Outline for Today

- **Self-Reference Revisited**
  - Programs that compute on themselves.
- **Self-Defeating Objects**
  - Objects “too powerful” to exist.
- **The Fortune Teller**
  - Can you escape the future?
- **Why Do Programs Loop?**
  - ... and can we eliminate loops?
- **Undecidable Problems**
  - Something beyond the reach of algorithms.
Recap from Last Time
R and RE

• A language \( L \) is **recognizable** if there is a TM \( M \) with the following property:

\[
\forall w \in \Sigma^*. \ (M \text{ accepts } w \leftrightarrow w \in L).
\]

• That is, for any string \( w \):
  - If \( w \in L \), then \( M \) accepts \( w \).
  - If \( w \notin L \), then \( M \) does not accept \( w \).
    - It might reject \( w \), or it might loop on \( w \).

• This is a “weak” notion of solving a problem.

• The class **RE** consists of all the recognizable languages.
A language $L$ is **decidable** if there is a TM $M$ with the following properties:

$$\forall w \in \Sigma^*. (M \text{ accepts } w \leftrightarrow w \in L).$$

*M halts on all inputs.*

That is, for any string $w$:

- If $w \in L$, then $M$ accepts $w$.
- If $w \notin L$, then $M$ rejects $w$.

This is a “strong” notion of solving a problem.

The class $\mathsf{R}$ consists of all the decidable languages.
The Universal TM

- The *universal Turing machine*, denoted $U_{\text{TM}}$, is a TM with the following behavior: when run on a string $\langle M, w \rangle$, where $M$ is a TM and $w$ is a string, $U_{\text{TM}}$ will
  
  ... accept $\langle M, w \rangle$ if $M$ accepts $w$,
  
  ... reject $\langle M, w \rangle$ if $M$ rejects $w$, and
  
  ... loop on $\langle M, w \rangle$ if $M$ loops on $w$.

- $A_{\text{TM}}$ is the language recognized by the universal TM. This is the language

  $$ A_{\text{TM}} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \} $$
New Stuff!
Part One: Self-Defeating Objects
A *self-defeating object* is an object whose essential properties ensure it doesn’t exist.
**Question:** Why is there no largest integer?

**Answer:** Because if $n$ is the largest integer, what happens when we look at $n+1$?
Self-Defeating Objects

**Theorem:** There is no largest integer.

**Proof sketch:** Suppose for the sake of contradiction that there is a largest integer. Call that integer \( n \).

Consider the integer \( n+1 \).

Notice that \( n < n+1 \).

But then \( n \) isn’t the largest integer.

Contradiction! ■-ish
Theorem: There is no largest integer.

Proof sketch: Suppose for the sake of contradiction that there is a largest integer. Call that integer \( n \).

Consider the integer \( n+1 \).

Notice that \( n < n+1 \).

But then \( n \) isn’t the largest integer.

Contradiction! ■-ish

We’re using \( n \) to construct something that undermines \( n \), hence the term “self-defeating.”
An Important Detail
Claim: There is a largest integer.

Proof: Assume $x$ is the largest integer. Notice that $x > x - 1$. So there's no contradiction. ■-ish

Careful – we’re assuming what we’re trying to prove!

How do we know there’s no contradiction? We just checked one case.
Self-Defeating Objects

- If you can show
  \[ x \text{ exists} \rightarrow \bot \]
  then you know that \( x \) doesn’t exist. (This is a proof by contradiction.)

- If you can show
  \[ x \text{ exists} \rightarrow \top \]
  you cannot conclude that \( x \) exists. (This is not a valid proof technique.)
Part Two: The Fortune Teller
The Fortune Teller

- A fortune teller appears who claims they can see into anyone’s future.
- For a nominal fee, the fortune teller will tell you anything you want to know about the future.
• One day, a trickster arrives. The trickster thinks the fortune teller is lying and can’t really see the future.

• The trickster says the following:

   “I have a yes/no question about the future. But before I ask my question, let’s talk payment.

   If you answer yes, then I’ll pay you $137.

   If you answer no, then I’ll pay you $42.

• The fortune teller thinks for a moment, then agrees.
The Fortune Teller

• The trickster then asks this question:
  “Am I going to pay you $42?”

• The fortune teller is trapped!

• Talk to your neighbor – why?

Trickster pays $137 if the fortune teller answers “yes.”

Trickster pays $42 if the fortune teller answers “no.”
The Fortune Teller

- The payment scheme the fortune teller agreed to means
  \[ \text{Fortune Teller Says Yes} \quad \leftrightarrow \quad \text{Trickster Pays $137}. \]
- The trickster’s question to the fortune teller means
  \[ \text{Fortune Teller Says Yes} \quad \leftrightarrow \quad \text{Trickster Pays $42}. \]
- Putting this together, we get
  \[ \text{Trickster Pays $42} \quad \leftrightarrow \quad \text{Trickster Pays $137}. \]
- This is impossible!

Trickster pays $137 if the fortune teller answers “yes.”

Trickster pays $42 if the fortune teller answers “no.”
The Fortune Teller

- The fortune teller is a self-defeating object.
- The trickster’s strategy is to couple the fortune teller’s behavior to what the future holds.
  - The trickster’s behavior is chosen in advance to make the fortune teller’s answer wrong.
- Therefore, the fortune teller can’t answer all questions about all people in the future.

Trickster pays $137 if the fortune teller answers “yes.”

Trickster pays $42 if the fortune teller answers “no.”
Part Three: Why Do Programs Loop?
Thoughts on Loops

• In practice, the programs we write sometimes go into infinite loops.

• In Theoryland, Turing machines are allowed to loop. This happens if they don’t accept and don’t reject.

• **Question:** Why are infinite loops possible?

• Or rather: are infinite loops an inherent part of computation, or are they some weird sort of “accident” in how we program computers?
Thoughts on Loops

- **Theorem:** The language $A_{TM}$ is recognizable, but undecidable.
  - There’s a recognizer for $A_{TM}$ (specifically, the universal Turing machine $U_{TM}$).
  - It is impossible to build a decider for this language.
  - Stated differently, there’s a program we can write (a universal TM) that has to loop infinitely on some inputs.

- **Goal:** Prove this theorem, and explore its theoretical and philosophical implications.
A_{TM} Revisited

• As a refresher, the language \( A_{TM} \) is
\[
A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}.
\]

• The universal TM \( U_{TM} \) has the following behavior when given as input a TM \( M \) and a string \( w \):
  
  • If \( M \) accepts \( w \), then \( U_{TM} \) accepts \( \langle M, w \rangle \).
  
  • If \( M \) rejects \( w \), then \( U_{TM} \) rejects \( \langle M, w \rangle \).
  
  • If \( M \) loops on \( w \), then \( U_{TM} \) loops on \( \langle M, w \rangle \).
  
• \( U_{TM} \) is a recognizer for \( A_{TM} \), but because of that last case it’s not a decider for \( A_{TM} \).
A_{TM} Revisited

• As a refresher, the language A_{TM} is

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

• Given a TM M and a string w, a decider D for A_{TM} would need to have this behavior:

  • If M accepts w, then \( D \) accepts \( \langle M, w \rangle \).
  • If M rejects w, then \( D \) rejects \( \langle M, w \rangle \).
  • If M loops on w, then \( D \) rejects \( \langle M, w \rangle \).

• This is basically the same set of requirements as U_{TM}, except for what happens if M loops on w.

• Our goal is to prove that there is no way to build a program that meets these requirements.
A_{TM} Revisited

• We can envision a decider for A_{TM} as a function
  
  \begin{verbatim}
  bool willAccept(string fn, string input)
  \end{verbatim}

  that takes as input the source code of a function (fn) and a string representing an input to that function (input).

• It then does the following:
  • If fn(input) returns true, willAccept(fn, input) returns true.
  • If fn(input) returns false, willAccept(fn, input) returns false.
  • If fn(input) loops, then willAccept(fn, input) returns false.

• We’re going to show it’s impossible to write a function that actually does this. But for now, let’s just explore what such a decider would do.
For each of these instances, what does `willAccept(function, input)` return?
Earlier this quarter you explored sums of four squares. Now, let’s talk about sums of three cubes.

Are there integers $x$, $y$, and $z$ where...

- $x^3 + y^3 + z^3 = 10$?
- $x^3 + y^3 + z^3 = 11$?
- $x^3 + y^3 + z^3 = 12$?
- $x^3 + y^3 + z^3 = 13$?

Deciding $A_{TM}$
Deciding $A_{TM}$

- Surprising fact: until 2019, no one knew whether there were integers $x$, $y$, and $z$ where
  \[ x^3 + y^3 + z^3 = 33. \]

- A heavily optimized computer search found this answer:
  \[
  x = 8,866,128,975,287,528 \\
  y = -8,778,405,442,862,239 \\
  z = -2,736,111,468,807,040
  \]

- As of August 2022, no one knows whether there are integers $x$, $y$, and $z$ where
  \[ x^3 + y^3 + z^3 = 114. \]
Deciding $A_{TM}$

- Consider the language

$$L = \{ a^n \mid \exists x \in \mathbb{Z}. \exists y \in \mathbb{Z}. \exists z \in \mathbb{Z}. x^3 + y^3 + z^3 = n \}$$

- Here’s code for a recognizer to see whether such a triple exists:

```cpp
bool hasTriple(int n) {
    for (int max = 0; ; max++)
        for (int x = -max; x <= max; x++)
            for (int y = -max; y <= max; y++)
                for (int z = -max; z <= max; z++)
                    if (x*x*x + y*y*y + z*z*z == n)
                        return true;
}
```

- Imagine calling `willAccept(/* hasTriple code */ , 114)`.
  - If such a triple exists, `willAccept` returns true.
  - If no such triple exists, `willAccept` returns false.

**Key Intuition:** However `willAccept` is implemented, it has to be clever enough to resolve open problems in mathematics!
Why is $A_{TM}$ Hard?

• **Intuition:** A decider for $A_{TM}$ would be able to...
  
  • ... determine whether the hailstone sequence terminates for any input. (Write a recognizer that runs the hailstone sequence, then feed it into the decider for $A_{TM}$.)
  
  • ... see if any number is the sum of three cubes. (Write a recognizer that tries all infinitely many triples of integers, then feed it into the decider for $A_{TM}$.)
  
  • ... and much, much more.

• In other words, this seemingly simple problem of “is this program going to terminate?” accidentally scoops up a bunch of other seemingly harder problems.
Time-Out for Announcements!
Office Hours Calendar

• We’ve shifted around the office hours calendar to add some more coverage earlier in the week. Please take a look on the course website!

• As usual, we’ll continue to be available on the EdStem forum as you finish out PS7 and prepare for the final exam.
Please evaluate this course in Axess.
Your comments really make a difference.
Back to CS103!
Part Four: Self-Referential Software
Self-Referential Programs

- If TMs can take other TMs as input, could they take themselves as input?
  
  YES.

- TMs can take their own code as input, and ask questions about (or even execute!) their own code.

- In fact, any computing system that’s equal in power to a Turing machine possesses some mechanism for self-reference.

- Want to see how deep the rabbit hole goes? Take CS154!
Quines

• A **Quine** is a special kind of self-referential program that, when run, prints its own source code.

• Believe it or not, it is possible to write such a program!

• *See zip file with lecture slides for code.*
Self-Referential Programs

- **Claim:** Going forward, assume that any function has the ability to get access to its own source code.
- This means we can write programs like the one shown here:

```cpp
bool narcissist(string input) {
    string me = /* source code of narcissist */;
    return input == me;
}
```
Part Five: Putting It All Together
To Recap

• We’re assuming that, somehow, someone wrote a function
  
  ```c
  bool willAccept(string function, string input);
  ```
  
  that takes the code of a function and an input to that function, then
  
  • returns true if `function(input)` returns true, and
  • returns false if `function(input)` doesn’t return true.

• **Goal:** Show that this decider is “self-defeating;” its power is so great that it undermines itself.

• **Idea:** Convert the fortune teller story into a program.
Trickster pays $137 if the fortune teller answers “yes.”

Trickster pays $42 if the fortune teller answers “no.”

Am I going to pay you $42?
bool willAccept(string function, string input) {
    // Returns true if function(input) returns true. Returns false otherwise.
}

bool trickster(string input) {
    string me = /* source code of trickster */;
    return !willAccept(me, input);
}
bool willAccept(string function, string input) {
    // Returns true if function(input) returns true. Returns false otherwise.
}

bool trickster(string input) {
    string me = /* source code of trickster */;
    return !willAccept(me, input);
}
bool willAccept(string function, string input) {
    // Returns true if function(input) returns true. Returns false otherwise.
}

bool trickster(string input) {
    string me = /* source code of trickster */;
    return !willAccept(me, input);
}

**Theorem:** There is no largest integer.

**Proof sketch:** Suppose for the sake of contradiction that there is a largest integer. Call that integer $n$.

Consider the integer $n+1$.

Notice that $n < n+1$.

But then $n$ isn’t the largest integer.

Contradiction! ■-ish
Theorem: $A_{TM} \notin R$.
**Theorem:** $A_{TM} \notin R$.

**Proof:**

By contradiction; assume that $A_{TM} \in R$. Then there is a decider $D$ for $A_{TM}$. We can represent $D$ as a function $\text{bool willAccept(string function, string w)}$ that takes in the source code of a function $function$ and a string $w$, then returns true if $function(w)$ returns true and returns false otherwise.

Given this, consider this function $\text{trickster}$:

```
bool trickster(string input) {
    string me = /* source code of trickster */;
    return !willAccept(me, input);
}
```

Choose a string $w$. We consider two cases:

**Case 1:** $\text{willAccept(me, input)}$ returns true. Since $\text{willAccept}$ decides $A_{TM}$, this means $\text{trickster}(w)$ returns true. However, given how $\text{trickster}$ is written, in this case $\text{trickster}(w)$ returns false.

**Case 2:** $\text{willAccept(me, input)}$ returns false. Since $\text{willAccept}$ decides $A_{TM}$, this means $\text{trickster}(w)$ returns false. However, given how $\text{trickster}$ is written, in this case $\text{trickster}(w)$ returns true.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \notin R$. ■
**Theorem:** $A_{TM} \notin R$.

**Proof:** By contradiction; assume that $A_{TM} \in R$.
**Theorem:** $A_{TM} \notin \mathbb{R}$.

**Proof:** By contradiction; assume that $A_{TM} \in \mathbb{R}$. Then there is a decider $D$ for $A_{TM}$. We can represent $D$ as a function $\text{bool}\ willAccept(\text{string function}, \text{string } w)$ that takes in the source code of a function $\text{function}$ and a string $w$, then returns true if $\text{function}(w)$ returns true and returns false otherwise.

Given this, consider this function $\text{trickster}$:

$$\text{bool}\ trickster(\text{string input}) = \text{willAccept}(\text{source code of trickster}, \text{input})$$

Choose a string $w$. We consider two cases:

**Case 1:** $\text{willAccept(\text{me}, \text{input})}$ returns true. Since $\text{willAccept}$ decides $A_{TM}$, this means $\text{trickster}(w)$ returns true. However, given how $\text{trickster}$ is written, in this case $\text{trickster}(w)$ returns false.

**Case 2:** $\text{willAccept(\text{me}, \text{input})}$ returns false. Since $\text{willAccept}$ decides $A_{TM}$, this means $\text{trickster}(w)$ returns false. However, given how $\text{trickster}$ is written, in this case $\text{trickster}(w)$ returns true.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \notin \mathbb{R}$. ■
**Theorem:** $A_{TM} \notin \mathbb{R}$.

**Proof:** By contradiction; assume that $A_{TM} \in \mathbb{R}$. Then there is a decider $D$ for $A_{TM}$. We can represent $D$ as a function

```cpp
bool willAccept(string function, string w);
```

that takes in the source code of a function `function` and a string `w`, then returns true if `function(w)` returns true and returns false otherwise.

Given this, consider this function `trickster`:

```cpp
bool trickster(string input) {
    string me = /* source code of trickster */;
    return !willAccept(me, input);
}
```

Choose a string $w$. We consider two cases:

**Case 1:** $\text{willAccept}(\text{me}, \text{input})$ returns true. Since $\text{willAccept}$ decides $A_{TM}$, this means $\text{trickster}(w)$ returns true. However, given how $\text{trickster}$ is written, in this case $\text{trickster}(w)$ returns false.

**Case 2:** $\text{willAccept}($me$, \text{input})$ returns false. Since $\text{willAccept}$ decides $A_{TM}$, this means $\text{trickster}(w)$ returns false. However, given how $\text{trickster}$ is written, in this case $\text{trickster}(w)$ returns true.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \notin \mathbb{R}$. ■
**Theorem:** $A_{TM} \notin R$.

**Proof:** By contradiction; assume that $A_{TM} \in R$. Then there is a decider $D$ for $A_{TM}$. We can represent $D$ as a function

```cpp
bool willAccept(string function, string w);
```

that takes in the source code of a function `function` and a string `w`, then returns true if `function(w)` returns true and returns false otherwise.

Given this, consider this function `trickster`:

```cpp
bool trickster(string input) {
    string me = /* source code of trickster */;
    return !willAccept(me, input);
}
```
**Theorem:** $A_{TM} \notin R$.

**Proof:** By contradiction; assume that $A_{TM} \in R$. Then there is a decider $D$ for $A_{TM}$. We can represent $D$ as a function

```cpp
bool willAccept(string function, string w);
```

that takes in the source code of a function $function$ and a string $w$, then returns true if $function(w)$ returns true and returns false otherwise.

Given this, consider this function `trickster`:

```cpp
bool trickster(string input) {
    string me = /* source code of trickster */;
    return !willAccept(me, input);
}
```

Choose a string $w$. In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \notin R$. ■
**Theorem:** \( A_{\text{TM}} \notin R \).

**Proof:** By contradiction; assume that \( A_{\text{TM}} \in R \). Then there is a decider \( D \) for \( A_{\text{TM}} \). We can represent \( D \) as a function

\[
\text{bool willAccept(string function, string } w);\]

that takes in the source code of a function \( \text{function} \) and a string \( w \), then returns true if \( \text{function}(w) \) returns true and returns false otherwise.

Given this, consider this function \( \text{trickster} \):

\[
\text{bool trickster(string input) }
\begin{align*}
&\quad \text{string me = /* source code of trickster */;} \\
&\quad \text{return !willAccept(me, input);} \\
&\end{align*}
\]

Choose a string \( w \). We consider two cases:
**Theorem:** $A_{TM} \not\in R$.

**Proof:** By contradiction; assume that $A_{TM} \in R$. Then there is a decider $D$ for $A_{TM}$. We can represent $D$ as a function

```cpp
bool willAccept(string function, string w);
```

that takes in the source code of a function $function$ and a string $w$, then returns true if $function(w)$ returns true and returns false otherwise.

Given this, consider this function `trickster`:

```cpp
bool trickster(string input) {
    string me = /* source code of trickster */;
    return !willAccept(me, input);
}
```

Choose a string $w$. We consider two cases:

*Case 1:* $willAccept(me, input)$ returns true.

*Case 2:* $willAccept(me, input)$ returns false.
**Theorem:** \( A_{\text{TM}} \notin R \).

**Proof:** By contradiction; assume that \( A_{\text{TM}} \in R \). Then there is a decider \( D \) for \( A_{\text{TM}} \). We can represent \( D \) as a function

\[
\text{bool willAccept(string function, string w)};
\]

that takes in the source code of a function \( \text{function} \) and a string \( w \), then returns true if \( \text{function}(w) \) returns true and returns false otherwise.

Given this, consider this function \( \text{trickster} \):

\[
\text{bool trickster(string input)} 
\begin{align*}
\text{string me} &= /* \text{source code of trickster} */; \\
\text{return !willAccept(me, input)};
\end{align*}
\]

Choose a string \( w \). We consider two cases:

*Case 1:* \( \text{willAccept(me, input)} \) returns true. Since \( \text{willAccept} \) decides \( A_{\text{TM}} \), this means \( \text{trickster}(w) \) returns true.

*Case 2:* \( \text{willAccept(me, input)} \) returns false. Since \( \text{willAccept} \) decides \( A_{\text{TM}} \), this means \( \text{trickster}(w) \) returns false.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, \( A_{\text{TM}} \notin R \). ■
**Theorem:** \( A_{TM} \notin R \).

**Proof:** By contradiction; assume that \( A_{TM} \in R \). Then there is a decider \( D \) for \( A_{TM} \). We can represent \( D \) as a function

\[
\text{bool willAccept(string function, string w);
}
\]

that takes in the source code of a function \( \text{function} \) and a string \( w \), then returns true if \( \text{function}(w) \) returns true and returns false otherwise.

Given this, consider this function \( \text{trickster} \):

\[
\text{bool trickster(string input) {
    string me = /* source code of trickster */;
    return !willAccept(me, input);
}}
\]

Choose a string \( w \). We consider two cases:

- **Case 1:** \( \text{willAccept}(me, input) \) returns true. Since \( \text{willAccept} \) decides \( A_{TM} \), this means \( \text{trickster}(w) \) returns true. However, given how \( \text{trickster} \) is written, in this case \( \text{trickster}(w) \) returns false.

- **Case 2:** \( \text{willAccept}(me, input) \) returns false. Since \( \text{willAccept} \) decides \( A_{TM} \), this means \( \text{trickster}(w) \) returns false. However, given how \( \text{trickster} \) is written, in this case \( \text{trickster}(w) \) returns true.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, \( A_{TM} \notin R \). ■
**Theorem:** $A_{TM} \not\in R$.

**Proof:** By contradiction; assume that $A_{TM} \in R$. Then there is a decider $D$ for $A_{TM}$. We can represent $D$ as a function

```
bool willAccept(string function, string w);
```

that takes in the source code of a function $function$ and a string $w$, then returns true if $function(w)$ returns true and returns false otherwise.

Given this, consider this function $trickster$:

```
bool trickster(string input) {
    string me = /* source code of trickster */;
    return !willAccept(me, input);
}
```

Choose a string $w$. We consider two cases:

- **Case 1:** $willAccept(me, input)$ returns true. Since $willAccept$ decides $A_{TM}$, this means $trickster(w)$ returns true. However, given how $trickster$ is written, in this case $trickster(w)$ returns false.

- **Case 2:** $willAccept(me, input)$ returns false. Since $willAccept$ decides $A_{TM}$, this means $trickster(w)$ doesn’t return true.
**Theorem:** $A_{TM} \notin R$.

**Proof:** By contradiction; assume that $A_{TM} \in R$. Then there is a decider $D$ for $A_{TM}$. We can represent $D$ as a function

```c
bool willAccept(string function, string w);
```

that takes in the source code of a function `function` and a string `w`, then returns true if `function(w)` returns true and returns false otherwise.

Given this, consider this function `trickster`:

```c
bool trickster(string input) {
    string me = /* source code of trickster */;
    return !willAccept(me, input);
}
```

Choose a string $w$. We consider two cases:

**Case 1:** `willAccept(me, input)` returns true. Since `willAccept` decides $A_{TM}$, this means `trickster(w)` returns true. However, given how `trickster` is written, in this case `trickster(w)` returns false.

**Case 2:** `willAccept(me, input)` returns false. Since `willAccept` decides $A_{TM}$, this means `trickster(w)` doesn’t return true. However, given how `trickster` is written, in this case `trickster(w)` returns true.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \notin R$. ■
**Theorem:** \( A_{TM} \not\in R. \)

**Proof:** By contradiction; assume that \( A_{TM} \in R. \) Then there is a decider \( D \) for \( A_{TM}. \) We can represent \( D \) as a function

\[
\text{bool willAccept(string function, string w)};
\]

that takes in the source code of a function \( \text{function} \) and a string \( w \), then returns true if \( \text{function}(w) \) returns true and returns false otherwise.

Given this, consider this function \( \text{trickster}: \)

\[
\text{bool trickster(string input) } \{ \\
\text{string me } = \text{ /* source code of trickster */}; \\
\text{return !willAccept(me, input);} \\
\}
\]

Choose a string \( w \). We consider two cases:

*Case 1:* \( \text{willAccept(me, input)} \) returns true. Since \( \text{willAccept} \) decides \( A_{TM} \), this means \( \text{trickster}(w) \) returns true. However, given how \( \text{trickster} \) is written, in this case \( \text{trickster}(w) \) returns false.

*Case 2:* \( \text{willAccept(me, input)} \) returns false. Since \( \text{willAccept} \) decides \( A_{TM} \), this means \( \text{trickster}(w) \) doesn’t return true. However, given how \( \text{trickster} \) is written, in this case \( \text{trickster}(w) \) returns true.

In both cases we reach a contradiction, so our assumption must have been wrong.
**Theorem:** $A_{TM} \notin R$.

**Proof:** By contradiction; assume that $A_{TM} \in R$. Then there is a decider $D$ for $A_{TM}$. We can represent $D$ as a function

```cpp
bool willAccept(string function, string w);
```

that takes in the source code of a function `function` and a string `w`, then returns true if `function(w)` returns true and returns false otherwise.

Given this, consider this function `trickster`:

```cpp
bool trickster(string input) {
    string me = /* source code of trickster */;
    return !willAccept(me, input);
}
```

Choose a string $w$. We consider two cases:

- **Case 1:** $\text{willAccept}(me, input)$ returns true. Since $\text{willAccept}$ decides $A_{TM}$, this means $\text{trickster}(w)$ returns true. However, given how `trickster` is written, in this case $\text{trickster}(w)$ returns false.

- **Case 2:** $\text{willAccept}(me, input)$ returns false. Since $\text{willAccept}$ decides $A_{TM}$, this means $\text{trickster}(w)$ doesn’t return true. However, given how `trickster` is written, in this case $\text{trickster}(w)$ returns true.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \notin R$. ■
**Theorem:** $A_{TM} \notin R$.

**Proof:** By contradiction; assume that $A_{TM} \in R$. Then there is a decider $D$ for $A_{TM}$. We can represent $D$ as a function

```cpp
bool willAccept(string function, string w);
```

that takes in the source code of a function `function` and a string `w`, then returns true if `function(w)` returns true and returns false otherwise.

Given this, consider this function `trickster`:

```cpp
bool trickster(string input) {
    string me = /* source code of trickster */;
    return !willAccept(me, input);
}
```

Choose a string $w$. We consider two cases:

**Case 1:** `willAccept(me, input)` returns true. Since `willAccept` decides $A_{TM}$, this means `trickster(w)` returns true. However, given how `trickster` is written, in this case `trickster(w)` returns false.

**Case 2:** `willAccept(me, input)` returns false. Since `willAccept` decides $A_{TM}$, this means `trickster(w)` doesn’t return true. However, given how `trickster` is written, in this case `trickster(w)` returns true.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \notin R$. ■
What Does This Mean?

• In one fell swoop, we've proven that
  • $A_{TM}$ is **undecidable**; there is no general algorithm that can determine whether a TM will accept a string.
  • $R \neq \text{RE}$, because $A_{TM} \notin R$ but $A_{TM} \in \text{RE}$.
• What do these three statements really mean? As in, why should you care?
What exactly does it mean for $A_{TM}$ to be undecidable?

*Intuition: The only general way to find out what a program will do is to run it.*

As you'll see, this means that it's provably impossible for computers to be able to answer most questions about what a program will do.
\[ A_{TM} \notin R \]

- At a more fundamental level, the existence of undecidable problems tells us the following:
  
  **There is a difference between what is true and what we can discover is true.**

- Given a TM \( M \) and a string \( w \), one of these two statements is true:
  
  \( M \) accepts \( w \) \hspace{1cm} \( M \) does not accept \( w \)

But since \( A_{TM} \) is undecidable, there is no algorithm that can always determine which of these statements is true!
Because $R \neq RE$, there is a difference between decidability and recognizability:

*In some sense, it is fundamentally harder to solve a problem than it is to check an answer.*

There are problems where, when the answer is “yes,” you can confirm it (run a recognizer), but where if you don’t have the answer, you can’t come up with it in a mechanical way (build a decider).
Next Time

- **Why All This Matters**
  - Important, practical, undecidable problems.
- **Intuiting RE**
  - What exactly is the class RE all about?
- **Verifiers**
  - A totally different perspective on problem solving.
- **Beyond RE**
  - Finding an impossible problem using very familiar techniques.