The Guide to Self-Reference
Hi everybody!
Self-reference proofs can be pretty hard to understand the first time you see them.
If you're confused – that's okay! It's totally normal. This stuff is tricky.
Once you get a better sense for how to structure these proofs, I think you’ll find that they’re not as bad as they initially seem.
This lecture slide was the first time that we really saw self-reference, and there’s a lot to take in here.
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);
}

Part of the reason why this can be tricky is that what you’re looking at is a finished product. If you don’t have a sense of where it comes from, it’s really hard to understand!
```c
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);
}
```

Let's see where it comes from!
We'll take it from the top.
Let's try to use self-reference to prove that $A_{TM}$ is undecidable.
At a high level, we’re going to do a proof by contradiction.
We’re going to start off by assuming that $A_{TM}$ is decidable.
Somehow, we’re going to try to use this to get to a contradiction.
If we can get a contradiction – any contradiction – we’ll see that our assumption was wrong.
The challenge is figuring out exactly how to go and do this.
Rather than just jumping all the way to the end, let's see what our initial assumption tells us.
We're assuming that $A_{TM}$ is decidable. What does that mean?

Contradiction!
A_{TM} \in \mathbb{R}

There is a decider \( D \) for \( A_{TM} \)

Well, a language is decidable if there's a decider for it, so that means there's some decider for \( A_{TM} \). Let's call that decider \( D \).

Contradiction!
$A_{TM} \in \mathbb{R}$

There is a decider $D$ for $A_{TM}$

What might this decider look like?

Contradiction!
A decider for a language is a Turing machine with a few key properties.

Contradiction!
$A_{TM} \in \mathbb{R}$

There is a decider $D$ for $A_{TM}$

Decider $D$ for $A_{TM}$

First, it has to always halt.

Contradiction!
That means that if you give it any input, it has to either accept or reject it. We’ll visualize this with these two possible outputs.

Contradiction!
There is a decider $D$ for $A_{TM}$

Next, the decider has to tell us something about $A_{TM}$. 

Contradiction!
Next, the decider has to tell us something about \( A_{TM} \).

As a reminder, \( A_{TM} \) is the language \( \{ \langle M, w \rangle | M \text{ is a TM and } M \text{ accepts } w \} \).
Specifically, the decider $D$ needs to take in an input and tell us whether that input is in $A_{TM}$.

As a reminder, $A_{TM}$ is the language

$\{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$
There is a decider \( D \) for \( A_{TM} \)

A\(_{TM}\) is a language of pairs of TMs and strings, so \( D \) will take in two inputs, a machine \( M \) and a string \( w \).

As a reminder, \( A_{TM} \) is the language 

\[ \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \} \]
There is a decider $D$ for $A_{TM}$

If $D$ accepts its input $(M, w)$, it means that $(M, w) \in A_{TM}$, and so $M$ accepts $w$.

As a reminder, $A_{TM}$ is the language \{ $(M, w)$ | $M$ is a TM and $M$ accepts $w$ \}
For any $A_{TM} \in \mathbb{R}$,

There is a decider $D$ for $A_{TM}$

Otherwise, if $D$ rejects its input, it means that $M$ doesn't accept $w$.

Yes, $M$ accepts $w$.

No, $M$ does not accept $w$.

Contradiction!

As a reminder, $A_{TM}$ is the language

\{ $\langle M, w \rangle$ | $M$ is a TM and $M$ accepts $w$ \}
So now we've got this TM $D$ lying around. What can we do with it?

Contradiction!
We've seen TMs that use other TMs as helper functions, and we can do the same here!

Contradiction!
Since TMs are kinda like programs, we can imagine that $D$ is a helper function that looks like this.

There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

Contradiction!
In mathematics, the convention is to use single-letter variable names for everything, which isn't good programming style.

Contradiction!
Here, the method name (willAccept) is just a fancier and more descriptive name for $D$. 

Contradiction!
The two arguments to `willAccept` then correspond to the inputs to the decider $D$. 

Contradiction!
When thinking of $D$ as a decider, we think of it accepting or rejecting. In programming-speak, it's like returning a boolean.

There is a decider $D$ for $A_{TM}$.

We can write programs that use $D$ as a helper function.

Contradiction!
So at this point we’ve just set up the fact that this subroutine exists. What exactly are we going to do with it?
There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

Ultimately, we’re trying to get a contradiction.

Contradiction!
Specifically, we’re going to write a function – which we’ll call *trickster* – that has some really broken behavior... it will accept its input if and only if it doesn’t accept its input!
If you're wondering how on earth you were supposed to figure out that that's the next step, don't panic. The first time you see it, it looks totally crazy. Once you've done this a few times, you'll get a lot more comfortable with it.
We haven’t actually written this trickster function yet. That’s the next step.
If you look at what we’ve said, right now we have a goal of what \textit{trickster} \textbf{should} do, not how \textit{trickster} actually does that.

Contradiction!
There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

Contradiction!

You can think of this requirement as a sort of "design specification."
\[ A_{TM} \in \mathbb{R} \]

There is a decider \( D \) for \( A_{TM} \)

We can write programs that use \( D \) as a helper function

Let's actually go write out a spec for what trickster needs to do!

Contradiction!
Since this requirement is an "if and only if," we can break it down into two cases.

Contradiction!
First, if \textit{trickster} is supposed to accept its input, then it needs to not accept its input.

Contradiction!

We can write programs that use $D$ as a helper function.

\textbf{trickster} design specification:

\textbf{If} \textit{trickster} accepts its input, then \textit{trickster} does not accept its input.

There is a decider $D$ for $A_{\text{TM}}$.
There is a decider $D$ for $A_{TM}$.

We can write programs that use $D$ as a helper function.

Next, if trickster is supposed to not accept its input, then it needs to accept its input.

Contradiction!
We now have a specification for what trickster is supposed to do. Let’s see how to write it!

Contradiction!
A_{TM} \in \mathbb{R}

There is a decider \( D \) for \( A_{TM} \)

We can write programs that use \( D \) as a helper function

\( \text{trickster} \) accepts its input if and only if \( \text{trickster} \) does not accept its input

Contradiction!

\[ \text{Decider } D \text{ for } A_{TM} \]

\[ \text{willAccept} \]

\[ M \]

\[ w \]

\[ \text{input} \]

\[ \text{Yes, } M \text{ accepts } w. \]

\[ \text{No, } M \text{ does not accept } w. \]

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

\[ \text{trickster design specification:} \]

\[ \text{If trickster accepts its input, then} \]

\[ \text{trickster does not accept its input.} \]

\[ \text{If trickster does not accept its input, then} \]

\[ \text{trickster accepts its input.} \]

We’ll write it in the space over to the left.
There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

Contradiction!

bool willAccept(string function, string input)

trickster design specification:
    If trickster accepts its input, then
    trickster does not accept its input.
    If trickster does not accept its input, then
    trickster accepts its input.

bool trickster(string input) {
    This function will take in a single input, then return a boolean.
}
\( A_{TM} \in \mathbb{R} \)

There is a decider \( D \) for \( A_{TM} \)

We can write programs that use \( D \) as a helper function

\textbf{Contradiction!}

\begin{itemize}
  \item We can write programs that use \( D \) as a helper function.
  \item \textbf{Contradiction!}
\end{itemize}
There is a decider $D$ for $A_{TM}$.

We can write programs that use $D$ as a helper function.

**trickster** design specification:
- If trickster accepts its input, then trickster does not accept its input.
- If trickster does not accept its input, then trickster accepts its input.

Contradiction!
We've got this function lying around that will let us know whether any function will accept any input.

Contradiction!

There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

$A_{TM} \in \mathbb{R}$

$M$ accepts $w$.

$M$ does not accept $w$.

bool willAccept(string function, string input)

trickster design specification:
   If trickster accepts its input, then
   trickster does not accept its input.
   If trickster does not accept its input, then
   trickster accepts its input.

bool trickster(string input) {
    
}

We've got this function lying around that will let us know whether any function will accept any input.
There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

trickster accepts its input if and only if trickster does not accept its input

Contradiction!
There is a decider \( D \) for \( A_{TM} \):

- We can write programs that use \( D \) as a helper function.

**trickster design specification:**
- If \( \text{trickster} \) accepts its input, then \( \text{trickster} \) does not accept its input.
- If \( \text{trickster} \) does not accept its input, then \( \text{trickster} \) accepts its input.

```cpp
bool trickster(string input) {
    // Crazy as it seems, that's something we can actually do!
}
```
\( A_{TM} \in R \)

There is a decider \( D \) for \( A_{TM} \)

We can write programs that use \( D \) as a helper function

\textbf{Contradiction!}

\textbf{Decider \( D \) for} \( A_{TM} \)

\textbf{bool} \hspace{1em} \textbf{willAccept(string function, string input)}

\textbf{trickster design specification:}

\begin{itemize}
  \item If \textbf{trickster} accepts its input, then \textbf{trickster} does not accept its input.
  \item If \textbf{trickster} does not accept its input, then \textbf{trickster} accepts its input.
\end{itemize}

\textbf{bool} \hspace{1em} \textbf{trickster(string input) \{ }

\begin{verbatim}
  string me = /* source code of * trickster */;
\end{verbatim}

\}

First, let's have our program get its own source code. (We know this is possible! We saw how to do it in class.)

\begin{itemize}
  \item Yes, \( M \) accepts \( w \).
  \item No, \( M \) does not accept \( w \).
\end{itemize}
There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

We can write programs that use $D$ as a helper function.

$A_{TM} \in \mathbb{R}$

Next, let's call `willAccept` to ask whether we (trickster) are going to accept our input.

Contradiction!

```
bool willAccept(string function, string input) {
    string me = /* source code of trickster */;
    if (willAccept(me, input)) {
    } else {
    }
}

bool trickster(string input) {
    string me = /* source code of trickster */;
    if (willAccept(me, input)) {
    } else {
    }
}
```
\( A_{TM} \in R \)

There is a decider \( D \) for \( A_{TM} \)

- We can write programs that use \( D \) as a helper function

- \( \text{trickster} \) accepts its input if and only if \( \text{trickster} \) does not accept its input

Contradiction!

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

\textbf{trickster design specification:}

- If \text{trickster} accepts its input, then \text{trickster} does not accept its input.
- If \text{trickster} does not accept its input, then \text{trickster} accepts its input.

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

Now, let's look back at our design specification and see what we need to do.

\[
\text{Decider } D \text{ for } A_{TM}\]
\( A_{TM} \in R \)

There is a decider for \( A_{TM} \)

- We can write programs that use \( D \) as a helper function
- \( \text{trickster} \) accepts its input if and only if \( \text{trickster} \) does not accept its input

Contradiction!

\begin{align*}
\text{bool } \text{willAccept} & \text{(string function, string input)} \\
\text{trickster design specification:} & \text{ If } \text{trickster} \text{ accepts its input, then } \text{trickster} \text{ does not accept its input.} \\
& \text{If } \text{trickster} \text{ does not accept its input, then } \text{trickster} \text{ accepts its input.}\end{align*}

\begin{verbatim}
bool trickster(string input) {
    string me = /* source * code of * trickster */;
    if (willAccept(me, input)) {
    } else {
    }
}
\end{verbatim}

Our specification says that, if \( \text{trickster} \) is supposed to accept its input, then it needs to not accept its input.
A_{TM} \in R

There is a decider D for A_{TM}

We can write programs that use D as a helper function

trickster accepts its input if and only if trickster does not accept its input

Contradiction!

There is a decider D for A_{TM}

\textbf{bool} \ willAccept(string function, string input)

\textbf{trickster design specification:}
If \textit{trickster} accepts its input, then \textit{trickster} does not accept its input.
If \textit{trickster} does not accept its input, then \textit{trickster} accepts its input.

\textbf{bool} \ trickster(string input) {
  string me = /* source * code of * trickster */;
  \textbf{if} (willAccept(me, input)) {
  } \textbf{else} {
  }
}
There is a decider $D$ for $A_{TM}$.

We can write programs that use $D$ as a helper function.

trickster accepts its input if and only if trickster does not accept its input.

Contradiction!
There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

$A_{TM} \in \mathbb{R}$

Contradiction!

bool willAccept(string function, string input)

trickster design specification:

✓ If trickster accepts its input, then trickster does not accept its input.

If trickster does not accept its input, then trickster accepts its input.

bool trickster(string input) {
    string me = /* source code of trickster */;
    if (willAccept(me, input)) {
        return false;
    } else {
    }
}
A_{TM} \in \mathbb{R}

There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

trickster accepts its input if and only if trickster does not accept its input

Contradiction!

\begin{align*}
\text{bool willAccept(string function, string input)} \\
\text{trickster design specification:} \\
\checkmark \quad \text{If trickster accepts its input, then trickster does not accept its input.} \\
\text{If trickster does not accept its input, then trickster accepts its input.}
\end{align*}

\begin{verbatim}
bool trickster(string input) {
    string me = /* source code of trickster */;
    if (willAccept(me, input)) {
        return false;
    } else {
        // What about this part?
    }
}
\end{verbatim}
A_{TM} \in \mathbb{R}

There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

$\text{trickster}$ accepts its input if and only if $\text{trickster}$ does not accept its input

Contradiction!

\begin{align*}
\text{bool} \quad \text{willAccept}(\text{string function}, \text{string input}) \\
\text{trickster design specification:} \\
\checkmark \quad \text{If trickster accepts its input, then trickster does not accept its input.} \\
\text{If trickster does not accept its input, then trickster accepts its input.}
\end{align*}

\begin{verbatim}
bool trickster(string input) {
    string me = /* source code of trickster */;
    if (willAccept(me, input)) {
        return false;
    } else {
    }
}
\end{verbatim}

This says that if we aren't supposed to accept the input, then we should accept the input.
There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

contradiction!

Trickster design specification:

- If trickster accepts its input, then trickster does not accept its input.
- If trickster does not accept its input, then trickster accepts its input.

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

so let's go add this line to our program.
A_{TM} \in R

There is a decider \( D \) for \( A_{TM} \)

We can write programs that use \( D \) as a helper function

\textbf{Contradiction!}

\[
\begin{align*}
\text{Decider } D \\
\text{for } A_{TM} \\
\end{align*}
\]

\[
\begin{align*}
M & \rightarrow \text{willAccept} \\
W & \rightarrow \text{input} \\
\end{align*}
\]

\[
\begin{align*}
\text{Yes, } M \text{ accepts } w. \\
\text{No, } M \text{ does not accept } w. \\
\end{align*}
\]

\[
\begin{align*}
\text{bool willAccept(string function, string input)} \\
\end{align*}
\]

\[
\begin{align*}
\text{trickster design specification:} \\
\checkmark \text{ If trickster accepts its input, then trickster does not accept its input.} \\
\checkmark \text{ If trickster does not accept its input, then trickster accepts its input.} \\
\end{align*}
\]

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

And hey! We’re done with this part of the design spec.
There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

**Contradiction!**

\[ A_{TM} \in \mathbb{R} \]

let's take a quick look over our trickster function.

There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

**Contradiction!**
A_{TM} \in R

There is a decider \( D \) for \( A_{TM} \)

We can write programs that use \( D \) as a helper function

\textbf{trickster} accepts its input if and only if \textbf{trickster} does not accept its input

Contradiction!

\[ \begin{array}{c}
\text{function} \quad \text{willAccept} \\
M \\
\text{Decider } D \\
\text{for } A_{TM} \\
\text{input} \quad w \\
\text{Yes, } M \text{ accepts } w. \\
\text{No, } M \text{ does not accept } w. \\
\end{array} \]

\textbf{bool} \ willAccept(string \ function, \ string \ input)

\textbf{trickster} design specification:
\[ \begin{align*}
\checkmark & \quad \text{If } \textbf{trickster} \text{ accepts its input, then} \\
& \quad \textbf{trickster} \text{ does not accept its input.} \\
\checkmark & \quad \text{If } \textbf{trickster} \text{ does not accept its input, then} \\
& \quad \textbf{trickster} \text{ accepts its input.} \\
\end{align*} \]

\textbf{bool} \ trickster(string \ input) { 
\begin{verbatim}
    string me = /* source
                * code of
                * trickster
                */;
    if (willAccept(me, input)) {
        return false;
    } else {
        return true;
    }
\end{verbatim}
}

This is what we said \textbf{trickster} was supposed to do. And hey! That's what it does.
The whole point of this exercise was to get a contradiction.

There is a decider $D$ for $A_{TM}$.

We can write programs that use $D$ as a helper function.

```cpp
bool willAccept(string function, string input) {
  string me = /* source code of trickster */;
  if (willAccept(me, input)) {
    return false;
  } else {
    return true;
  }
}
```

trickster accepts its input if and only if trickster does not accept its input.

Contradiction!
There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

trickster accepts its input if and only if trickster does not accept its input

Contradiction!

$A_{TM} \in \mathbb{R}$

And, indeed, that's what we've done! trickster accepts its input if and only if it doesn't accept.
There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

$	ext{trickster}$ accepts its input if and only if $	ext{trickster}$ does not accept its input

Contradiction!

\[
\begin{align*}
A_{TM} \in \mathbb{R} \\
\implies \quad \text{There is a decider } D \text{ for } A_{TM} \\
\implies \quad \text{We can write programs that use } D \text{ as a helper function} \\
\implies \quad \text{trickster} \text{ accepts its input if and only if trickster does not accept its input} \\
\implies \quad \text{Contradiction!}
\end{align*}
\]

\[
\begin{align*}
\text{bool willAccept(string function, string input)} \\
\text{trickster design specification:} \\
\hspace{1em} \checkmark \quad \text{If trickster accepts its input, then trickster does not accept its input.} \\
\hspace{1em} \checkmark \quad \text{If trickster does not accept its input, then trickster accepts its input.}
\end{align*}
\]

\[
\begin{align*}
\text{bool trickster(string input) \{} \\
\quad \text{string me = /* source code of trickster */;} \\
\quad \text{if (willAccept(me, input)) \{} \\
\quad \hspace{1em} \text{return false;} \\
\quad \text{\} else \{} \\
\quad \hspace{1em} \text{return true;} \\
\quad \text{\} } \\
\\}
\end{align*}
\]

So if you trace through the implications here...
A_{TM} \in R

There is a decider \( D \) for \( A_{TM} \)

We can write programs that use \( D \) as a helper function

trickster accepts its input if and only if trickster does not accept its input

Contradiction!

There is a decider \( D \) for \( A_{TM} \)

\[ M \]

Decider \( D \) for \( A_{TM} \)

\[ w \]

\[ \text{input} \]

\[ \text{willAccept} \]

\[ \text{function} \]

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

\[
\text{trickster design specification:}
\begin{align*}
\checkmark & \quad \text{If trickster accepts its input, then trickster does not accept its input.} \\
\checkmark & \quad \text{If trickster does not accept its input, then trickster accepts its input.}
\end{align*}
\]

\[
\text{bool trickster(string input)} \{
\begin{align*}
\text{string me} & = /* \text{source code of trickster} */; \\
\text{if (willAccept(me, input))} \{ \\
\text{\hspace{1em} return false;}
\} \ 	ext{else} \{ \\
\text{\hspace{1em} return true;}
\}
\}
\]

So if you trace through the implications here...
There is a decider $D$ for $A_{\text{TM}}$.

We can write programs that use $D$ as a helper function.

contradiction!

\[
\text{Decider } D \\
\text{for } A_{\text{TM}}
\]

$M$ willAccept($function$, $input$)

$w$

\[
\begin{align*}
\text{true} & \quad \text{Yes, } M \text{ accepts } w. \\
\text{false} & \quad \text{No, } M \text{ does not accept } w.
\end{align*}
\]

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

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

So if you trace through the implications here...
There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

trickster accepts its input if and only if trickster does not accept its input

Contradiction!

$A_{TM} \in \mathbb{R}$

function

$M$

Decider $D$ for $A_{TM}$

$w$

input

willAccept

Yes, $M$ accepts $w$.

No, $M$ does not accept $w$.

$\text{bool willAccept(string function, string input)}$

trickster design specification:

✓ If trickster accepts its input, then trickster does not accept its input.

✓ If trickster does not accept its input, then trickster accepts its input.

$\text{bool trickster(string input)}$

string me = /* source code of trickster */;

if (willAccept(me, input)) {
    return false;
}
else {
    return true;
}
There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

trickster accepts its input if and only if trickster does not accept its input

Contradiction!
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);
}

Here's that initial lecture slide again.
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);
}

Take a look at it more closely.

willAccept(me, input) returns true
↔
trickster(input) returns false
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);
}
There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

trickster accepts its input if and only if trickster does not accept its input

Contradiction!

$A_{TM} \in \mathbb{R}$

bool willAccept(string function, string input)

trickster design specification:

✓ If trickster accepts its input, then trickster does not accept its input.
✓ If trickster does not accept its input, then trickster accepts its input.

The key idea here is what’s given over there on the left column.
function

\( M \xrightarrow{w} \text{Decider } D \text{ for } A_{\text{TM}} \xrightarrow{\text{willAccept}} \)

\begin{align*}
\text{Yes, } M \text{ accepts } w. \\
\text{No, } M \text{ does not accept } w.
\end{align*}

\begin{verbatim}
bool willAccept(string function, string input) {
    return true;
}
\end{verbatim}

\begin{verbatim}
bool trickster(string input) {
    string me = /* source code of trickster */;
    if (willAccept(me, input)) {
        return false;
    } else {
        return true;
    }
}
\end{verbatim}

We can write programs that use \( D \) as a helper function.

---

A_{\text{TM}} \in \mathbb{R}

There is a decider \( D \) for \( A_{\text{TM}} \).

We can write programs that use \( D \) as a helper function.

\textbf{Contradiction!}

---

\textit{trickster} design specification:

\begin{itemize}
    \item If \textit{trickster} accepts its input, then \textit{trickster} does not accept its input.
    \item If \textit{trickster} does not accept its input, then \textit{trickster} accepts its input.
\end{itemize}

This progression comes up in all the self-reference proofs we’ve done this quarter.
\( A_{TM} \in R \)

There is a decider \( D \) for \( A_{TM} \)

We can write programs that use \( D \) as a helper function

**trickster** accepts its input if and only if **trickster** does not accept its input

Contradiction!

\( bool \) \( willAccept \) (string function, string input)

**trickster** design specification:

\( \forall \) If **trickster** accepts its input, then **trickster** does not accept its input.

\( \forall \) If **trickster** does not accept its input, then **trickster** accepts its input.

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

We'll do another example of this in a little bit.
```cpp
bool trickster(string input) {
    string me = /* source code of trickster */;
    if (willAccept(me, input)) {
        return false;
    } else {
        return true;
    }
}
```

If trickster accepts its input, then trickster does not accept its input. If trickster does not accept its input, then trickster accepts its input.

There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

Contradiction!

Before we move on, though, I thought I'd take a minute to talk about a few common questions we get.
There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

$A_{TM} \in R$

$M \xrightarrow{w} \text{Decider } D$ for $A_{TM}$

$\text{willAccept}$

$\text{willAccept}(\text{string function, string input})$

**trickster design specification:**

- If trickster accepts its input, then trickster does not accept its input.
- If trickster does not accept its input, then trickster accepts its input.

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

First, let’s jump back to this part of trickster.

Contradiction!
\[ A_{TM} \in \mathbb{R} \]

There is a decider \( D \) for \( A_{TM} \)

We can write programs that use \( D \) as a helper function

\[ \text{trickster accepts its input if and only if trickster does not accept its input} \]

Contradiction!

\[ \text{willAccept} \]

\[ \text{Decider } D \text{ for } A_{TM} \]

\[ \text{Input } w \]

\[ \text{Function } M \]

\[ \text{If } M \text{ accepts } w, \text{ then } D \text{ accepts } w. \]

\[ \text{If } M \text{ does not accept } w, \text{ then } D \text{ does not accept } w. \]

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

\[ \text{trickster design specification:} \]

\[ \checkmark \text{ If trickster accepts its input, then trickster does not accept its input.} \]

\[ \checkmark \text{ If trickster does not accept its input, then trickster accepts its input.} \]

\[ \text{bool trickster(string input)} \]

\[ \text{string me = /* source code of } \]

\[ \text{/* trickster */}; \]

\[ \text{if (willAccept(me, input))} \]

\[ \text{return false}; \]

\[ } \text{else } \{
\text{return true; \}
\]
\[ A_{TM} \in \mathbb{R} \]

There is a decider \( D \) for \( A_{TM} \)

We can write programs that use \( D \) as a helper function

trickster accepts its input if and only if trickster does not accept its input

Contradiction!

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

**trickster design specification:**

\[
\begin{align*}
\checkmark & \quad \text{If trickster accepts its input, then trickster does not accept its input.} \\
\checkmark & \quad \text{If trickster does not accept its input, then trickster accepts its input.}
\end{align*}
\]

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

Here, the way we ended up doing that was by having trickster reject its input.
There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

$\textbf{contradiction!}$

$A_{TM} \in \mathbb{R}$

$\textbf{Decider } D \text{ for } A_{TM}$

$M \xrightarrow{w} \text{input}$

$\textbf{bool } \text{willAccept(string function, string input)}$

$\textbf{trickster design specification:}$

$\checkmark$ If trickster accepts its input, then trickster does not accept its input.

$\checkmark$ If trickster does not accept its input, then trickster accepts its input.

$\textbf{bool } \text{trickster(string input)}$

```cpp
string me = /* source code of trickster */;
if (willAccept(me, input)) {
    return false; // <-- here
} else {
    return true;
}
```

I mentioned that there were other things we could do here as well.
There is a decider $D$ for $A_{TM}$.

We can write programs that use $D$ as a helper function.

bool willAccept(string function, string input)

void trickster(string input) {
    string me = /* source code of trickster */;
    if (willAccept(me, input)) {
        while (true) { } 
    } else {
    }
    return true;
}
There is a decider $D$ for $A_{TM}$
We can write programs that use $D$ as a helper function

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

Contradiction!

The design spec here says trickster needs to not accept in this case, and indeed, that's what happens!
bool willAccept(string function, string input) {
    string me = /* source */
        * code of
        * trickster
    */;
    if (willAccept(me, input)) {
        while (true) { }
    } else {
        return true;
    }
}

trickster design specification:

✓ If trickster accepts its input, then
  trickster does not accept its input.
✓ If trickster does not accept its input, then
  trickster accepts its input.

A lot of people ask whether this is allowed, since we were assuming we had a decider and deciders can’t loop.
there is a decider $D$ for $A_{TM}$

we can write programs that use $D$ as a helper function

trickster accepts its input if and only if trickster does not accept its input.

contradiction!
$A_{TM} \in \mathbb{R}$

There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

trickster accepts its input if and only if trickster does not accept its input

Contradiction!

bool willAccept(string function, string input)

trickster design specification:

✓ If trickster accepts its input, then trickster does not accept its input.

✓ If trickster does not accept its input, then trickster accepts its input.

bool trickster(string input) {
    string me = /* source */
    * code of
    * trickster
    * /
    if (willAccept(me, input)) {
        while (true) {
        }
    } else {
        return true;
    }
}
There is a decider $D$ for $A_{TM}$

- We can write programs that use $D$ as a helper function
- trickster accepts its input if and only if trickster does not accept its input

Contradiction!

A_{TM} \in \mathbb{R}

**trickster design specification:**
- If trickster accepts its input, then trickster does not accept its input.
- If trickster does not accept its input, then trickster accepts its input.

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

First, there's this decider $D$. $D$ is a decider, so it's required to halt on all inputs.
There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

Contradiction!

**trickster design specification:**

- If `trickster` accepts its input, then `trickster` does not accept its input.
- If `trickster` does not accept its input, then `trickster` accepts its input.

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

There's also a function `trickster`. `trickster` isn't the decider for $A_{TM}$, so it's not required to halt.
A_{TM} \in \mathbb{R}

There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

trickster accepts its input if and only if trickster does not accept its input

Contradiction!

bool willAccept(string function, string input)

function willAccept

Decider $D$ for $A_{TM}$

$M \rightarrow w$

input

Yes, $M$ accepts $w$.

No, $M$ does not accept $w$.

bool trickster(string input) {
    string me = /* source code of */
        * trickster */;
    if (willAccept(me, input)) {
        while (true) {
        }
    } else {
        return true;
    }
}
\( A_{TM} \in \mathbb{R} \)

There is a decider \( D \) for \( A_{TM} \)

We can write programs that use \( D \) as a helper function

\textbf{trickster} accepts its input if and only if \( \textbf{trickster} \) does not accept its input

Contradiction!

\begin{itemize}
  \item \textbf{willAccept} \textbf{function} \( \text{string function, string input} \)
  \item \textbf{bool} \textbf{willAccept} (string function, string input)
\end{itemize}

\textbf{trickster} design specification:
- If \textbf{trickster} accepts its input, then \textbf{trickster} does not accept its input.
- If \textbf{trickster} does not accept its input, then \textbf{trickster} accepts its input.

\begin{itemize}
  \item \textbf{bool} \textbf{trickster} (string input) {
    \begin{verbatim}
    string me = /* source code of * trickster */;
    if (\textbf{willAccept}(me, input)) {
        \textbf{while} (true) {
    }
    } else {
        return true;
    }
    
    \end{verbatim}
  }
\end{itemize}

The decider is always required to halt, but \textbf{trickster} is not.
\( A_{TM} \in \mathbb{R} \)

There is a decider \( D \) for \( A_{TM} \)

We can write programs that use \( D \) as a helper function

\textbf{trickster design specification:}

\begin{itemize}
  \item If trickster accepts its input, then trickster does not accept its input.
  \item If trickster does not accept its input, then trickster accepts its input.
\end{itemize}

\begin{verbatim}
bool willAccept(string function, string input) {
    string me = /* source code of trickster */;
    if (willAccept(me, input)) {
        while (true) { }
    } else {
        return true;
    }
}
\end{verbatim}

Let's undo all these changes so that we can talk about the next common question.
A_{TM} \in \mathbb{R}

There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

trickster accepts its input if and only if trickster does not accept its input

Contradiction!

$$\text{function } M \quad \text{willAccept}$$

Decider $D$ for $A_{TM}$

$w$ as input

Yes, $M$ accepts $w$.

No, $M$ does not accept $w$.

bool willAccept(string function, string input)

trickster design specification:

✓ If trickster accepts its input, then trickster does not accept its input.
✓ If trickster does not accept its input, then trickster accepts its input.

bool trickster(string input) {
    string me = /* source code of trickster */;
    if (willAccept(me, input)) {
        return false;
    } else {
        return true;
    }
}

Much better!
There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

```cpp
bool willAccept(string function, string input) {
    string me = /* source code of trickster */;
    if (willAccept(me, input)) {
        return false;
    } else {
        return true;
    }
}
```

### trickster design specification:

- If trickster accepts its input, then trickster does not accept its input.
- If trickster does not accept its input, then trickster accepts its input.

---

Contradiction!
If trickster accepts its input, then trickster does not accept its input.

If trickster does not accept its input, then trickster accepts its input.

We can write programs that use $D$ as a helper function.

There is a decider $D$ for $A_{TM}$.

$A_{TM} \in R$

A lot of people take a look at the program we’ve written...

```
bool willAccept(string function, string input) {
    string me = /* source code of trickster */;
    if (willAccept(me, input)) {
        return false;
    } else {
        return true;
    }
}
```
If trickster accepts its input, then trickster does not accept its input.

If trickster does not accept its input, then trickster accepts its input.

There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

trickster accepts its input if and only if trickster does not accept its input

Contradiction!

$$A_{TM} \in R$$

function $M$

Decider $D$ for $A_{TM}$

w

input

$$\text{bool willAccept(string function, string input)}$$

trickster design specification:

✓ If trickster accepts its input, then trickster does not accept its input.
✓ If trickster does not accept its input, then trickster accepts its input.

```cpp
bool trickster(string input) {
    string me = /* source code of trickster */;
    if (willAccept(me, input)) {
        return false; // <-- here
    } else {
        return true; // <-- here
    }
}
```
A_{TM} \in \mathbb{R}

There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

trickster accepts its input if and only if trickster does not accept its input

Contradiction!

function $M$ \rightarrow willAccept

Decider $D$ for $A_{TM}$

\[ \begin{align*}
    w & \rightarrow \text{input} \\
    \text{Yes, } M \text{ accepts } w. \\
    \text{No, } M \text{ does not accept } w.
\end{align*} \]

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

trickster design specification:

\[ \begin{align*}
    &\checkmark \quad \text{If trickster accepts its input, then} \\
    &\quad \text{trickster does not accept its input}. \\
    &\checkmark \quad \text{If trickster does not accept its input, then} \\
    &\quad \text{trickster accepts its input}.
\end{align*} \]

\[ \text{bool trickster(string input)} \{
    \text{string me = /* source code of trickster */;
    if (willAccept(me, input)) { \\
        return true; // <-- swap!
    } else {
        return false; // <-- swap!
    }
\} \]

... and swap them like this.
If \( A_{TM} \in R \)

There is a decider \( D \) for \( A_{TM} \)

We can write programs that use \( D \) as a helper function

\( \text{trickster} \) accepts its input if and only if \( \text{trickster} \) does not accept its input

Contradiction!

\[ A_{TM} \in R \]

\[ \text{willAccept} \]

\[ \text{Decider } D \text{ for } A_{TM} \]

\[ \begin{align*} w &\quad \text{ input} \quad M \quad \text{ function} \quad \text{willAccept} \quad \text{Decider } D \text{ for } A_{TM} \quad \text{Yes, } M \text{ accepts } w. \quad \text{No, } M \text{ does not accept } w. \end{align*} \]

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

\[ \text{trickster design specification:} \]

\[ \begin{align*} \checkmark \ &\text{If trickster accepts its input, then} \quad \text{trickster does not accept its input.} \\ \checkmark \ &\text{If trickster does not accept its input, then} \quad \text{trickster accepts its input.} \end{align*} \]

\[ \text{bool trickster(string input)} \]

\[ \begin{align*} &\text{string me} = \text{/* source} \\ &\quad \text{/* code of} \\ &\quad \text{/* trickster} \\ &\quad \text{/*}; \\ &\text{if (willAccept(me, input))} \{ \\ &\quad \text{return true;} \\ &\} \text{ else} \{ \\ &\quad \text{return false;} \\ &\} \} \]

\[ \text{Usually, people ask whether we could have done this and proved that } A_{TM} \in R. \]
A_{TM} \in R

There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

trickster accepts its input if and only if 
trickster does not accept its input

Contradiction!

bool willAccept(string function, string input)

trickster design specification:
✓ If trickster accepts its input, then 
trickster does not accept its input.
✓ If trickster does not accept its input, then 
trickster accepts its input.

bool trickster(string input) {
    string me = /* source
    * code of
    * trickster
    */;
    if (willAccept(me, input)) {
        return true;
    } else {
        return false;
    }
}

Turns out, that doesn’t work. Let’s see why.
There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

Contradiction!

**trickster** design specification:

- If trickster accepts its input, then trickster does not accept its input.
- If trickster does not accept its input, then trickster accepts its input.

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

Notice that this trickster doesn’t have the behavior given over here.
\( A_{TM} \in \mathbb{R} \)

There is a decider \( D \) for \( A_{TM} \)

We can write programs that use \( D \) as a helper function

\textbf{trickster} accepts its input if and only if \textbf{trickster} accepts its input

Contradiction!

\textbf{bool} \ willAccept(string \ function, \ string \ input) \n
\textbf{trickster} design specification:
\begin{itemize}
  \item If \textbf{trickster} accepts its input, then \textbf{trickster} does not accept its input.
  \item If \textbf{trickster} does not accept its input, then \textbf{trickster} accepts its input.
\end{itemize}

\textbf{bool} \ trickster(string \ input) \{
  \textbf{string} \ me \ = \ /* \ source \n      \ * \ code \ of \n      \ * \ trickster \n      */;
  \textbf{if} \ (\textbf{willAccept}(me, \ input)) \{
    \textbf{return} \ \textbf{true};
  \} \ \textbf{else} \{
    \textbf{return} \ \textbf{false};
  \}
\}

If you think about the behavior it \textit{does} have, it looks more like this.
\( A_{TM} \in R \)

There is a decider \( D \) for \( A_{TM} \)

We can write programs that use \( D \) as a helper function

\textbf{Contradiction!}

\begin{verbatim}
bool willAccept(string function, string input) {
    string me = /* source
        * code of
        * trickster
    */;
    if (willAccept(me, input)) {
        return true;
    } else {
        return false;
    }
}
\end{verbatim}

\textbf{trickster} design specification:
\begin{itemize}
    \item If \textbf{trickster} accepts its input, then
        \textbf{trickster} does not accept its input.
    \item If \textbf{trickster} does not accept its input, then
        \textbf{trickster} accepts its input.
\end{itemize}

Notice that this is a true statement.
If trickster accepts its input, then trickster does not accept its input.

If trickster does not accept its input, then trickster accepts its input.

\[ A_{TM} \in \mathbb{R} \]

There is a decider \( D \) for \( A_{TM} \)

We can write programs that use \( D \) as a helper function

\textbf{trickster} accepts its input if and only if trickster accepts its input

Contradiction!

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

\textbf{trickster} design specification:

\checkmark \text{If trickster accepts its input, then trickster does not accept its input.}

\checkmark \text{If trickster does not accept its input, then trickster accepts its input.}

\begin{verbatim}
bool trickster(string input) {
    string me = /* source code of */ trickster */ ;
    if (willAccept(me, input)) {
        return true;
    } else {
        return false;
    }
}
\end{verbatim}

Originally, we got a contradiction here.
If trickster accepts its input, then trickster does not accept its input.
If trickster does not accept its input, then trickster accepts its input.

There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

```
bool willAccept(string function, string input) {
    string me = /* source code of trickster */;
    if (willAccept(me, input)) {
        return true;
    } else {
        return false;
    }
}
```

We can write programs that use $D$ as a helper function

Instead, we've shown that we end up at a true statement.
If trickster accepts its input, then trickster does not accept its input.

If trickster does not accept its input, then trickster accepts its input.

```cpp
bool willAccept(string function, string input) {
    string me = /* source code of */
       /* trickster */;
    if (willAccept(me, input)) {
        return true;
    } else {
        return false;
    }
}
```

We can write programs that use $D$ as a helper function.

There is a decider $D$ for $A_{TM}$

A_TM ∈ R

Decider $D$ for $A_{TM}$

$M$ → $w$

input

Yes, $M$ accepts $w$.

No, $M$ does not accept $w$.

However, take a minute to look at the giant implication given here.
If `trickster` accepts its input, then `trickster` does not accept its input.

If `trickster` does not accept its input, then `trickster` accepts its input.

There is a decider `D` for `A_{TM}`

Decider `D` for `A_{TM}`

We can write programs that use `D` as a helper function.

We can write programs that use `D` as a helper function.

Trickster design specification:

- If trickster accepts its input, then trickster does not accept its input.
- If trickster does not accept its input, then trickster accepts its input.

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

Overall, this shows that

`A_{TM} ∈ R → T`
If trickster accepts its input, then trickster does not accept its input.

If trickster does not accept its input, then trickster accepts its input.

A_{\text{TM}} \in \mathbb{R}

There is a decider D for A_{\text{TM}}

We can write programs that use D as a helper function

trickster accepts its input if and only if trickster accepts its input

bool willAccept(string function, string input)

trickster design specification:

✓ If trickster accepts its input, then trickster does not accept its input.
✓ If trickster does not accept its input, then trickster accepts its input.

bool trickster(string input) {
    string me = /* source code of trickster */;
    if (willAccept(me, input)) {
        return true;
    } else {
        return false;
    }
}
If trickster accepts its input, then trickster does not accept its input.

If trickster does not accept its input, then trickster accepts its input.

There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

trickster accepts its input if and only if trickster accepts its input

$A_{TM} \in \mathbb{R}$

There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

trickster accepts its input if and only if trickster accepts its input

$bool$ willAccept($string$ function, $string$ input)

trickster design specification:
- If trickster accepts its input, then trickster does not accept its input.
- If trickster does not accept its input, then trickster accepts its input.

$bool$ trickster($string$ input) {
    string me = /* source
    * code of
    * trickster
    */;
    if (willAccept(me, input)) {
        return true;
    } else {
        return false;
    }
}
If trickster accepts its input, then trickster does not accept its input.

If trickster does not accept its input, then trickster accepts its input.

There is a decider $D$ for $A_{TM}$.

We can write programs that use $D$ as a helper function.

trickster accepts its input if and only if trickster accepts its input.

We have no way of knowing whether $A_{TM} \in \mathbb{R}$ or not just by looking at this statement.
If trickster accepts its input, then trickster does not accept its input.
If trickster does not accept its input, then trickster accepts its input.

\[
\text{A}_{\text{TM}} \in \mathbb{R}
\]

There is a decider \( D \) for \( A_{\text{TM}} \)

We can write programs that use \( D \) as a helper function

\text{trickster} accepts its input if and only if trickster accepts its input

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

\[
\text{trickster design specification:}
\]
\[
\checkmark \quad \text{If trickster accepts its input, then}
\]
\[
\checkmark \quad \text{trickster does not accept its input.}
\]
\[
\checkmark \quad \text{If trickster does not accept its input, then}
\]
\[
\checkmark \quad \text{trickster accepts its input.}
\]

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

The fact that we didn’t get a contradiction doesn’t mean that \( A_{\text{TM}} \) is decidable.
If trickster accepts its input, then trickster does not accept its input.
If trickster does not accept its input, then trickster accepts its input.

There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

trickster accepts its input if and only if trickster accepts its input

A_{TM} \in \mathbb{R}

Decider $D$ for $A_{TM}$

function $M$ willAccept

$w$ input

Yes, $M$ accepts $w.$
No, $M$ does not accept $w.$

bool willAccept(string function, string input)

trickster design specification:
✓ If trickster accepts its input, then trickster does not accept its input.
✓ If trickster does not accept its input, then trickster accepts its input.

bool trickster(string input) {
    string me = /* source * code of * trickster */;
    if (willAccept(me, input)) {
        return true;
    } else {
        return false;
    }
}

Just so we don’t get confused, let’s reset everything back to how it used to be.
A_{TM} \in \mathbb{R}

There is a decider \( D \) for \( A_{TM} \)

We can write programs that use \( D \) as a helper function

\textbf{trickster} accepts its input if and only if \( \text{trickster} \) does not accept its input.

Contradiction!

**bool** \( \text{willAccept} \)(string function, string input)

\textbf{trickster} design specification:

\[ \begin{align*}
\therefore & \text{If trickster accepts its input, then trickster does not accept its input.} \\
\therefore & \text{If trickster does not accept its input, then trickster accepts its input.}
\end{align*} \]

\begin{verbatim}
bool trickster(string input) {
    string me = /* source code of trickster */;
    if (willAccept(me, input)) {
        return false;
    } else {
        return true;
    }
}
\end{verbatim}
A_{TM} \in \mathbb{R}

There is a decider $D$ for $A_{TM}$

We can write programs that use $D$ as a helper function

trickster accepts its input if and only if trickster does not accept its input

Contradiction!

bool willAccept(string function, string input)

trickster design specification:

✓ If trickster accepts its input, then trickster does not accept its input.
✓ If trickster does not accept its input, then trickster accepts its input.

bool trickster(string input) {
    string me = /* source code of trickster */;
    if (willAccept(me, input)) {
        return false;
    } else {
        return true;
    }
}
Do you remember the secure voting problem from lecture?
We said that a TM $M$ is a secure voting machine if it obeys the above rule. $M$ is a secure voting machine if and only if it accepts its input precisely if it has more $r$'s than $d$'s.
Our goal was to show that it’s not possible to build a program that can tell whether an arbitrary TM is a secure voting machine.

$M$ is a secure voting machine if and only if $M$ accepts its input precisely if it has more $r$’s than $d$’s.
$M$ is a secure voting machine if and only if $M$ accepts its input precisely if it has more $r$’s than $d$’s.

Notice that our goal was not to show that you can’t build a secure voting machine.
$M$ is a secure voting machine if and only if $M$ accepts its input precisely if it has more $r$’s than $d$’s.

It’s absolutely possible to do that.

```cpp
bool countVotes(string input) {
    return countRs(input) > countDs(input);
}
```
The hard part is being able to tell whether an arbitrary program is a secure voting machine.

\[ M \text{ is a secure voting machine if and only if } \]
\[ M \text{ accepts its input precisely if it has more } r\text{'s than } d\text{'s.} \]

```cpp
bool countVotes(string input) {
    return countRs(input) > countDs(input);
}
```
Here's a program where no one knows whether it's a secure voting machine.

```cpp
bool mystery(string input) {
    int n = countRs(input);
    while (n > 1) {
        if (n % 2 == 0) n = n / 2;
        else n = 3*n + 1;
    }
    return countRs(input) > countDs(input);
}
```

$M$ is a secure voting machine if and only if $M$ accepts its input precisely if it has more \texttt{r}'s than \texttt{d}'s.
$M$ is a secure voting machine if and only if $M$ accepts its input precisely if it has more $r$’s than $d$’s.

You can see this because no one knows whether this part will always terminate.

```c
bool mystery(string input) {
    int n = countRs(input);
    while (n > 1) {
        if (n % 2 == 0) n = n / 2;
        else n = 3*n + 1;
    }
    return countRs(input) > countDs(input);
}
```
$M$ is a secure voting machine if and only if $M$ accepts its input precisely if it has more $r$’s than $d$’s.

It’s entirely possible that this goes into an infinite loop on some input – we’re honestly not sure!

```cpp
bool mystery(string input) {
    int n = countRs(input);
    while (n > 1) {
        if (n % 2 == 0) n = n / 2;
        else n = 3*n + 1;
    }
    return countRs(input) > countDs(input);
}
```
So, to recap:
Building a secure voting machine isn’t hard.
Checking whether an arbitrary program is a secure voting machine is really hard.

\[ M \text{ is a secure voting machine if and only if } M \text{ accepts its input precisely if it has more } r\text{'s than } d\text{'s.} \]
Our goal is to show that the secure voting problem – the problem of checking whether a program is a secure voting machine – is undecidable.
Following our pattern from before, we'll assume that the secure voting problem is decidable.
The secure voting problem is decidable.

We're ultimately trying to get some kind of contradiction here.
As before, we'll take it one step at a time.

Contradiction!
First, since we're assuming that the secure voting problem is decidable, we're assuming that there's a decider for it.
The secure voting problem is decidable.

Contradiction!

There is a decider $D$ for the secure voting problem.

Decider $D$ for the secure voting problem

So what does that look like?
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

A decider for the secure voting problem will take in some TM $M$, which is the machine we want to specifically check.
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem

The decider will then accept if $M$ is a secure voting machine and reject otherwise.

Contradiction!

Yes, $M$ is a secure voting machine.

No, $M$ is not a secure voting machine.
The secure voting problem is decidable.

There is a decider \( D \) for the secure voting problem

We can write programs that use \( D \) as a helper function

Contradiction!

Following our pattern from before, we’ll then say that we can use this decider as a subroutine in other TMs.
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

Contradiction!

In software, that decider $D$ might look something like what’s given above.

```cpp
bool isSecure(string function) {
    // Implementation details...
}
```
The secure voting problem is decidable.  

There is a decider $D$ for the secure voting problem.  

We can write programs that use $D$ as a helper function.

Contradiction!

Here, $\text{isSecure}$ is just another name for the decider $D$, but with a more descriptive name.

```python
bool isSecure(string function)
```

Yes, $M$ is a secure voting machine.  

No, $M$ is not a secure voting machine.
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem

We can write programs that use $D$ as a helper function

Contradiction!
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem

We can write programs that use $D$ as a helper function

Contradiction!

Function

$M$ for the secure voting problem $isSecure$

Decider $D$

Yes, $M$ is a secure voting machine.

No, $M$ is not a secure voting machine.

bool isSecure(string function)

This was the point in the previous proof where we started to write a design spec for some self-referential function $\text{trickster}$. 
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

Contradiction!

Previously, we wrote trickster to get this contradiction:

“trickster accepts its input if and only if trickster doesn’t accept its input.”
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

Contradiction!

That was a great contradiction to get when we had a decider that would tell us whether a program would accept a given input.
The problem here is that our decider doesn't do that. Instead, it tells us whether a program is a secure voting machine.

The secure voting problem is decidable. There is a decider $D$ for the secure voting problem. We can write programs that use $D$ as a helper function.

Contradiction!
Following the maxim of "do what you can with what you have where you are," we’ll try to set up a contradiction concerning whether a program is or is not a voting machine.

The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem

We can write programs that use $D$ as a helper function

Contradiction!
Specifically, we're going to write trickster so it's a secure voting machine if and only if it's not a secure voting machine.

The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

**Contradiction!**

```
bool isSecure(string function)

Specifically, we're going to write trickster so it's a secure voting machine if and only if it's not a secure voting machine.
```
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

Contradiction!

Generally speaking, you’ll try to set up a contradiction where the program has the property given by the decider if and only if it doesn’t have the property given by the decider.

Contradiction!
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem

We can write programs that use $D$ as a helper function

trickster is secure if and only if trickster is not secure.

Contradiction!

function

$M$

Decider $D$ for the secure voting problem

isSecure

Yes, $M$ is a secure voting machine.

No, $M$ is not a secure voting machine.

**bool** isSecure(string function)

Generally speaking, you’ll try to set up a contradiction where the program has the property given by the decider if and only if it doesn’t have the property given by the decider.

Pay attention to that other guy! That’s really, really good advice!
So now we have to figure out how to write this function `trickster`.

The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

Contradiction!

```
bool isSecure(string function)
```

So now we have to figure out how to write this function `trickster`.
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

Contradiction!

**bool** isSecure(string function)

trickster design specification:

As before, let’s start by writing out a design specification for what it’s supposed to do.
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem $M$.

We can write programs that use $D$ as a helper function.

Contradiction!

This first part takes care of the first half of the biconditional.
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

Contradiction!

This second part takes care of the other direction.

\textbf{bool} isSecure(string function)

\textbf{trickster} design specification:

- If \textbf{trickster} is a secure voting machine, then \textbf{trickster} is not a secure voting machine.
- If \textbf{trickster} is not a secure voting machine, then \textbf{trickster} is a secure voting machine.
The secure voting problem is decidable.

Contradiction!

We can write programs that use $D$ as a helper function.

trickster design specification:

- If trickster is a secure voting machine, then trickster is not a secure voting machine.
- If trickster is not a secure voting machine, then trickster is a secure voting machine.

At this point, we have written out a spec for what we want trickster to do. All that’s left to do now is to code it up!
In lecture, we wrote one particular program that met these requirements. For the sake of simplicity, I’m going to write a different one here. Don’t worry! It works just fine.

The secure voting problem is decidable.

There is a decider \( D \) for the secure voting problem

We can write programs that use \( D \) as a helper function

\( \text{isSecure} \) design specification:

- If \text{trickster} is a secure voting machine, then \text{trickster} is not a secure voting machine.
- If \text{trickster} is not a secure voting machine, then \text{trickster} is a secure voting machine.
The secure voting problem is decidable.

Contradiction!

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

trickster is secure if and only if trickster is not secure.

Decider $D$ for the secure voting problem

$M$ function $isSecure$

Yes, $M$ is a secure voting machine.

No, $M$ is not a secure voting machine.

bool isSecure(string function)

design specification:

trickster is a secure voting machine, then

If trickster is a secure voting machine, then

If trickster is not a secure voting machine, then

trickster is a secure voting machine.

bool trickster(string input) {

}

Our trickster starts off as a regular boolean function.
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

Contradiction!

Decider $D$ for the secure voting problem

$M$

isSecure

$M$

isSecure

M is a secure voting machine.

M is not a secure voting machine.

$$\textbf{bool} \ \text{isSecure(}\text{string function}\text{)}$$

trickster design specification:

If trickster is a secure voting machine, then
trickster is not a secure voting machine.
If trickster is not a secure voting machine, then
trickster is a secure voting machine.

$$\textbf{bool} \ \text{trickster(}\text{string input}\text{)} \ {\{ \}}$$

Ultimately, we need to figure out if we’re a secure voting machine or not.
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

trickster is secure if and only if trickster is not secure.

Contradiction!
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem

We can write programs that use $D$ as a helper function

contradiction!

We can write programs that use $D$ as a helper function

trickster design specification:

If trickster is a secure voting machine, then trickster is not a secure voting machine.
If trickster is not a secure voting machine, then trickster is a secure voting machine.

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

As before, we'll use the fact that we have this decider lying around to make trickster figure out what exactly it does.
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

Contradiction!

$\text{bool } \text{isSecure}(\text{string function})$

**trickster design specification:**

- If trickster is a secure voting machine, then trickster is not a secure voting machine.
- If trickster is not a secure voting machine, then trickster is a secure voting machine.

```c
bool trickster(string input) {
    string me = /* source code */
               * of trickster */;
    if (isSecure(me)) {
    } else {
    }
}
```

Specifically, let's have trickster ask what it's going to do.
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

Contradiction!

```c
bool isSecure(string function) {
    string me = /* source code */
                * of trickster */;
    if (isSecure(me)) {
    } else { 
    } 
}
```

**trickster design specification:**
- If `trickster` is a secure voting machine, then `trickster` is not a secure voting machine.
- If `trickster` is not a secure voting machine, then `trickster` is a secure voting machine.

Let's take it one step at a time.
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

Contradiction!

### bool isSecure(string function)

**trickster design specification:**

If trickster is a secure voting machine, then trickster is not a secure voting machine.

If trickster is not a secure voting machine, then trickster is a secure voting machine.

```cpp
bool trickster(string input) {
    string me = /* source code */
    if (isSecure(me)) {
    } else {
    }
}
```

Oddly enough, let's look at the second requirement first.

Why? I ask: why not?
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem

We can write programs that use $D$ as a helper function

trickster is secure if and only if trickster is not secure.

Contradiction!
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

Contradiction!

\begin{align*}
\text{trickster design specification:} & \\
\text{If trickster is a secure voting machine, then trickster is not a secure voting machine.} & \\
\text{If trickster is not a secure voting machine, then trickster is a secure voting machine.} & \\
\end{align*}

\begin{verbatim}
bool trickster(string input) {
    string me = /* source code of trickster */;
    if (isSecure(me)) {
    } else {
    }
}
\end{verbatim}

This case is the part that drops us in the “else” branch of this if statement, so let’s focus on that part for now.
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

Contradiction!

The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

Contradiction!
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

Contradiction!

The function $M$ for the secure voting problem is decidably.

Yes, $M$ is a secure voting machine.

No, $M$ is not a secure voting machine.

Function $isSecure$ (string function)

bool isSecure(string function) {
}

trickster design specification:

If trickster is a secure voting machine, then trickster is not a secure voting machine.

If trickster is not a secure voting machine, then trickster is a secure voting machine.

bool trickster(string input) {
    string me = /* source code of trickster */;
    if (isSecure(me)) {
    } else {
    }
}

That means we need to make trickster accept all strings with more r's than d's and not accept anything else.
The secure voting problem is decidable.

There is a decider D for the secure voting problem

We can write programs that use D as a helper function

trickster is secure if and only if trickster is not secure.

Contradiction!
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

Contradiction!

The code looks something like this.

```cpp
bool isSecure(string function) {
    string me = /* source code */
    if (isSecure(me)) {
    } else {
        return countRsIn(input) > countDsIn(input);
    };
}
```

```cpp
bool trickster(string input) {
    string me = /* source code */
    if (isSecure(me)) {
    } else {
        return countRsIn(input) > countDsIn(input);
    }
}
```

$M$ is a decider for the secure voting problem.

Yes, $M$ is a secure voting machine.

No, $M$ is not a secure voting machine.

**trickster design specification:**

If trickster is a secure voting machine, then trickster is not a secure voting machine.

If trickster is not a secure voting machine, then trickster is a secure voting machine.
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem

We can write programs that use $D$ as a helper function

Contradiction!

Function $isSecure$ is defined:

```cpp
bool isSecure(string function) {
    Decider $D$ for the secure voting problem
}
```

**trickster design specification:**

- If $trickster$ is a secure voting machine, then $trickster$ is not a secure voting machine.
- If $trickster$ is not a secure voting machine, then $trickster$ is a secure voting machine.

```cpp
bool trickster(string input) {
    string me = /* source code */ of trickster */;
    if (isSecure(me)) {
    } else {
        return countRsIn(input) > countDsIn(input);
    }
}
```

Just to confirm that this works - notice that if the input has more r's than d's, we accept it, and otherwise we reject.
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem

We can write programs that use $D$ as a helper function

Contradiction!

function $M$ for the secure voting problem

$D$ is a secure voting decider.

Yes, $M$ is a secure voting machine.

No, $M$ is not a secure voting machine.

bool isSecure(string function)

taxtiker design specification:
If trickster is a secure voting machine, then
taxtiker is not a secure voting machine.
√ If trickster is not a secure voting machine, then
taxtiker is a secure voting machine.

bool trickster(string input) {
    string me = /* source code
        * of trickster */;
    if (isSecure(me)) {
    } else {
        return countRsIn(input) > countDsIn(input);
    }
}
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

Contradiction!

We can write programs that use $D$ as a helper function.

The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

Contradiction!

bool isSecure(string function)

trickster design specification:

If trickster is a secure voting machine, then
trickster is not a secure voting machine.

If trickster is not a secure voting machine, then
trickster is a secure voting machine.

bool trickster(string input) {
    string me = /* source code
    * of trickster
    */;
    if (isSecure(me)) {
    }
    else {
        return countRsIn(input) >
        countDsIn(input);
    }
}
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

Contradiction!

The boolean function $\text{isSecure}(\text{string function})$ accepts a string function as input and returns true if the function is secure, and false otherwise.

The boolean function $\text{trickster}(\text{string input})$ takes a string input and returns true if there are more 'R's than 'D's in the input, and false otherwise.

The trickster design specification states:

- If $\text{trickster}$ is a secure voting machine, then $\text{trickster}$ is not a secure voting machine.
- If $\text{trickster}$ is not a secure voting machine, then $\text{trickster}$ is a secure voting machine.

This says that if $\text{trickster}$ is supposed to be a secure voting machine, it needs to not be a secure voting machine.
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

The secure voting problem

Contradiction!

There are a lot of ways to get trickster to not be a secure voting machine.

$\text{trickster} \text{ design specification:}$

- If $\text{trickster}$ is a secure voting machine, then $\text{trickster}$ is not a secure voting machine.
- If $\text{trickster}$ is not a secure voting machine, then $\text{trickster}$ is a secure voting machine.

```c
bool trickster(string input) {
    string me = /* source code */;
    if (isSecure(me)) {
    } else {
        return countRsIn(input) > countDsIn(input);
    }
}
```
The secure voting problem is decidable.

There is a decider \( D \) for the secure voting problem

We can write programs that use \( D \) as a helper function

Contradiction!

The secure voting problem is decidable.

There is a decider \( D \) for the secure voting problem

We can write programs that use \( D \) as a helper function

Contradiction!
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

Contradiction!

We can write programs that use $D$ as a helper function.

```
bool trickster(string input) {
    string me = /* source code */;
    if (isSecure(me)) {
        return true;
    } else {
        return countRsIn(input) > countDsIn(input);
    }
}
```

Among the many things we can do that falls into the "literally anything else" camp would be to just accept everything.
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

Contradiction!

```cpp
bool isSecure(string function) {
    return true;
}
```

```cpp
bool trickster(string input) {
    string me = /* source code */;
    if (isSecure(me)) {
        return true;
    } else {
        return countRsIn(input) > countDsIn(input);
    }
}
```

trickster design specification:

- If `trickster` is a secure voting machine, then `trickster` is not a secure voting machine.
- If `trickster` is not a secure voting machine, then `trickster` is a secure voting machine.

Notice that in this case, trickster is not a secure voting machine: it accepts everything, including a ton of strings it's not supposed to.
The secure voting problem is decidable.

There is a decider \( D \) for the secure voting problem.

We can write programs that use \( D \) as a helper function.

Contradiction!

\[
\text{isSecure}(\text{string function})
\]

\[
\text{trickster design specification:}
\]

- If \( \text{trickster} \) is a secure voting machine, then \( \text{trickster} \) is not a secure voting machine.
- If \( \text{trickster} \) is not a secure voting machine, then \( \text{trickster} \) is a secure voting machine.

\[
\text{bool trickster(string input)} \{
\quad \text{string me = /* source code of trickster */;}
\quad \text{if (isSecure(me))} \{
\quad\quad \text{return true;}
\quad\} \quad \text{else} \{
\quad\quad \text{return countRsIn(input) > countDsIn(input);} 
\quad\}
\}
\]

So we’re done with this part of the design!
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem

We can write programs that use $D$ as a helper function

contradiction!

We can write programs that use $D$ as a helper function

trickster is secure if and only if trickster is not secure.

Putting it all together, take a look at what we accomplished. 
trickster is a secure voting machine if and only if it isn't a secure voting machine!
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

Contradiction!

The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

Contradiction!
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

true

contradiction!

We can write programs that use $D$ as a helper function.

true

Contradiction!

**Theorem:** If $M$ is a secure voting machine, then $M$ is not a secure voting machine. If $M$ is not a secure voting machine, then $M$ is a secure voting machine.

```c
bool isSecure(string function) {
    // source code of trickster
    string me = /* source code of trickster */;
    if (isSecure(me)) {
        return true;
    } else {
        return countRsIn(input) > countDsIn(input);
    }
}
```

We're done! We've shown that starting with the assumption that the secure voting problem is decidable, we reach a contradiction.
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem

We can write programs that use $D$ as a helper function

Contradiction!

```c
bool isSecure(string function) {
    string me = /* source code of trickster */;
    if (isSecure(me)) {
        return true;
    } else {
        return countRsIn(input) > countDsIn(input);
    }
}
```

You might have noticed that this program isn’t the one we used in lecture. But that’s okay!
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

contradiction!

**Theorem:** Yes, $M$ is a secure voting machine. No, $M$ is not a secure voting machine.

**bool** isSecure(string function)

**trickster** design specification:

✓ If trickster is a secure voting machine, then trickster is not a secure voting machine.
✓ If trickster is not a secure voting machine, then trickster is a secure voting machine.

```c++
bool trickster(string input) {
    string me = /* source code of trickster */;
    if (isSecure(me)) {
        return true;
    }
    else {
        return countRsIn(input) > countDsIn(input);
    }
}
```

There can be all sorts of programs that meet the design specification we set out above.
The secure voting problem is decidable.

There is a decider \( D \) for the secure voting problem.

We can write programs that use \( D \) as a helper function.

Contradiction!

The secure voting problem is decidable.

There is a decider \( D \) for the secure voting problem.

We can write programs that use \( D \) as a helper function.

Contradiction!

\[
\text{isSecure}(\text{string function})
\]

\[
\text{trickster} \text{ design specification:}
\]

- If \( \text{trickster} \) is a secure voting machine, then \( \text{trickster} \) is not a secure voting machine.
- If \( \text{trickster} \) is not a secure voting machine, then \( \text{trickster} \) is a secure voting machine.

\[
\text{bool trickster(string input) }
\]

\[
\text{string me = /* source code of trickster */;}
\]

\[
\text{if (isSecure(me)) } \{
\text{ return true;}
\} \text{ else } \{
\text{ return countRsIn(input) > countDsIn(input);}
\}
\]

That's great news for you, because it means that these sorts of proofs aren't about finding a needle in a haystack.
The secure voting problem is decidable.

There is a decider $D$ for the secure voting problem.

We can write programs that use $D$ as a helper function.

Contradiction!

---

**bool** isSecure(string function)

---

**trickster** design specification:

- If **trickster** is a secure voting machine, then **trickster** is not a secure voting machine.
- If **trickster** is not a secure voting machine, then **trickster** is a secure voting machine.

```cpp
bool trickster(string input) {
    string me = /* source code * of trickster */;
    if (isSecure(me)) {
        return true;
    } else {
        return countRsIn(input) > countDsIn(input);
    }
}
```

As long as you meet the design criteria, you should be good to go!
Let's take a minute to review the general process that we followed to get these results to work.
Let's take a minute to review the general process that we followed to get these results to work.

That other guy is going to tell you a general pattern to follow. You might want to take notes.
Let's suppose that you want to prove that some language about TMs is undecidable.
The problem in question is decidable.

Start off by assuming it’s decidable.
The goal is to get a contradiction.

Contradiction!
The problem in question is decidable.

To get there...

Contradiction!
The problem in question is decidable.

There is a decider $D$ for that problem.

...the first step is to suppose that you have a decider for the language in question.

Contradiction!
The problem in question is decidable.

There is a decider $D$ for that problem.

It's often a good idea to draw a picture showing what that decider looks like.
Think about what the inputs to the decider are going to look like. That depends on the language.

The problem in question is decidable.

There is a decider $D$ for that problem.
In the cases we're exploring in this class, there will always be at least one input that's a TM of some sort.

There is a decider $D$ for that problem.

The problem in question is decidable.

Contradiction!
The problem in question is decidable.

There is a decider $D$ for that problem.

Decider $D$ for this problem

$M$

Next, think about what the decider is going to tell you about those inputs. That depends on the problem at hand.

Contradiction!
For example, if your language is the set of TMs that have some property X, then the decider will tell you whether the TM has property X.

Contradiction!

Yes, $M$ has property X.

No, $M$ doesn't have property X.
The problem in question is decidable.

There is a decider $D$ for that problem.

We can write programs that use $D$ as a helper function.

The next step is to think about how to use that decider as a subroutine in some program.

Contradiction!
Think about what the decider would look like as a method in some high-level programming language.
The problem in question is decidable.

There is a decider $D$ for that problem.

We can write programs that use $D$ as a helper function.

You already know what inputs it’s going to take and what it says, so try to come up with a nice, descriptive name for the function.

Contradiction!
In this case, since our decider says whether the program has some property $X$, a good name would be `hasPropertyX`.

The problem in question is decidable.

There is a decider $D$ for that problem.

We can write programs that use $D$ as a helper function.

Contradiction!
The problem in question is decidable.

There is a decider $D$ for that problem.

We can write programs that use $D$ as a helper function.

Contradiction!
The problem in question is decidable.

There is a decider $D$ for that problem.

We can write programs that use $D$ as a helper function.

The next step is to build a self-referential function, which we typically call **trickster**, that gives you some sort of contradiction.

Contradiction!
You're going to want to get a contradiction by building trickster so that it has property X if and only if it doesn't have property X.

The problem in question is decidable.

There is a decider $D$ for that problem.

We can write programs that use $D$ as a helper function.

trickster has property $X$ if and only if it doesn't have property $X$.
The problem in question is decidable.

There is a decider $D$ for that problem.

We can write programs that use $D$ as a helper function.

trickster has property $X$ if and only if it doesn't have property $X$.

Contradiction!

Now, you have to figure out how to write trickster.

```cpp
bool hasPropertyX(string function) {
    // Code to check if function has property X
    // Returns true if hasPropertyX, false otherwise
}
```
We recommend writing out a design specification for the function that you’re going to write.

The problem in question is decidable.

There is a decider $D$ for that problem.

We can write programs that use $D$ as a helper function.

**Contradiction!**

**Decider $D$ for this problem**

- $M$: Decides if $M$ has property $X$.
- $\text{hasPropertyX}$

We recommend writing out a design specification for the function that you’re going to write.

**bool hasPropertyX(string function)**

trickster design specification:

- trickster has property $X$ if and only if it doesn't have property $X$
The problem in question is decidable.

There is a decider $D$ for that problem.

We can write programs that use $D$ as a helper function.

trickster has property $X$ if and only if it doesn't have property $X$

Contradiction!

We have

$M$ \quad \rightarrow \quad \text{Decider } D \text{ for this problem}

Yes, $M$ has property $X$.

No, $M$ doesn't have property $X$.

The function $\text{hasPropertyX}$

```cpp
bool hasPropertyX(string function) {
    //Implementation
}
```

trickster design specification:

- If trickster has property $X$, then trickster does not have property $X$.
- If trickster does not have property $X$, then trickster has property $X$.

You can fill out that spec by reasoning about both directions of the implication.
Finally, you have to go and write a function that gives you a contradiction. 

The problem in question is decidable. 

There is a decider $D$ for that problem. 

We can write programs that use $D$ as a helper function. 

trickster has property $X$ if and only if it doesn't have property $X$. 

Contradiction!

```python
bool hasPropertyX(string function)

trickster design specification:
    If trickster has property $X$, then trickster does not have property $X$.
    If trickster does not have property $X$, then trickster has property $X$.
```

Finally, you have to go and write a function that gives you a contradiction.
The problem in question is decidable.

There is a decider $D$ for that problem.

We can write programs that use $D$ as a helper function.

`trickster` design specification:
- If `trickster` has property $X$, then `trickster` does not have property $X$.
- If `trickster` does not have property $X$, then `trickster` has property $X$.

```cpp
return-type trickster(args) {
  string me = /* source code of trickster */;
  if (hasPropertyX(me)) {
    // do something so trickster doesn't have property X.
  } else {
    // do something so trickster does have property X.
  }
}
```

If you follow the design spec, you'll likely get something like this. Filling in the blanks takes some creativity.
The problem in question is decidable.

There is a decider $D$ for that problem.

We can write programs that use $D$ as a helper function.

Contradiction!

And now you have a contradiction!

```
return-type trickster(args) {
    string me = /* source code */ of trickster */;

    if (hasPropertyX(me)) {
        // do something so trickster doesn't have property X.
    } else {
        // do something so trickster has property X.
    }
}
```
Hope this helps!

Please feel free to ask questions if you have them.
Did you find this useful? If so, let us know! We can go and make more guides like these.