For each of the following languages is the language decidable?

- $A_{\text{DFA}} = \{ \langle B, w \rangle | B \text{ is a DFA that accepts } w \}$
- $A_{\text{NFA}} = \{ \langle B, w \rangle | B \text{ is a NFA that accepts } w \}$
CS/ECE-374: Lecture 24 - Decidability

**Lecturer:** Nickvash Kani  
**Chat moderator:** Samir Khan  
**April 20, 2021**  

University of Illinois at Urbana-Champaign
For each of the following languages is the language decidable?

\( A_{\text{DFA}} = \{ \langle B, w \rangle | B \text{ is a DFA that accepts } w \} \)

\( A_{\text{NFA}} = \{ \langle B, w \rangle | B \text{ is a NFA that accepts } w \} \)

Yes, b/c simulate a DFA using linear time algorithms we've been using

Reduces to \( A_{\text{DFA}} \)
Turing machines...

\[ \langle \text{TM} \rangle \Rightarrow \text{string that encodes \text{TM}} \]

\[ \text{L}(\text{TM}) \Rightarrow \text{language that consists of strings \text{TM} accepts} \]

\[ \text{TM}: \quad \text{return accept}; \]

\[ \text{L}(\text{GTM}) = \varepsilon^* \]
Reminder: Undecidability

Definition
Language \( L \subseteq \Sigma^* \) is \textbf{undecidable} if no program \( P \), given \( w \in \Sigma^* \) as input, can \textbf{always stop} and output whether \( w \in L \) or \( w \notin L \).

(Usually defined using \textbf{TM} not programs. But equivalent.)

\[
\text{Decidable} \ L \Rightarrow \text{program exists which always stops and outputs accept/reject}
\]
Definition
Language $L \subseteq \Sigma^*$ is undecidable if no program $P$, given $w \in \Sigma^*$ as input, can always stop and output whether $w \in L$ or $w \notin L$.

(Usually defined using $\text{TM}$ not programs. But equivalent.)
Reminder: Undecidability

**Definition**
Language $L \subseteq \Sigma^*$ is undecidable if no program $P$, given $w \in \Sigma^*$ as input, can always stop and output whether $w \in L$ or $w \notin L$.

(Usually defined using TM not programs. But equivalent.)
Reminder: The following language is undecidable

Decide if given a program $M$, and an input $w$, does $M$ accepts $w$. Formally, the corresponding language is

$$A_{TM} = \left\{ \langle M, w \rangle \mid M \text{ is a } TM \text{ and } M \text{ accepts } w \right\}.$$
Reminder: The following language is undecidable

Decide if given a program $M$, and an input $w$, does $M$ accepts $w$. Formally, the corresponding language is

$$A_{TM} = \left\{ \langle M, w \rangle \mid M \text{ is a } TM \text{ and } M \text{ accepts } w \right\}.$$  

**Definition**

A *decider* for a language $L$, is a program (or a $TM$) that always stops, and outputs for any input string $w \in \Sigma^*$ whether or not $w \in L$.

A language that has a decider is *decidable*. 
Reminder: The following language is undecidable

Decide if given a program $M$, and an input $w$, does $M$ accepts $w$. Formally, the corresponding language is

$$A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}.$$  

**Definition**

A *decider* for a language $L$, is a program (or a TM) that always stops, and outputs for any input string $w \in \Sigma^*$ whether or not $w \in L$.

A language that has a decider is *decidable*.

Turing proved the following:

**Theorem**

$A_{TM}$ is undecidable.
The halting problem
$A_{TM}$ is not TM decidable!

$A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a } TM \text{ and } M \text{ accepts } w \}.$

**Theorem (The halting theorem.)**

$A_{TM}$ is not Turing decidable.
$A_{TM}$ is not TM decidable!

$$A_{TM} = \left\{ \langle M, w \rangle \mid M \text{ is a } TM \text{ and } M \text{ accepts } w \right\}.$$ 

Theorem (The halting theorem.)

$A_{TM}$ is not Turing decidable.

Proof: Assume $A_{TM}$ is TM decidable...
A_{TM} is not TM decidable!

A_{TM} = \{ \left\langle M, w \right\rangle \mid M \text{ is a } TM \text{ and } M \text{ accepts } w \}.

Theorem (The halting theorem.)
A_{TM} is not Turing decidable.

Proof: Assume A_{TM} is TM decidable...

Halt: TM deciding A_{TM}. Halt always halts, and works as follows:

\[
Halt(\left\langle M, w \right\rangle) = \begin{cases} 
\text{accept} & M \text{ accepts } w \\
\text{reject} & M \text{ does not accept } w.
\end{cases}
\]
We build the following new function:

\[
\text{Flipper}(\langle M \rangle) \\
\text{res } \leftarrow \text{Halt}(\langle M, M \rangle) \\
\text{if res is accept then reject} \\
\text{else accept}
\]
We build the following new function:

\[ \text{Flipper}(\langle M \rangle) = \begin{cases} 
\text{replay} & \text{M accepts } \langle M \rangle \\
\text{accept} & \text{M does not accept } \langle M \rangle.
\end{cases} \]

\text{Flipper} \text{ always stops:}

\[ \text{Flipper}(\langle M \rangle) = \begin{cases} 
\text{replay} & \text{if res is accept then reject} \\
\text{accept} & \text{else accept}
\end{cases} \]

This is decidable.
Halting theorem proof continued 2

\[
\text{Flipper}(\langle M \rangle) = \begin{cases} 
  \text{reject} & M \text{ accepts } \langle M \rangle \\
  \text{accept} & M \text{ does not accept } \langle M \rangle 
\end{cases}
\]

Flipper is a TM (duh!), and as such it has an encoding \langle Flipper \rangle. Run Flipper on itself:

\[
\text{HALT} (\text{Flipper}, \langle \text{Flipper} \rangle)
\]

\[
\text{Flipper}(\langle \text{Flipper} \rangle) = \begin{cases} 
  \text{reject} & \text{Flipper accepts } \langle \text{Flipper} \rangle \\
  \text{accept} & \text{Flipper does not accept } \langle \text{Flipper} \rangle 
\end{cases}
\]
Halting theorem proof continued 2

\[
\text{Flipper}(\langle M \rangle) = \begin{cases} 
\text{reject} & \text{if } M \text{ accepts } \langle M \rangle \\
\text{accept} & \text{if } M \text{ does not accept } \langle M \rangle.
\end{cases}
\]

**Flipper** is a TM (duh!), and as such it has an encoding \langle Flipper \rangle. Run **Flipper** on itself:

\[
\text{Flipper}(\langle \text{Flipper} \rangle) = \begin{cases} 
\text{reject} & \text{if } \text{Flipper} \text{ accepts } \langle \text{Flipper} \rangle \\
\text{accept} & \text{if } \text{Flipper} \text{ does not accept } \langle \text{Flipper} \rangle.
\end{cases}
\]

This is absurd. Ridiculous even!
Halting theorem proof continued 2

\[ \text{Flipper}(\langle M \rangle) = \begin{cases} 
\text{reject} & M \text{ accepts } \langle M \rangle \\
\text{accept} & M \text{ does not accept } \langle M \rangle.
\end{cases} \]

\textbf{Flipper} is a TM (duh!), and as such it has an encoding \( \langle \text{Flipper} \rangle \).
Run \textbf{Flipper} on itself:

\[ \text{Flipper}(\langle \text{Flipper} \rangle) = \begin{cases} 
\text{reject} & \text{Flipper accepts } \langle \text{Flipper} \rangle \\
\text{accept} & \text{Flipper does not accept } \langle \text{Flipper} \rangle.
\end{cases} \]

This is absurd. Ridiculous even!

Assumption that \textbf{Halt} exists is false. \( \Rightarrow \) \( A_{TM} \) is not TM decidable.

\textit{Seed Idea of Decidability:} \( A_{TM} \) is undecidable.
Reductions
**Meta definition:** Problem $X$ reduces to problem $Y$, if given a solution to $Y$, then it implies a solution for $X$. Namely, we can solve $Y$ then we can solve $X$. We will done this by $X \rightarrow Y$. 

$x \leq y$ 

undecidable

accept

reject
Meta definition: Problem $X$ reduces to problem $B$, if given a solution to $B$, then it implies a solution for $X$. Namely, we can solve $Y$ then we can solve $X$. We will done this by $X \implies Y$.

Definition

oracle ORAC for language $L$ is a function that receives as a word $w$, returns $\text{TRUE} \iff w \in L$. 

Trying to prove $Y$ is undecidable.
**Meta definition:** Problem $X$ reduces to problem $B$, if given a solution to $B$, then it implies a solution for $X$. Namely, we can solve $Y$ then we can solve $X$. We will done this by $X \implies Y$.

**Definition**

*oracle* ORAC for language $L$ is a function that receives as a word $w$, returns $\text{TRUE} \iff w \in L$.

**Lemma**

*A language $X$ reduces to a language $Y$, if one can construct a TM decider for $X$ using a given oracle ORAC$_Y$ for $Y$. We will denote this fact by $X \implies Y$.***
Reduction proof technique

- **Y**: Problem/language for which we want to prove undecidable.
Reduction proof technique

- **Y**: Problem/language for which we want to prove undecidable.
- Proof via reduction. Result in a proof by contradiction.
Reduction proof technique

- **Y**: Problem/language for which we want to prove undecidable.
- Proof via reduction. Result in a proof by contradiction.
- **L**: language of **Y**.
Reduction proof technique

- **Y**: Problem/language for which we want to prove undecidable.
- Proof via reduction. Result in a proof by contradiction.
- **L**: language of **Y**.
- Assume **L** is decided by **TM** **M**.
Reduction proof technique

- **Y**: Problem/language for which we want to prove undecidable.
- Proof via reduction. Result in a proof by contradiction.
- **L**: language of **Y**.
- Assume **L** is decided by **TM M**.
- Create a decider for known undecidable problem **X** using **M**.
Reduction proof technique

- **Y**: Problem/language for which we want to prove undecidable.
- Proof via reduction. Result in a proof by contradiction.
- **L**: language of **Y**.
- Assume **L** is decided by **TM M**.
- Create a decider for known undecidable problem **X** using **M**.
- Result in decider for **X** (i.e., \( A_{TM} \)).
Reduction proof technique

- **Y**: Problem/language for which we want to prove undecidable.
- Proof via reduction. Result in a proof by contradiction.
- **L**: language of **Y**.
- Assume **L** is decided by **TM M**.
- Create a decider for known undecidable problem **X** using **M**.
- Result in decider for **X** (i.e., **A_{TM}**).
- Contradiction **X** is not decidable.
Reduction proof technique

- **Y**: Problem/language for which we want to prove undecidable.
- Proof via reduction. Result in a proof by contradiction.
- **L**: language of **Y**.
- Assume **L** is decided by **TM M**.
- Create a decider for known undecidable problem **X** using **M**.
- Result in decider for **X** (i.e., **A_{TM}**).
- Contradiction **X** is not decidable.
- Thus, **L** must be not decidable.
Lemma
Let $X$ and $Y$ be two languages, and assume that $X \implies Y$. If $Y$ is decidable then $X$ is decidable.

Proof.
Let $T$ be a decider for $Y$ (i.e., a program or a TM). Since $X$ reduces to $Y$, it follows that there is a procedure $T_{X|Y}$ (i.e., decider) for $X$ that uses an oracle for $Y$ as a subroutine. We replace the calls to this oracle in $T_{X|Y}$ by calls to $T$. The resulting program $T_X$ is a decider and its language is $X$. Thus $X$ is decidable (or more formally $TM$ decidable).
The contraposition...

Lemma
Let X and Y be two languages, and assume that $X \implies Y$. If X is undecidable then Y is undecidable.
Halting
The halting problem

Language of all pairs $\langle M, w \rangle$ such that $M$ halts on $w$:

$$A_{\text{Halt}} = \left\{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ stops on } w \right\}.$$

Similar to language already known to be undecidable:

$$A_{\text{TM}} = \left\{ \langle M, w \rangle \mid M \text{ is a } \text{TM} \text{ and } M \text{ accepts } w \right\}.$$

$$A_{\text{TM}} \supseteq A_{\text{Halt}}.$$
On way to proving that Halting is undecidable...

**Lemma**
The language $A_{TM}$ reduces to $A_{\text{Halt}}$. Namely, given an oracle for $A_{\text{Halt}}$ one can build a decider (that uses this oracle) for $A_{TM}$.

$\text{ORAC}_{\text{Halt}} = \begin{cases} 
\text{accept} & \text{if } M \text{ halts on } w \\
\text{reject} & \text{if } M \text{ does not halt on } w \\
\langle M, w \rangle \notin A_{TM} & \text{resimulate } M \text{ on } w \\
\text{informs } & \text{if } \langle M, w \rangle \in A_{TM} 
\end{cases}$
On way to proving that Halting is undecidable...

Proof. Let $\text{ORAC}_{\text{Halt}}$ be the given oracle for $A_{\text{Halt}}$. We build the following decider for $A_{\text{TM}}$.

```
AnotherDecider-\text{A}_{\text{TM}}(\langle M, w \rangle)
res \leftarrow \text{ORAC}_{\text{Halt}}(\langle M, w \rangle)
// if $M$ does not halt on $w$ then reject.
if res = reject then
    halt and reject.
// $M$ halts on $w$ since res = accept.
// Simulating $M$ on $w$ terminates in finite time.
res_2 \leftarrow \text{Simulate} ~ M ~ \text{on} ~ w.
return res_2.
```

This procedure always return and as such its a decider for $A_{\text{TM}}$. \qed
The Halting problem is not decidable

**Theorem**

The language $A_{\text{Halt}}$ is not decidable.

**Proof.**

Assume, for the sake of contradiction, that $A_{\text{Halt}}$ is decidable. As such, there is a $TM$, denoted by $TM_{\text{Halt}}$, that is a decider for $A_{\text{Halt}}$. We can use $TM_{\text{Halt}}$ as an implementation of an oracle for $A_{\text{Halt}}$, which would imply that one can build a decider for $A_{TM}$. However, $A_{TM}$ is undecidable. A contradiction. It must be that $A_{\text{Halt}}$ is undecidable. □
The same proof by figure...

... if $A_{\text{Halt}}$ is decidable, then $A_{\text{TM}}$ is decidable, which is impossible.
Emptiness
The language of empty languages

- $E_{\text{TM}} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset \}$.  
- $TM_{ETM}$: Assume we are given this decider for $E_{\text{TM}}$. Assume $E_{\text{TM}}$ is decidable.
- Need to use $TM_{ETM}$ to build a decider for $A_{\text{TM}}$.
- Decider for $A_{\text{TM}}$ is given $M$ and $w$ and must decide whether $M$ accepts $w$.
- Restructure question to be about Turing machine having an empty language.
- Somehow make the second input ($w$) disappear.
The language of empty languages

- \( E_\text{TM} = \{ \langle M \rangle \mid M \text{ is a } \text{TM and } L(M) = \emptyset \} \).
- \( TM_{ETM} \): Assume we are given this decider for \( E_\text{TM} \).
- Need to use \( TM_{ETM} \) to build a decider for \( A_\text{TM} \).
- Decider for \( A_\text{TM} \) is given \( M \) and \( w \) and must decide whether \( M \) accepts \( w \).
- Restructure question to be about Turing machine having an empty language.
- Somehow make the second input (\( w \)) disappear.
- Idea: hard-code \( w \) into \( M \), creating a \( \text{TM} M'_w \) which runs \( M \) on the fixed string \( w \).
- \( \text{TM} M'_w \):
  1. Input = \( x \) (which will be ignored)
  2. Simulate \( M \) on \( w \).
  3. If the simulation accepts, accept. If the simulation rejects, reject.
Embedding strings...

- Given program \( \langle M \rangle \) and input \( w \)...
- ...can output a program \( \langle M_w \rangle \).
- The program \( M_w \) simulates \( M \) on \( w \). And accepts/rejects accordingly.
- **EmbedString**\((\langle M, w \rangle)\) input two strings \( \langle M \rangle \) and \( w \), and output a string encoding \( \langle \text{TM} \rangle \) \( \langle M_w \rangle \).
Embedding strings...

• Given program $\langle M \rangle$ and input $w$...
• ...can output a program $\langle M_w \rangle$.
• The program $M_w$ simulates $M$ on $w$. And accepts/rejects accordingly.
• $\text{EmbedString}(\langle M, w \rangle)$ input two strings $\langle M \rangle$ and $w$, and output a string encoding (TM) $\langle M_w \rangle$.
• What is $L(M_w)$?
Embedding strings...

- Given program $\langle M \rangle$ and input $w$...
- ...can output a program $\langle M_w \rangle$.
- The program $M_w$ simulates $M$ on $w$. And accepts/rejects accordingly.
- **EmbedString**($\langle M, w \rangle$) input two strings $\langle M \rangle$ and $w$, and output a string encoding (TM) $\langle M_w \rangle$.
- What is $L(M_w)$?
- Since $M_w$ ignores input $x$.. language $M_w$ is either $\Sigma^*$ or $\emptyset$. It is $\Sigma^*$ if $M$ accepts $w$, and it is $\emptyset$ if $M$ does not accept $w.$
Emptiness is undecidable

Theorem
The language $E_{TM}$ is undecidable.

- Assume (for contradiction), that $E_{TM}$ is decidable.
- $TM_{ETM}$ be its decider.
- Build decider $\text{AnotherDecider-}A_{TM}$ for $A_{TM}$.

\[
\text{AnotherDecider-}A_{TM}(\langle M, w \rangle) \leftarrow \text{EmbedString}(\langle M, w \rangle)
\]
\[
r \leftarrow TM_{ETM}(\langle M_w \rangle).
\]
if $r =$ accept then

\[
\text{return reject}
\]

// $TM_{ETM}(\langle M_w \rangle)$ rejected its input

return accept
Emptiness is undecidable...

Consider the possible behavior of $\text{AnotherDecider-} \ A_{TM}$ on the input $\langle M, w \rangle$.

- If $TM_{ETM}$ accepts $\langle M_w \rangle$, then $L(M_w)$ is empty. This implies that $M$ does not accept $w$. As such, $\text{AnotherDecider-} \ A_{TM}$ rejects its input $\langle M, w \rangle$.
- If $TM_{ETM}$ accepts $\langle M_w \rangle$, then $L(M_w)$ is not empty. This implies that $M$ accepts $w$. So $\text{AnotherDecider-} \ A_{TM}$ accepts $\langle M, w \rangle$. 
Emptiness is undecidable...

Consider the possible behavior of $\text{AnotherDecider-}A_{TM}$ on the input $\langle M, w \rangle$.

- If $TM_{ETM}$ accepts $\langle M_w \rangle$, then $L(M_w)$ is empty. This implies that $M$ does not accept $w$. As such, $\text{AnotherDecider-}A_{TM}$ rejects its input $\langle M, w \rangle$.
- If $TM_{ETM}$ accepts $\langle M_w \rangle$, then $L(M_w)$ is not empty. This implies that $M$ accepts $w$. So $\text{AnotherDecider-}A_{TM}$ accepts $\langle M, w \rangle$.

$\implies \text{AnotherDecider-}A_{TM}$ is decider for $A_{TM}$.

But $A_{TM}$ is undecidable...
Consider the possible behavior of $\text{AnotherDecider-}A_{TM}$ on the input $\langle M, w \rangle$.

- If $TM_{ETM}$ accepts $\langle M_w \rangle$, then $L(M_w)$ is empty. This implies that $M$ does not accept $w$. As such, $\text{AnotherDecider-}A_{TM}$ rejects its input $\langle M, w \rangle$.

- If $TM_{ETM}$ accepts $\langle M_w \rangle$, then $L(M_w)$ is not empty. This implies that $M$ accepts $w$. So $\text{AnotherDecider-}A_{TM}$ accepts $\langle M, w \rangle$.

$\implies \text{AnotherDecider-}A_{TM}$ is decider for $A_{TM}$.

But $A_{TM}$ is undecidable...

...must be assumption that $E_{TM}$ is decidable is false.
Emptiness is undecidable via diagram

AnotherDecider-\(A_{TM}\) never actually runs the code for \(M_w\). It hands the code to a function \(TM_{ETM}\) which analyzes what the code would do if run it. So it does not matter that \(M_w\) might go into an infinite loop.
Equality
Equality is undecidable

\[ EQ_{TM} = \{ \langle M, N \rangle \mid M \text{ and } N \text{ are TM's and } L(M) = L(N) \} . \]

Lemma

The language \( EQ_{TM} \) is undecidable.

Let's use \( TM_{\text{Empty}} \) is undecidable

\( TM_{\text{Empty}}(\langle M \rangle) \):
- accept if \( L(M) = \emptyset \)
- reject otherwise

\( ORAC_{EQ}(\langle M, N \rangle) \):
- accept if \( L(M) = L(N) \)
- reject otherwise
Proof.
Suppose that we had a decider $\text{DeciderEqual}$ for $EQ_{TM}$. Then we can build a decider for $E_{TM}$ as follows:

**TM $R$:**
1. Input = $\langle M \rangle$
2. Include the (constant) code for a $TM$ $T$ that rejects all its input. We denote the string encoding $T$ by $\langle T \rangle$.
3. Run $\text{DeciderEqual}$ on $\langle M, T \rangle$.
4. If $\text{DeciderEqual}$ accepts, then accept.
5. If $\text{DeciderEqual}$ rejects, then reject.
Regularity
Many undecidable languages

- Almost any property defining a TM language induces a language which is undecidable.
- proofs all have the same basic pattern.
- Regularity language:
  \[ \text{Regular}_{TM} = \left\{ \langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is regular} \right\}. \]
- \textbf{DeciderRegL}: Assume TM decider for Regular_{TM}.
- Reduction from halting requires to turn problem about deciding whether a TM \( M \) accepts \( w \) (i.e., is \( w \in A_{TM} \)) into a problem about whether some TM accepts a regular set of strings.
Proof continued...

- Given $M$ and $w$, consider the following TM $M'_w$:
  
  $M'_w$:
  
  (i) Input = $x$
  (ii) If $x$ has the form $a^n b^n$, halt and accept.
  (iii) Otherwise, simulate $M$ on $w$.
  (iv) If the simulation accepts, then accept.
  (v) If the simulation rejects, then reject.

- not executing $M'_w$!

- feed string $\langle M'_w \rangle$ into DeciderRegL

- EmbedRegularString: program with input $\langle M \rangle$ and $w$, and outputs $\langle M'_w \rangle$, encoding the program $M'_w$.

- If $M$ accepts $w$, then any $x$ accepted by $M'_w$: $L(M'_w) = \Sigma^*$. 
- If $M$ does not accept $w$, then $L(M'_w) = \{a^n b^n \mid n \geq 0\}$. 
Proof continued...

- \(a^n b^n\) is not regular...
- Use **DeciderRegL** on \(M'_w\) to distinguish these two cases.
- Note - cooked \(M'_w\) to the decider at hand.
- A decider for \(A_{TM}\) as follows.

```
AnotherDecider-A_{TM}(\langle M, w\rangle)

\langle M'_w \rangle \leftarrow \text{EmbedRegularString}(\langle M, w\rangle)

r \leftarrow \text{DeciderRegL}(\langle M'_w \rangle).

\text{return } r
```

- If **DeciderRegL** accepts \(\implies L(M'_w)\) regular (its \(\Sigma^*\))
Proof continued...

- $a^n b^n$ is not regular...
- Use DeciderRegL on $M'_w$ to distinguish these two cases.
- Note - cooked $M'_w$ to the decider at hand.
- A decider for $A_{TM}$ as follows.

$$\text{AnotherDecider-}A_{TM}(\langle M, w \rangle)$$

$$\langle M'_w \rangle \leftarrow \text{EmbedRegularString}(\langle M, w \rangle)$$

$$r \leftarrow \text{DeciderRegL}(\langle M'_w \rangle).$$

return $r$

- If DeciderRegL accepts $\implies L(M'_w)$ regular (its $\Sigma^*$) $\implies M$ accepts $w$. So AnotherDecider-$A_{TM}$ should accept $\langle M, w \rangle$. 
Proof continued...

- $a^n b^n$ is not regular...
- Use $\text{DeciderRegL}$ on $M'_w$ to distinguish these two cases.
- Note - cooked $M'_w$ to the decider at hand.
- A decider for $A_{TM}$ as follows.

```
AnotherDecider-A_{TM}(<M, w>)

    <M'_w> ← EmbedRegularString(<M, w>)
    r ← DeciderRegL(<M'_w>).

    return r
```

- If $\text{DeciderRegL}$ accepts $\implies L(M'_w)$ regular (its $\Sigma^*$) $\implies M$ accepts $w$. So $\text{AnotherDecider}-A_{TM}$ should accept $<M, w>$.
- If $\text{DeciderRegL}$ rejects $\implies L(M'_w)$ is not regular $\implies L(M'_w) = a^n b^n$
Proof continued...

- \(a^n b^n\) is not regular...
- Use \textbf{DeciderRegL} on \(M'_w\) to distinguish these two cases.
- Note - cooked \(M'_w\) to the decider at hand.
- A decider for \(A_{TM}\) as follows.

\[
\text{AnotherDecider-} A_{TM}(\langle M, w \rangle) \\
\langle M'_w \rangle \leftarrow \text{EmbedRegularString}(\langle M, w \rangle) \\
r \leftarrow \text{DeciderRegL}(\langle M'_w \rangle).
\]

\[
\text{return } r
\]

- If \textbf{DeciderRegL} accepts \(\implies L(M'_w)\) regular (its \(\Sigma^*\) \(\implies M\) accepts \(w\). So \textbf{AnotherDecider-} A_{TM} should accept \(\langle M, w \rangle\).
- If \textbf{DeciderRegL} rejects \(\implies L(M'_w)\) is not regular \(\implies L(M'_w) = a^n b^n \implies M\) does not accept \(w\) \(\implies\) \textbf{AnotherDecider-} A_{TM} should reject \(\langle M, w \rangle\).
The above proofs were somewhat repetitious... 
...they imply a more general result.

**Theorem (Rice’s Theorem.)**
Suppose that $L$ is a language of Turing machines; that is, each word in $L$ encodes a $\text{TM}$. Furthermore, assume that the following two properties hold.

(a) Membership in $L$ depends only on the Turing machine’s language, i.e. if $L(M) = L(N)$ then $\langle M \rangle \in L \iff \langle N \rangle \in L$.

(b) The set $L$ is “non-trivial,” i.e. $L \neq \emptyset$ and $L$ does not contain all Turing machines.

Then $L$ is a undecidable.