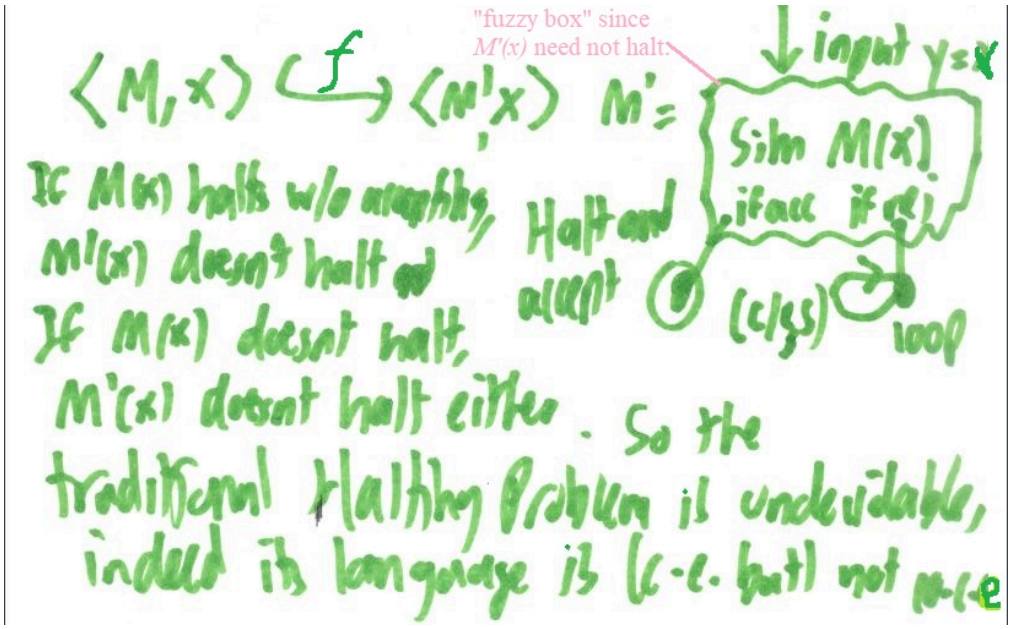
CSE396 Spr26 Lecture Thu. Apr. 16: Doing Mapping Reductions

Resuming with the function f reducing A_{TM} to HP_{TM}, I use the acronym "**CCC**" for the three things one needs to say: **[The entire lecture was done on the chalkboard, brown parts for later.]**

1. **C**onstruction: how M' is built from M, so as to define $f(\langle M, w\rangle) = \langle M', w\rangle$.
2. **C**omputability: Often we could say this is "obvious", but it helps to give knowledge of the **c**omplexity of the reduction too. E.g., the mappings f_1 and f_4 above are super-simple, while the Cartesian-product f_2 is not so simple---but its quadratic time counts as "polynomial time." The mapping f_3 involves converting any given NFAs into DFAs, so it is exponential time, but still counts as computable. This mapping f is super-simple.
3. **C**orrectness: the "$x \in A \iff f(x) \in B$" part, where here the "x" is $\langle M, w\rangle$. It often helps to break the " $\iff$ " into two implications going from the source problem to the target problem. So to show that M accepts $w \iff M'$ on input w halts, we verify:.

M accepts $w \implies M(w)$ goes to $q_{acc} \implies M'(w)$ goes to its own q'_{acc} as well $\implies M'(w)\downarrow$.
M does not accept $w \implies$ either $M(w)\uparrow$ or $M(w)$ goes to $q_{rej} \implies M'(w)\uparrow$ either way.

Put together, we have that $\langle M, w\rangle \in A_{TM} \iff \langle M', w\rangle \in HP_{TM}$.

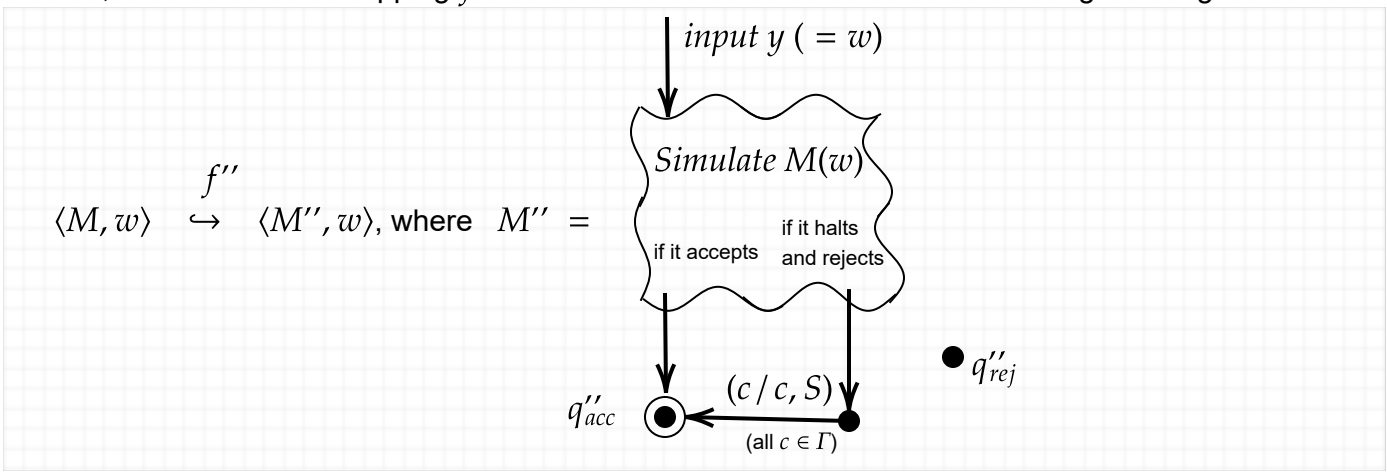


This finally shows that the classic Halting Problem is undecidable.

An important self-study question: **Does the same f also reduce HP_{TM} back to A_{TM}?**

That would need us to say that for all M and w, $M(w)\downarrow \iff M'$ **accepts** w. But that is **not** what happens in the code constructed by the above f mapping.

Instead, we make a new mapping f'' with a different "code modification" that brings this logic about:



This mapping f'' is equally super-simple to compute: it adds arcs from the old q_{rej} to the accepting state rather than loops at q_{rej}. The correctness logic is:

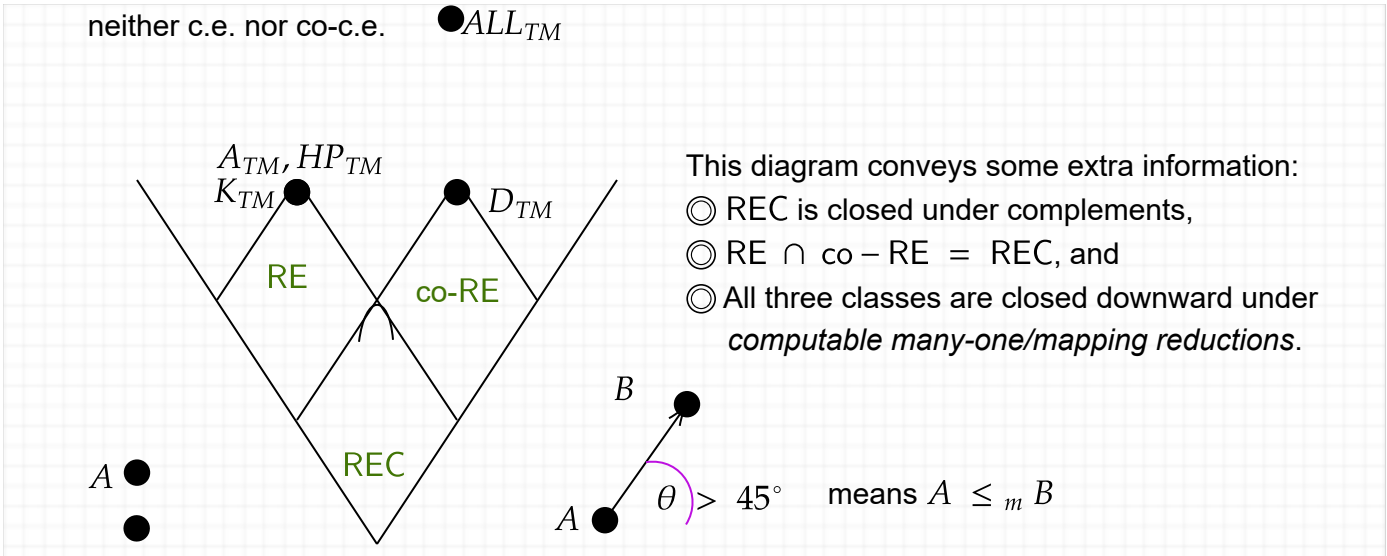
$M(w)\downarrow \implies M(w)$ goes to q_{acc} or to $q_{rej} \implies M''(w)$ goes to q''_{acc} either way $\implies M''$ accepts w.
$M(w)\uparrow \implies M''(w)$ does not halt either, so M'' does not accept w.

This entitles us to say $\langle M, w\rangle \in HP_{TM} \iff \langle M'', w\rangle \in AP_{TM}$. ☒

Thus, in fact, the Acceptance and Halting Problems are **mapping equivalent**, for which we write

$$A_{TM} \equiv_m HP_{TM}.$$

This underscores why, historically, "accepting" and "halting" were considered the same thing, and why accepting states are called "final" states. We can show mapping equivalence graphically by putting the "dots" for each language in the same place in our diagrams:



This diagram conveys some extra information:
◎ REC is closed under complements,
◎ RE ∩ co − RE = REC, and
◎ All three classes are closed downward under *computable many-one/mapping reductions*.

Actually, K_{TM} is mapping-equivalent to A_{TM} as well. This may seem surprising because K_{TM} has "less stuff": its **instance type** is "just an M" rather than "an M and a w". The converse reduction $A_{TM} \leq_m K_{TM}$ will be an incidental benefit of the "All-Or-Nothing Switch" ***reduction design pattern*** below. Thus we can move its dot into the same location at the very top of RE.

Why the very top? It's because every c.e. language A accepted by a fixed machine M_A mapping reduces to A_{TM} via the "super-simple" mapping

$$f(x) = \langle M_A, x \rangle,$$

which is correct because $x \in A \iff M_A$ accepts $x \iff \langle M_A, x \rangle \in A_{TM}$. This state of affairs is summarized by the following key definition.

Definition: A language B is **complete** for a class C of languages (such as C = RE) **under** a reducibility relation $\leq_r$ (such as computable mapping reductibility $\leq_m$) if:

1. $B \in$ C, and
2. for all languages $A \in$ C, $A \leq_r B$.

If only the latter holds, we say that B is **hard** for C under the reducibility. In the case where C is RE, we also say that B is RE-**complete** (or RE-**hard** if we don't have $B \in$ RE), and the synonyms **r.e.-complete**, **c.e.-complete** or just "complete" come into play (but not "recognizably complete").

Thus A_{TM}, HP_{TM}, and K_{TM} are all complete for RE. Moreover, D_{TM} is complete for co-RE. We will also see that ALL_{TM} is not in RE, so it is RE-hard without being RE-complete. The class REC should actually "collapse to a single point" under $\leq_m$ because of the following trivial theorem:

Theorem 3: All decidable languages are $\equiv_m$-equivalent (technically except for $\emptyset$ and Σ^*).

Proof: Suppose A and B are decidable, and B is neither is $\emptyset$ or Σ^*. Then there is a "yes string" $y_0 \in B$ and a "no string" $z_0 \notin B$. By A being decidable, we can take a total TM M_A such that $L(M_A) = A$. Then define the mapping f as follows, for all $x \in \Sigma^*$:

$$f(x) = \begin{cases} y_0 & \text{if } M_A \text{ accepts } x \\ z_0 & \text{if } M_A \text{ rejects } x \end{cases}.$$

Because M_A is total, we can compute $f(x)$ in all cases, and clearly $x \in A \iff f(x) \in B$ by the choice of the two strings. Since the exception of $\emptyset$ and Σ^* technically reducing only *from* themselves is often ignored, we can say all decidable sets are trivially complete for REC. $\boxtimes$

But under simpler reductions than $\leq_m$, such as **polynomial-time mapping reducibility** $\leq_m^p$, the equivalence no longer holds globally---e.g., if M_A does not run in polynomial time. The classes REC,

RE, and co-RE all "keep their shape" under $\leq_m^p$ (and in fact, basically every reduction seen in this course except ones like f_3 needing NFA-to-DFA will be computable in quadratic time at worst). Indeed, A_{TM}, HP_{TM}, K_{TM}, etc. are all complete under $\leq_m^p$, though next wee what we will care about is completeness for the class NP under $\leq_m^p$. The one place where the diagram misleads is that REC does *not* have complete sets under $\leq_m^p$, which we try to signify by putting a little round arc under its "peaked top."

Three Design Patterns For Reductions

The motivation is similar to that in general code: the ideas of reductions are often reusable.

I. "Wait For It"

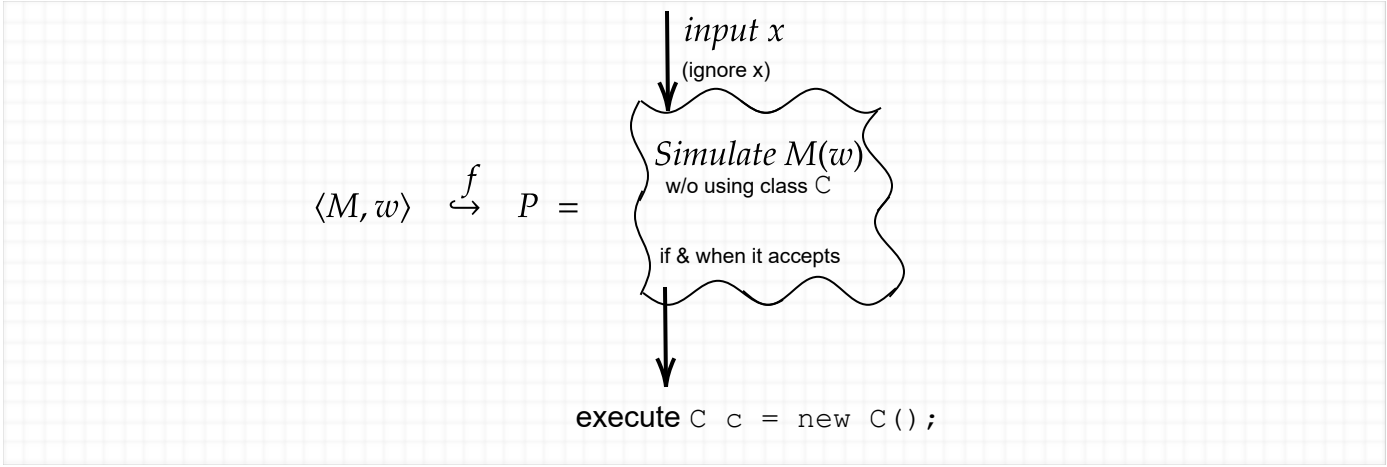
Long ago, certainly before *Hamilton*, I used to call the first one "Waiting For Godot" after the Samuel Beckett play in which (spoiler alert---wait, giving a spoiler alert for that play is an ultimate existential absurdity) ... When we first had the *Turing Kit* and Java was new and intimations of the "Internet of Things" started to buzz, I called this the generic reduction to the "Brew Coffee" problem: if you switch on your Java-enabled coffee maker M', will it brew coffee? You see, M' might ask Alexa to invoke the *Turing Kit* on a given $\langle M, w \rangle$, and brew your coffee only if and when M accepts w. Now with Turing on the UK £50 note, I've considered joking about the "ATM Problem": if you put your card in and try to withdraw £50, will it give you a Turing note, or will it do a background check that never halts? But let's use a problem that is actually highly relevant and attempted in practice when trying to cut down "code bloat" by removing unused classes from object-oriented code.

USEFULCLASS
Instance: A Java program P and a class `C` defined in the code of P.
Question: Is there an input x such that $P(x)$ creates an object of class `C`?

We mapping-reduce A_{TM} to the language of this decision problem. We need to compute $f(\langle M, w \rangle) = P$ such that:

- M accepts $w \implies$ for some x, $P(x)$ executes an instruction like `C c = new C();`
- M does not accept $w \implies$ for all x, $P(x)$ never executes any statement involving `C`.

I like to picture f as dropping M and w into a flowchart for P:



A key fine point in the correctness logic is that the class `C` does not appear anywhere else in the code of P. The main body of P can be entirely a call to the *Turing Kit* program with M and w pre-packaged. This body does not use any classes besides those in the *Turing Kit* itself. Even if $M(w) \uparrow$, whereupon P never halts either, it remains true that the class `C` is never used---so that removing it would not change the behavior of P, not on any input x.

Building the program P is straightforward given any M and w: just fix M and w to be the arguments in the call to the *Turing Kit*'s main simulation routine and append the statement shown in the diagram after the place in the *Turing Kit*'s own java code where it shows the `String accepted` dialog box. Thus the code mapping f itself is computable, indeed, easily linear-time computable.

The conclusion is that the problem of detecting (never-)used classes is undecidable. It may seem that programs $P = P_{M.w}$ are irrelevant ones by which to demonstrate this because they are so artificial and stupidly impractical. However:

1. the reduction to these programs shows that there is "no silver bullet" for deciding the useful-code problem in *all* cases; and
2. the programs $P_{M.w}$ are "tip of an iceberg" of cases that have so solidly resisted solution that most people don't try---exceptions such as the Microsoft Terminator Project are rare.

The moral is that most of these problems, by dint of being undecidable in their general theoretical formulations, are practically hard to solve. The practical problem of eliminating code bloat by removing never-used classes is one of them. Without strict version control, whether blocks of code have become truly "orphaned" and no longer executed can become hard to tell.

For a side note, the "type" of the target problem is "Just a progarm P", not "a program and an input string" as with A_{TM} itself. We did not map $\langle M, w \rangle$ to $\langle P, w \rangle$; w is not the input to P. Instead, x is quantified existentially in the statement of the problem. This makes sense: the code is useful so long as *some* input uses it. The language of the problem combines two existential conditions:

- there exists an x such that when P is run on x, ...
- ...there exists a step at which P creates an object of the class $\texttt{C}$.

A language defined by existential quantifiers in this way, down to "bedrock" predicates like creating a class object that are *decidable*, is generally ***c.e.*** The kind of algorithmic technique used to show this is commonly called "dovetailing." I like to picture dovetailing as occurring inside an enclosing arbitrary time-allowance loop. In this case, noting that we are trying to analyze P:

input $\langle P, \texttt{C} \rangle$
for $t = 1, 2, 3, 4, \ldots$:
  for each input x up to t (or you can say: of length up to t):
    run $P(x)$ for t steps. If $P(x)$ builds an object of class $\texttt{C}$ during those steps, **accept** $\langle P, \texttt{C} \rangle$.

This loop is a program R such that $L(R) = \{\langle P, \texttt{C} \rangle : (\exists x)[P(x) \textit{ builds an object of class } \texttt{C}]\}$, which is the language of the USEFULCLASS problem. So this language is c.e. but undecidable.

II The "All-or-Nothing Switch"

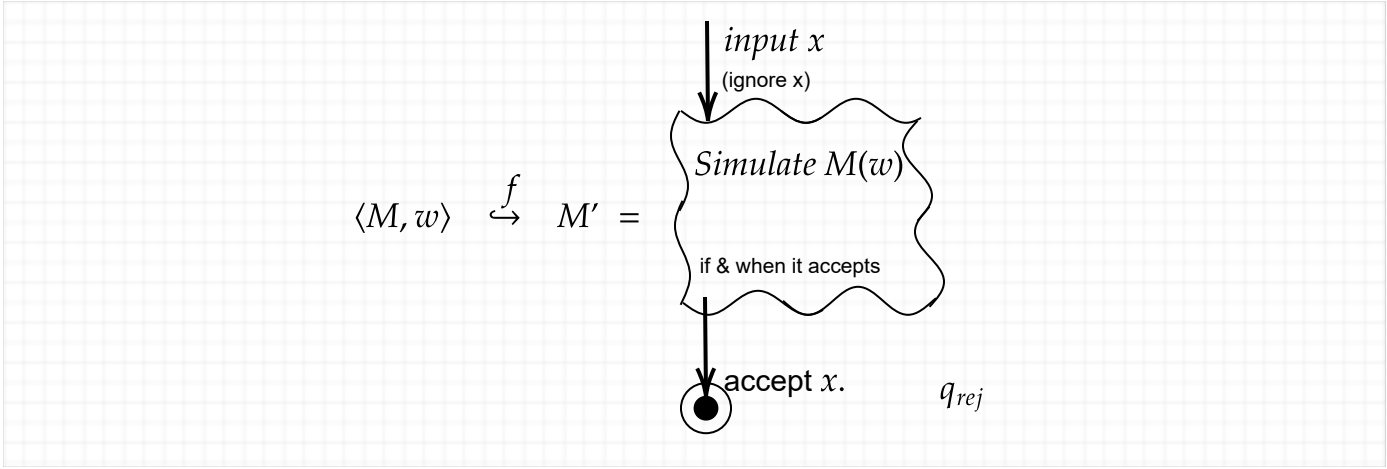
This actually builds on the "wait-for-it" kind of reductions. Note that HP_{TM} had an instance type that specified both "an M and an input x" but UsefulClass had the instance type "just a program P" where the x part was *quantified* as "Does there exist an x such that $P(x)$...?" When there is flexibility in how the "x" part is treated, we can often hit a whole bunch of problems with a reduction at once. Let's do:

NE_{TM}
Instance: A TM M.
Question: Is $L(M) \neq \varnothing$?

ALL_{TM}
Instance: A TM M.
Question: Is $L(M) = \Sigma^*$?

Eps_{TM}
Instance: A TM M.
Question: Does M accept ϵ?

In the first problem, it might seem more natural to phrase the question as "is $L(M) = \varnothing$?" but that would make the *language* of the problem become $\{\langle M \rangle : L(M) = \varnothing\}$, which is called E_{TM}. The reason we need to use $NE_{TM} = \{\langle M \rangle : L(M) \neq \varnothing\}$ is that when doing mapping reductions, we need to make "yes" cases of the *source* problem line up with "yes" answers to the *target* problem. We will see that usually it is impossible to do it the other way. Here is the reduction:



Here M' is a Turing machine, but we could get it by using the same call to the *Turing Kit* and then converting the resulting Java code to a Turing machine as proved in the Friday 10/2 lecture. Or we can just build M' by having M' (which depends on M and w) first write the fixed string $\langle M, w \rangle$ on its tape next to x (or even in place of x in this case) and then go to the start state of a universal TM U which is made to run on $\langle M, w \rangle$. Either way, f is computable---since U is fixed and the initial "write $\langle M, w \rangle$" step takes time proportional to the length of $\langle M, w \rangle$ to code, the latter more clearly makes f linear-time computable. So this **C**onstruction is **C**omputable.

Here is the one-shot **C**orrectness analysis for all three target problems:

M *accepts* w $\implies$ the "fuzzy box" main body of M' always exits, regardless of the input x;
$\implies$ for all inputs x, M' accepts x;
$\implies L(M') = \Sigma^*$, which also implies that $L(M') \neq \emptyset$ and M accepts ϵ.

Thus,
$\langle M, w \rangle \in A_{TM} \implies f(\langle M, w \rangle) = \langle M' \rangle$ is in all of ALL_{TM}, NE_{TM}, and Eps_{TM}. Whereas,

M *doesn't accept* w $\implies$ the main body of M' rejects or never finishes; either way, it never accepts;
$\implies$ for all inputs x, M' does not accept x;
$\implies L(M') = \emptyset$, which also implies that $L(M') \neq \Sigma^*$ and $\epsilon \notin L(M')$.

Thus,
$\langle M, w \rangle \notin A_{TM} \implies f(\langle M, w \rangle) \notin E_{TM}$, $f(\langle M, w \rangle) \notin ALL_{TM}$, and $f(\langle M, w \rangle) \notin Eps_{TM}$.

So we have simultaneously shown $A_{TM} \leq_m NE_{TM}$, $A_{TM} \leq_m ALL_{TM}$, and $A_{TM} \leq_m Eps_{TM}$. Thus
all three of these problems and their languages are undecidable.

In passing, here's a self-study question: How would you go about showing $A_{TM} \leq_m K_{TM}$? Showing $K_{TM} \leq_m A_{TM}$ was easy, but now we have to package an arbitrary pair $\langle M, w \rangle$ into a single machine M' that accepts its own code if and only if M accepts w. If you think about this task intensionally, it may seem daunting: how can we vary the code of M' for all the various w strings so

that M' does or does not accept its wn code depending on whether w gets accepted by M. How on earth can we pack two things into one? But if you think extensionally in terms of the correctness logic of a reduction, the answer might "jump off the page" at you...

By showing $A_{TM} \leq_m NE_{TM}$, we have not only shown that the NE_{TM} language is undecidable, we have shown it is not co-c.e. But since the A_{TM} language is c.e., NE_{TM} could be c.e.---and indeed it is, by dovetailing: Given any TM M', for $t = 1, 2, 3, \ldots$: try M' on all inputs $x < t$ for up to t steps. If M' is found to accept any of them within t steps, accept $\langle M' \rangle$, else continue. That the language (of) Eps_{TM} is c.e. is simpler to see: given M', just run $M'(\epsilon)$ and accept $\langle M' \rangle$ if and when M' accepts ϵ. But how about the language of ALL_{TM}? Hmmm....