there is no requirement that a TM read all its' input or halt to accept a string; simply arriving at a final state gets the string accepted.

- In fact, a TM could even accept a string, and then go into an infinite loop.
  - Example: Modify TM #1 to go into an infinite loop upon acceptance.
  - But that's not helpful…in fact, it's downright weird!
  - In technical terms, it's *non-intuitive,* we want our algorithms to terminate; you can't claim your algorithm "solves" a problem or "accepts" a language if it doesn't halt.

Accepting Computations & Halting

- So is there a way to get a TM to terminate when it accepts?
- Sure is, in fact, it's simple – don't allow any transitions out of a final state.
  - *Henceforth, this will be our assumption.*
  - Technically, this is a modification of the formal definition of a TM.
- So now, for all accepting computations, all TMs halt.
  - This is a good thing, it's intuitive
  - It's the way algorithms should work
  - We like it, the Consumer Products Safety Commission likes it, Walmart complies with it…everybody is happy…

Non-Accepting Computations & Halting

- How about non-accepting computations?
- Obviously, we would like a TM to halt on all inputs.
- However, with the current formal definition the language of a TM:
  - If x is not in L(M) then M may halt in a non-final state, or
  - M may also go into an infinite loop! (give a modified version of TM #1)
  - This also not particularly intuitive or desirable.

Non-Accepting Computations & Halting

- Can't we just "outlaw" infinite loops?
- Isn't there a simple way to do this, like we did for accepting computations, something that is easy to check for?
- That's what programmers do, isn't it?
- As we shall see, not really… "outlawing" infinite loops is technically impossible.

TM Computations

- In summary, given an arbitrary Turing machine M, and a string x, there are three possible types of computations:
  - M terminates in a final state, in which case we say x is accepted and in L(M).
  - M terminates in a non-final state, in which case we say x is not accepted and is not in L(M).
  - M goes into an infinite loop, in which case we say x is not accepted and is not in L(M).

- Stated another way, given an arbitrary Turing machine M, and a string x:
  - If x is in L(M) then M will halt in a final state.
  - If x is not in L(M) then M may enter an infinite loop, or halt in a non-final state.

- By the way, how difficult would it be to simulate an arbitrary TM with a (Java, C++, etc) computer program?

Recursive and Recursively Enumerable Languages

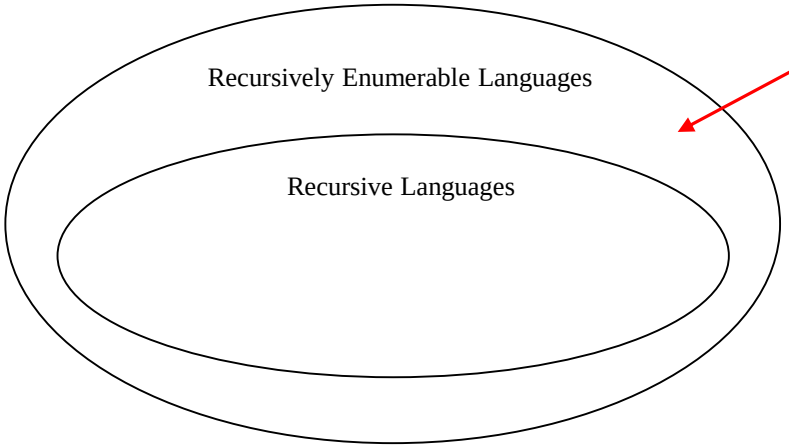
- **Definition:** Let L be a language. Then L is *recursively enumerable* if there exists a TM M such that L = L(M).
  - If L is r.e. then L = L(M) for some TM M, and
    - If x is in L then M halts in a final (accepting) state.
    - If x is not in L then M may halt in a non-final (non-accepting) state, or 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 a non-final (non-accepting) state.

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.

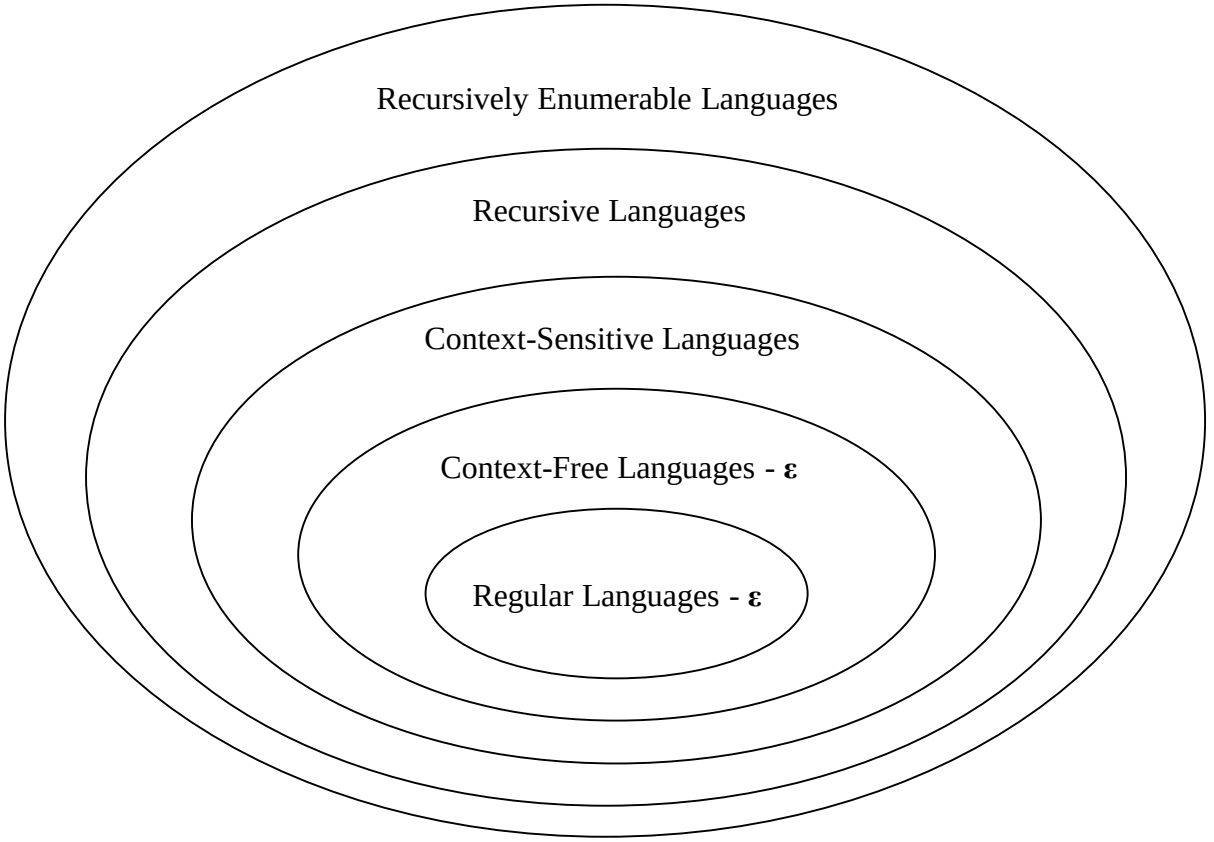
- **By definition:**

What would it mean for a language to be here, i.e., r.e. – r?



- But is the inclusion proper?
- As we shall see, it is…

- **The Language Hierarchy (so far):**



- Let L be a language.
- Saying that L = L(M) tells us:
  - For any string in L, M will halt and accept
  - For any string not in L, M will not halt and accept; in other words, M might halt in a non-final state or go into an infinite loop.
- In that sense, saying L = L(M) tells us more about accepted strings.
- Keep that in mind throughout the following…

- **Observation:** Let L be an r.e. language. Then there is an infinite list M_0, M_1, … of TMs such that $L = L(M_i)$.
  - Many of these are trivial modifications of each other.
  - Some of these contain infinite loops on non-accepted strings (recall the modifications to TM #1).
  - Some of these *might* halt on all inputs, in which case L is also recursive.
  - Conceivably, none of these halt on all inputs, in which case L is in r.e.-r

- **Observation:** Let L be a recursive language. Then there is an infinite list M_0, M_1, … of TMs such that $L = L(M_i)$.
  - Many of these are trivial modifications of each other.
  - Some of these also contain infinite loops on non-accepted strings (recall the modifications to TM #1).
  - However, at least one will halt on all inputs, by definition (in fact, an infinite number do).

- **Question:** Let L be a recursive language. Then there is a list M_0, M_1, … of TMs such that L = L(M_i). 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. Then there is a list M_0, M_1, … of TMs such that L = L(M_i). 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, which it could be, then it might.

- **Question:** Let L be a recursive enumerable language that is not recursive (L is in r.e. – r). Then there is a list M_0, M_1, … of TMs such that L = L(M_i). 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.

- **Let M be a TM.**
  - Question: Is L(M) *well-defined*?
  - Answer: Given any string w, either w is in L(M) or not, so in that sense, it is.
  - 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 that L(M) is also recursive. Does that mean that M always halts?
  - Answer: Not necessarily. However, some other TM M' must exist such that L(M') = L(M) and M' always halts (recall TM #1 and it's modified version).
  - 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.

  - 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.

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 TM model**

**In terms of the languages accepted, but not necessarily in terms of performance.*

Closure Properties for Recursive and Recursively Enumerable Languages

- **TMs Model General Purpose Computers: (Church-Turing Thesis)**
  - 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 given language L recursive?
  - This equates to - can a C/Java/etc. program be written that always halts and accepts L?
  - Is $0^n1^n2^m3^n4^m$ recursive?

One More Time!

- *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?

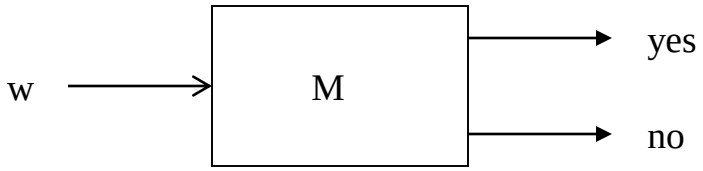
Another example:

- Is $0^n1^m2^n3^m4^{n+m}$ recursive?

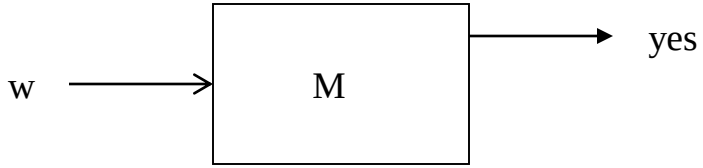
Can a C/C++/Java/etc. program be written that always halts and accepts it?

The Chip Metaphor

- **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" 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 "chip" that has one output.

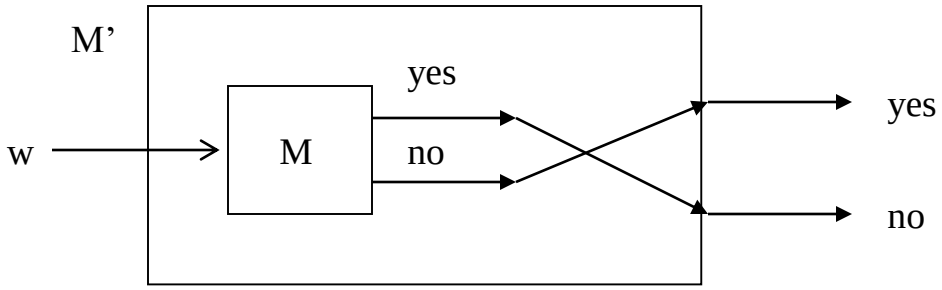


  - Conceivably, M could be provided with an output for "no," but this output cannot be counted on. This would make the following proofs more complicated, so consequently, we leave it off.

- **Theorem:** 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:** Suppose L is recursive, and let M be a TM such that L = L(M) where M always halts.

  Construct TM M' as follows:



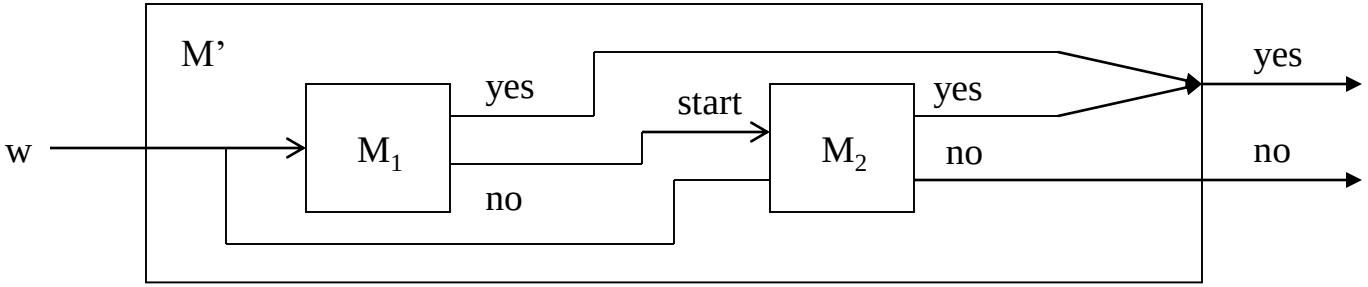
- **Note That:**
  - M' accepts iff M does not
  - M' always halts since M always halts

  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! In fact, given our requirement that there be no transitions out of a final state, the resulting machine is not even a TM.
  - Even if we revert back to the original definition of a TM, it is easy to show that this construction doesn't work (exercise).

- **Theorem:** 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 $L_2 = L(M_2)$ where M_1 and M_2 always halt. Construct TM M' as follows:

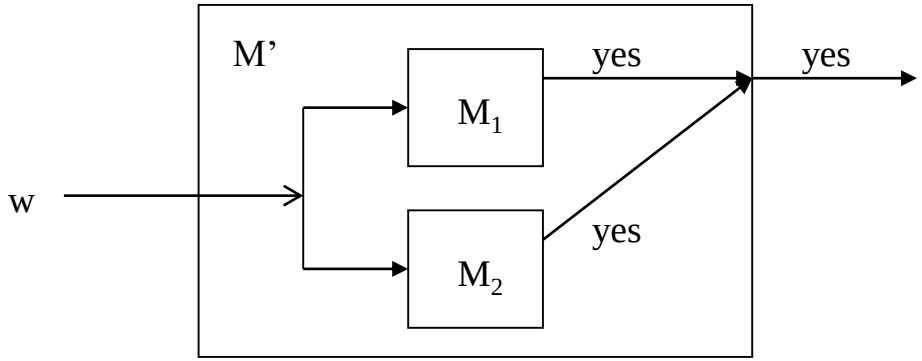


- **Note That:**
  - L(M') = L(M_1) U L(M_2)
    - L(M') is a subset of L(M_1) U L(M_2)
    - L(M_1) U L(M_2) is a subset of L(M')
  - M' always halts since M_1 and M_2 always halt

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

- **Theorem:** 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:



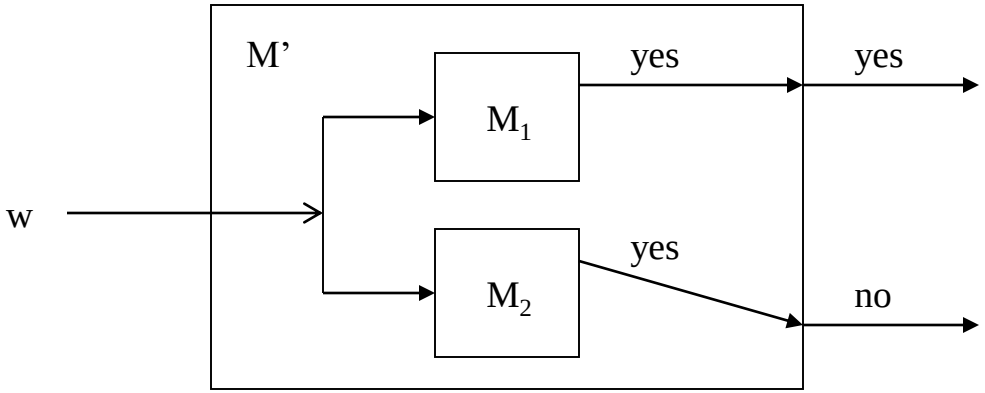
- **Note That:**
  - L(M') = L(M_1) U L(M_2)
    - L(M') is a subset of L(M_1) U L(M_2)
    - L(M_1) U L(M_2) is a subset of L(M')
  - M' halts and accepts iff M_1 or M_2 halts and accepts

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

- **Question:** How do you run two TMs in parallel? Could you do it with a Java program?

- **Theorem:** 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:

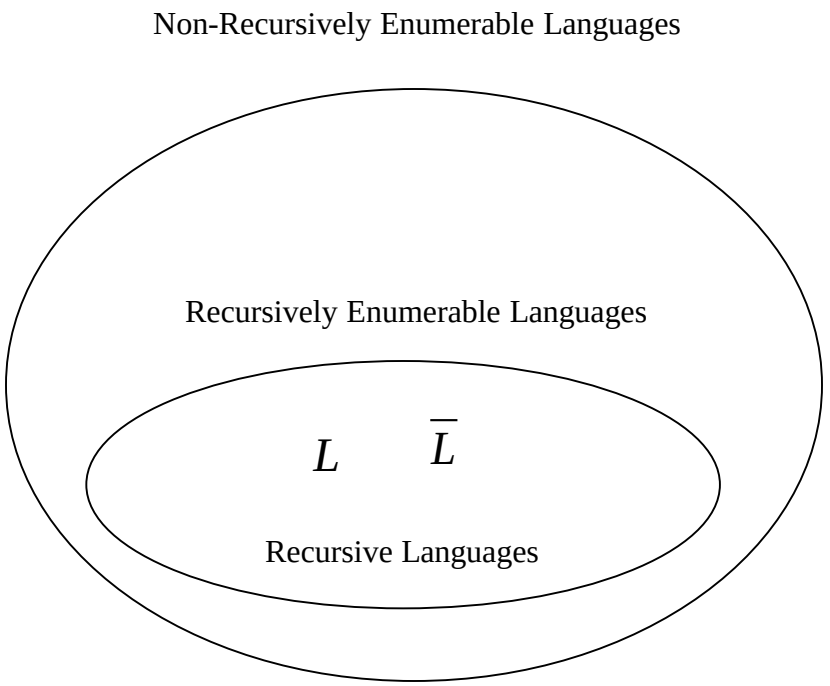


- **Note That:**
  - L(M') = L
    - L(M') is a subset of L
    - L is a subset of L(M')
  - M' always halts since M_1 or M_2 halts for any given string

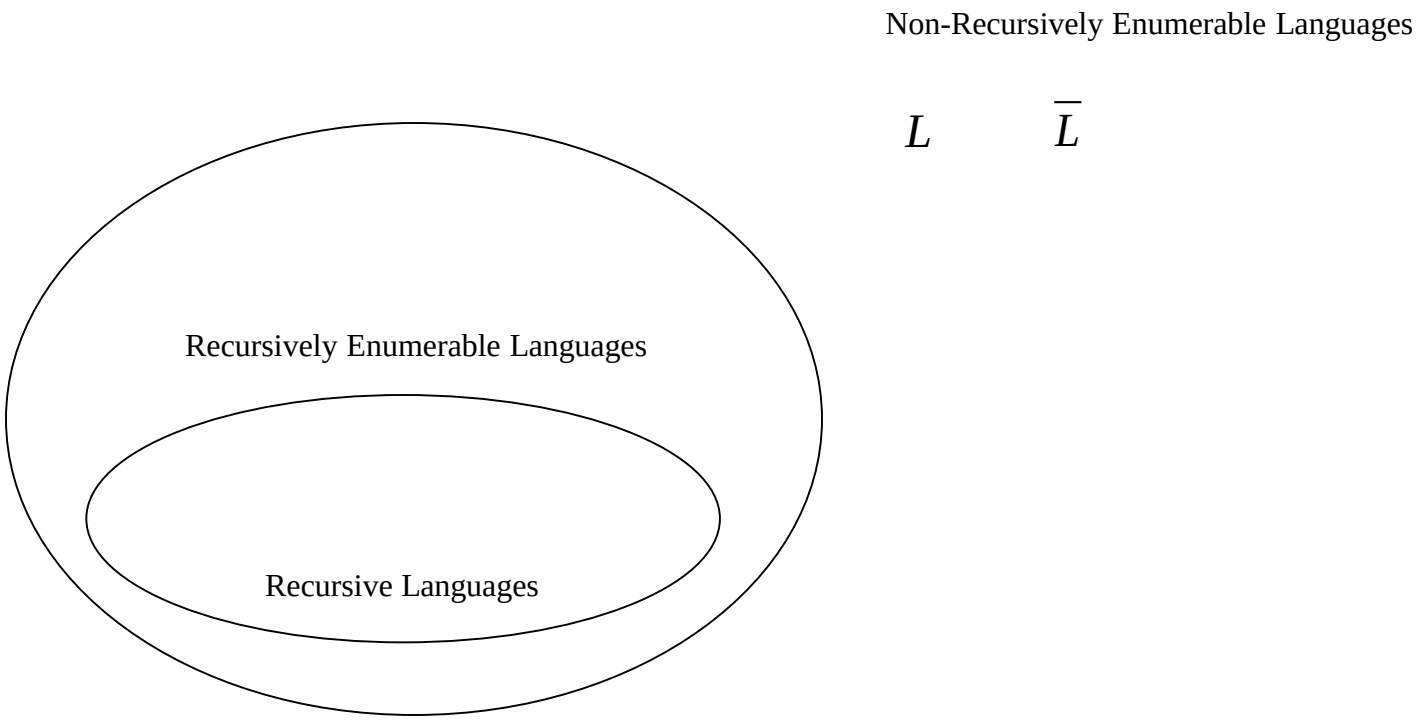
It follows from this that L (and therefore its' complement) is recursive.•

- **Corollary:** 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,

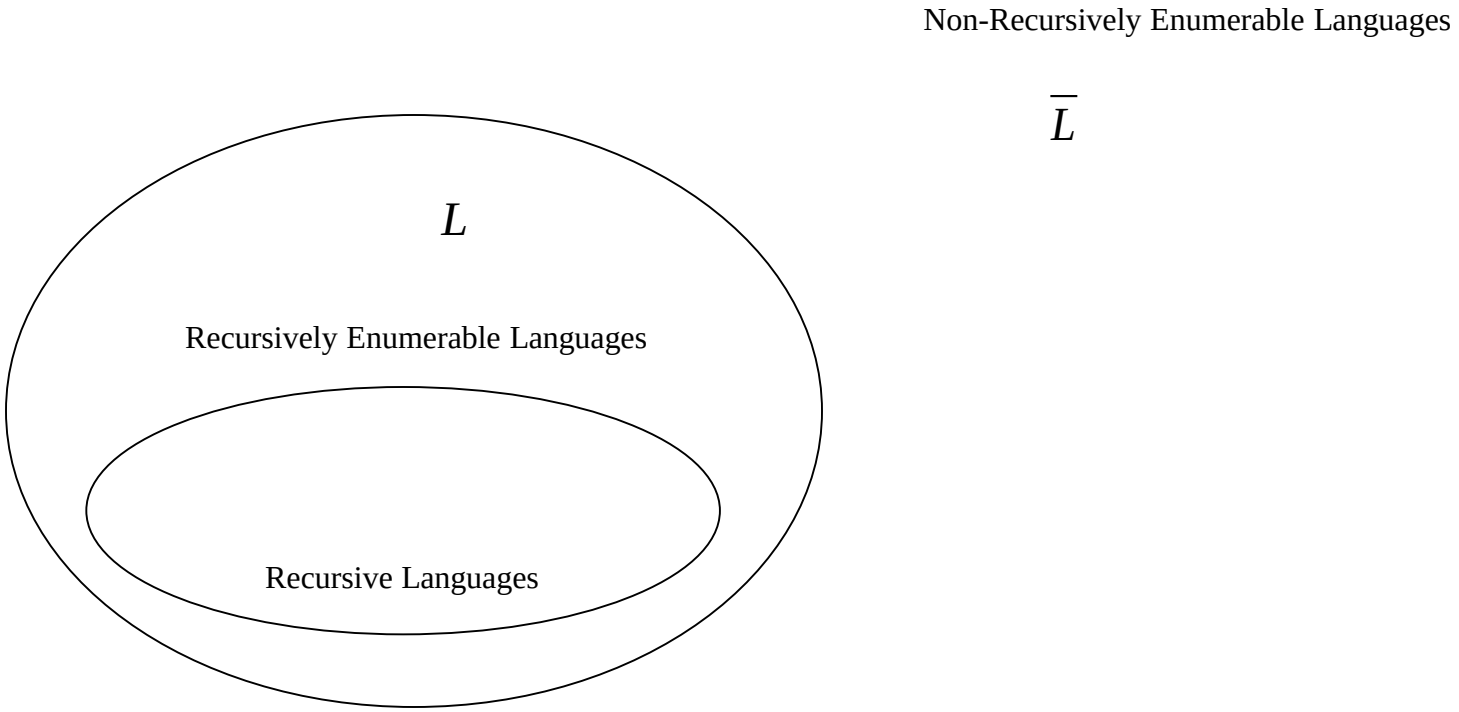
- **In terms of the hierarchy:** (possibility #1)



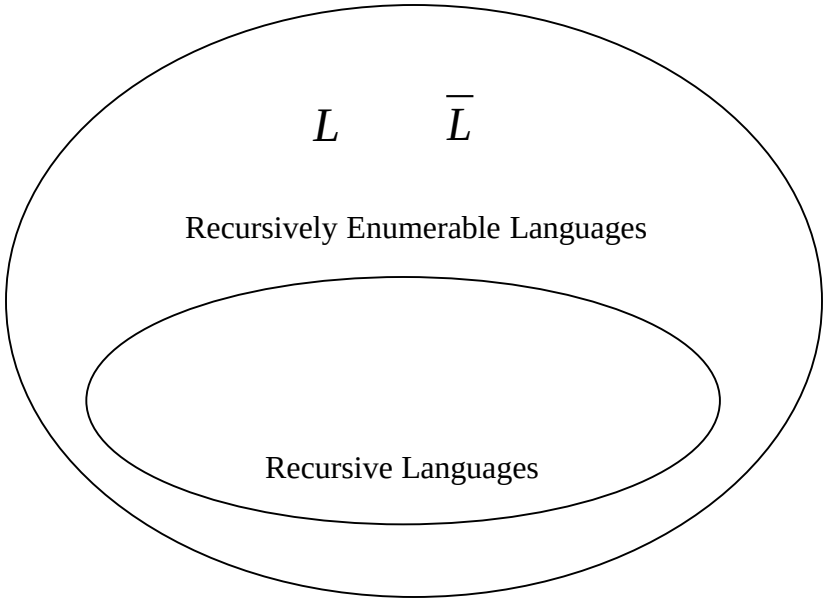
- **In terms of the hierarchy:** (possibility #2)



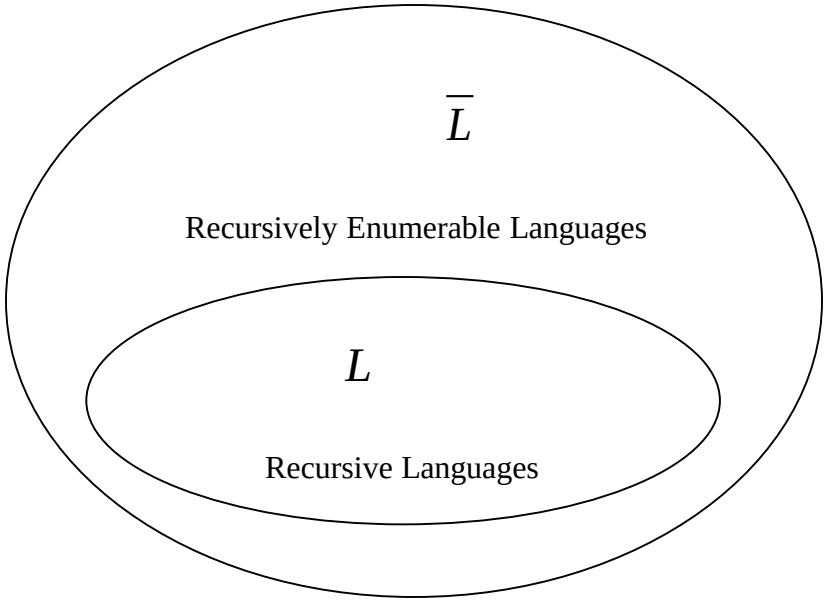
- **In terms of the hierarchy:** (possibility #3)



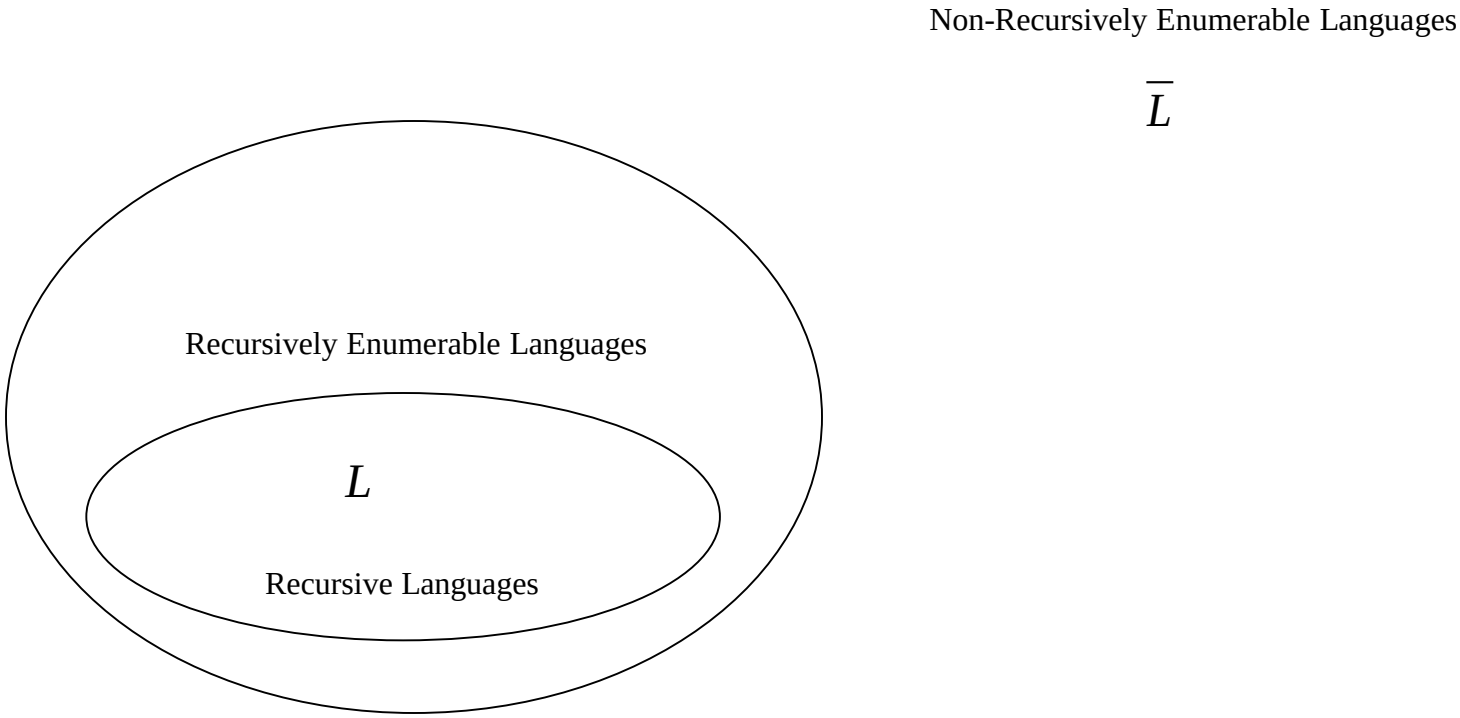
- **In terms of the hierarchy:** (Impossibility #1)



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



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



- We would now like to show that some languages are not recursive.
- Additionally, we would also like to show that some languages are not even recursively enumerable.
- Sometimes showing that a language is not recursive is a bit tricky…
- The previous slides give us three indirect approaches for showing that a language is not recursive, which is sometimes easier:
  - Show that the language's complement is not recursive
  - Show that the language's complement is recursively enumerable but not recursive
  - Show that the language's complement is not recursively enumerable

The Halting Problem - Background

- **Definition:** A decision problem is a problem having a yes/no answer:
  - Given a number x, is x even?
  - Given a list of numbers, is that list sorted?
  - 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 are typically at least as difficult to solve.
  - Decision problems can be encoded as languages.

- Going forward, note that the following terms and phrases are analogous:

Algorithm	-	A halting TM program
Decision Problem	-	A language
(un)Decidable	-	(non)Recursive

Statement of the Halting Problem

- **Practical Form:** (P0)
  Input: Program P.
  Question: Does P contain an infinite loop?

- **Theoretical Form:** (P1)
  Input: Turing machine M with input alphabet Σ.
  Question: Does there exist a string w in Σ* such that M does not halt on w?

Another version…

- **Practical Form:** (P3)
  Input: Program P and input I.
  Question: Does P terminate on input I?

- **Theoretical Form:** (P4)
  Input: Turing machine M with input alphabet Σ and string w in Σ*.
  Question: Does M halt on w?

Statement of the Halting Problem

- **A Related Problem We Will Consider First:** (P5)
  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 (the DFA version, that is) decidable? Yes!

- **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:**
  - P5 will correspond to the language L_u
  - Proving P5 (un)decidable is equivalent to proving L_u (non)recursive

Converting the Problem to a Language

- Let $M = (Q, \Sigma, \Gamma, \delta, q_1, B, \{q_n\})$ be a TM, where

  $Q = \{q_1, q_2, \ldots, q_n\}$
  $\Sigma = \{x_1, x_2\} = \{0, 1\}$
  $\Gamma = \{x_1, x_2, x_3\} = \{0, 1, B\}$

- Encode:

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

  as:

  $0^i10^j10^k10^l10^m$

- 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

  - Example:

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

    Is encoded as:

    00100100010100

  - Note that 1's have a special role as a delimiter.

- A complete TM M can then be encoded as:

  $111code_1 11code_2 11code_3 11 \ldots 11code_r 111$

  where each code i is one transitions' encoding. Let this encoding of M be denoted by <M>.

	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	-	-	-

11101010101001101001010010011010001001000101100101000101001111

01100001110001

111111

Similarly, we could encode DFAs, NFAs, or any machine model for that matter.

	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	-	-	-

111010101010011010010100100111010001001000101100101000101001111

01100001110001

111111

Similarly, we could encode DFAs, NFAs, or any machine model for that matter.

- **Definition:**

  $L_{tm} = \{x \mid x \text{ is in } \{0, 1\}^* \text{ and } x \text{ encodes a TM}\}$

  - Question: Is L_{tm} recursive?
  - Answer: Yes.

  - Question: Is L_{tm} decidable:
  - Answer: Yes (same question).

- **Definition:** (similarly)

  $L_{dfa} = \{x \mid x \text{ is in } \{0, 1\}^* \text{ and } x \text{ encodes a DFA}\}$

  - Question: Is L_{dfa} recursive?
  - Answer: Yes.

  - Question: Is L_{dfa} 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?**

  1110101010100111010010100100111010001001000101100101000101001110 1110

  1110101010100111010010100100111010001001000101100101000101001111001101 11

  1110101010100111010010100100111010001001000101100101000101001111 00

  01100001110001

  01100001110000

- **Compare P5 and L_u:**

  (P5):
  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)}

- **Notes:**
  - L_u is P5 expressed as a language
  - Asking if L_u is recursive is the same as asking if P5 is decidable.
  - We will show that L_u is not recursive, and from this it will follow that P5 is un-decidable.
  - Note that $L_{u\text{-}dfa}$ is recursive if M is a DFA.

- Define another language L_d as follows:

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

- 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?**

  1110101010100110100101001001101000100100010110010100010100111

  01100001110001

  1110101010100110100101001001101000100100010110010010001001001111

  How about the string representing TM #2?

- **Lemma:** L_d is not recursively enumerable:

- **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.

Case 1) **w_j is in L_d** (3)

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 (that's were it came from). Therefore:

$$w_j \text{ is not in } L(M) \qquad (4)$$

But then (2) and (4) imply that **w_j is not in L_d** contradicting (3).

Case 2) **w_j is not in L_d** (5)

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

$$w_j \text{ is in } L(M) \qquad (6)$$

But then (2) and (6) imply that **w_j is in L_d** contradicting (5).

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

- As a small sanity check, what would happen if you tried to write a Java, C or C++ program for L_d ?

- Now consider $\overline{L_d}$:

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

- **Which of the following are in $\overline{L_d}$?**

  1110101010100110100101001001101000100100010110010100010100111

  01100001110001

  1110101010100110100101001001101000100100010110010010001001001 00111

  How about the string representing TM #2?

- **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.

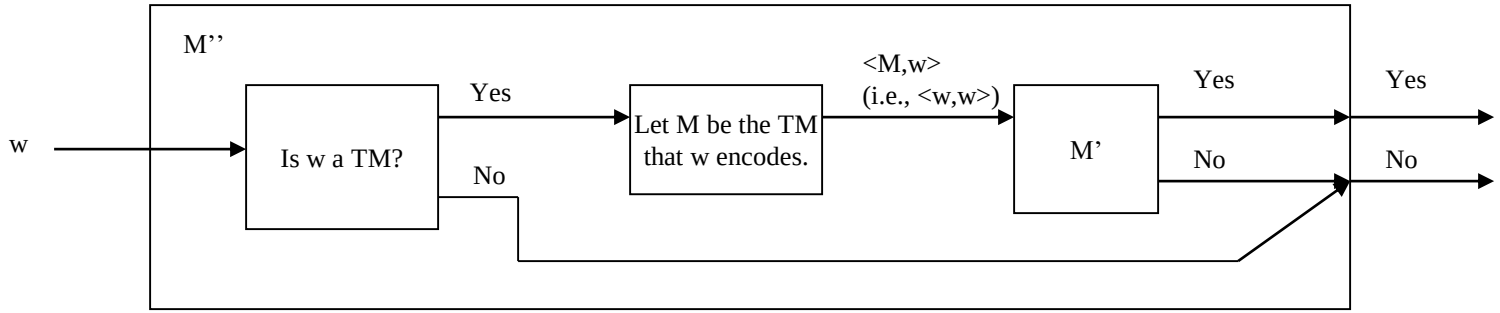
- Question: Is $\overline{L_d}$ recursively enumerable?

- **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)}

  Then $L_u = \underline{L}(M')$ 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.•

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

1. If L_u is recursive, then so is $\overline{L_d}$
2. $\overline{L_d}$ is not recursive
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 the 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

- **Stated a slightly different way:**

    1. If *M'* exists, then so *M''*

    2. *M''* does not exist

    3. It follows that *M*' does not exist.

- **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 P4 expressed as a language:

  (P4):
  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.

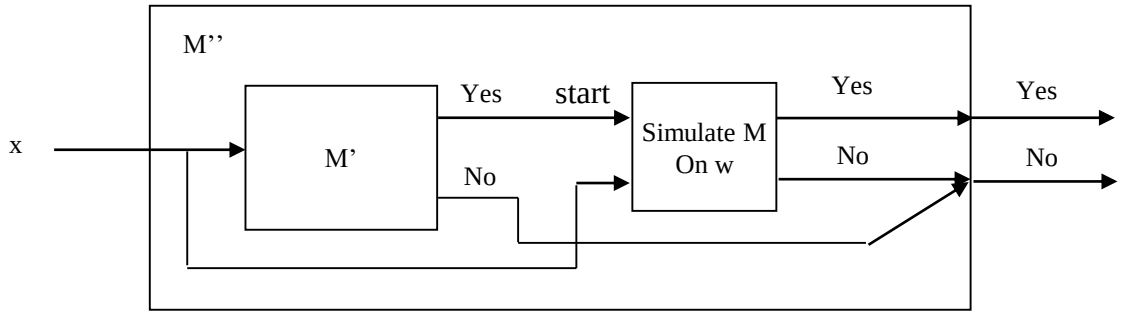
- **Proof:** (by contradiction)
  Suppose that L_h is recursive. Recall that:

  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.

- **Define another language L_t:**

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

Or equivalently:

L_t = {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:

$\overline{L_t}$ = {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 problem P0:

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

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

- **Define another language L_t: (slightly different)**

L_t = {x | x is in {0, 1}*, x encodes a TM M, and M contains an infinite loop}

Or equivalently:

L_t = {x | x is in {0, 1}*, x encodes a TM M, and there exists a string w in {0, 1}*
such that M does not terminate on w}

Note that:

$\overline{L_t}$ = {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 **no** string w in {0, 1}* such that M does **not** terminate on w}

Note that the above languages correspond to the problem P0:

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

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