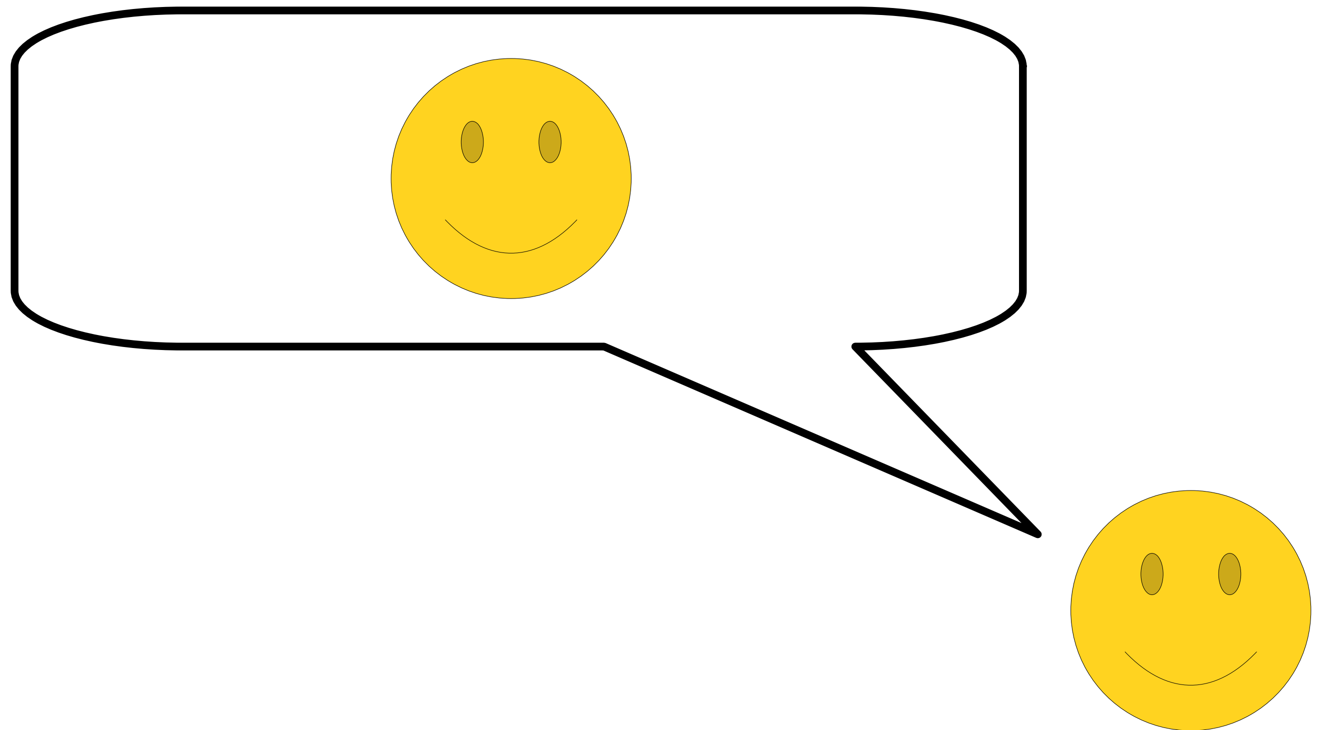
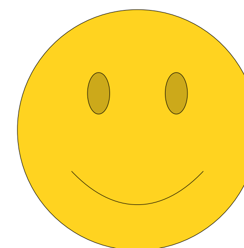
# The Guide to Self-Reference

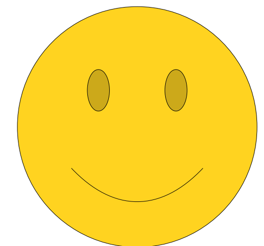


Hi everybody!

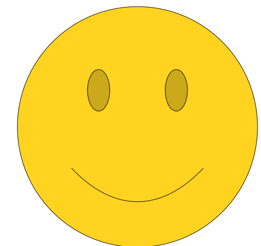


Self-reference proofs can be pretty hard to understand the first time you see them.

If you're confused - that's okay! It's totally normal. This stuff is tricky.



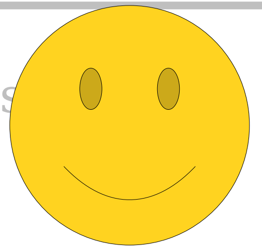
Once you get a better sense for
how to structure these proofs,
I think you'll find that they're not
as bad as they initially seem.



```
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);
}
```

This lecture slide was the first time that we really saw self-reference, and there's a lot to take in here.

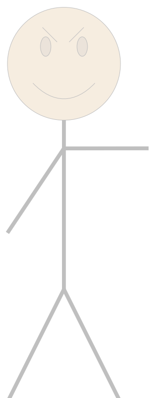


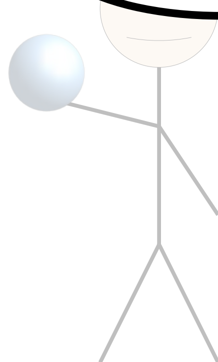
↔

willAccept(me, input) returns true

↔

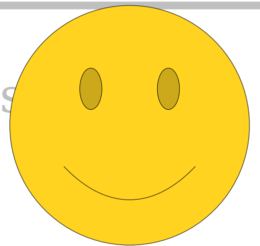
trickster(input) returns false



```
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);
}
```



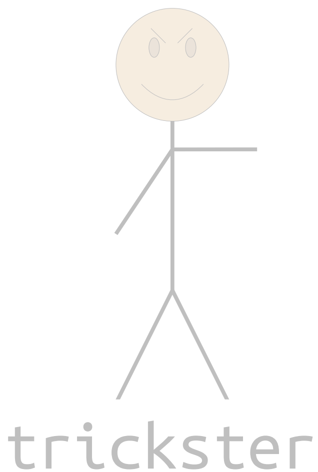

returns

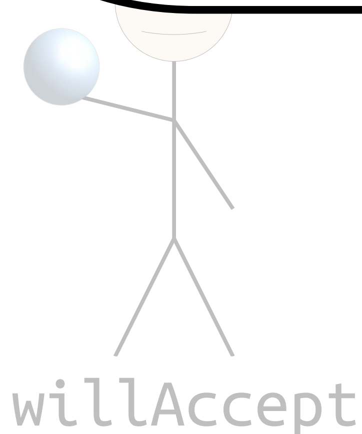
↔

willAccept(me, input) returns true

↔

trickster(input) returns false



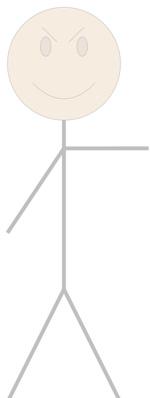



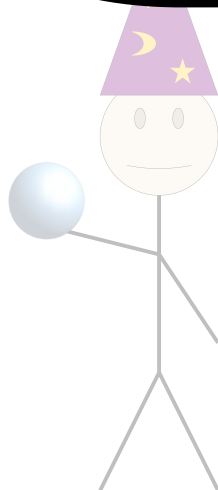
```
bool willAccept(string function, string input) {
    // Returns true if function(input) returns true.
    // Returns false otherwise.
}

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

Let's see where it comes from!

We'll take it from the top.



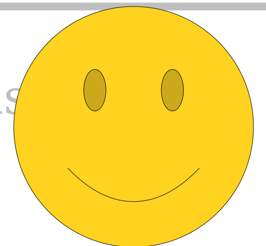

trickster(input) returns true

↔

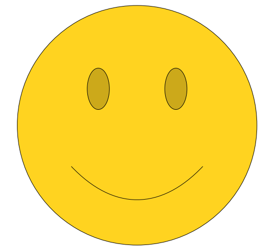
willAccept(me, input) returns true

↔

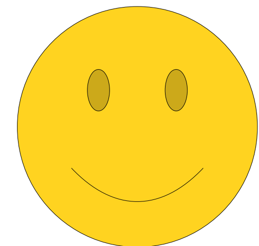
trickster(input) returns false



Let's try to use self-reference to prove that $A_{TM}$ is undecidable.

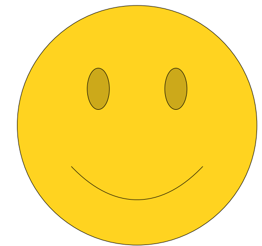


At a high level, we're going to do a proof by contradiction.



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

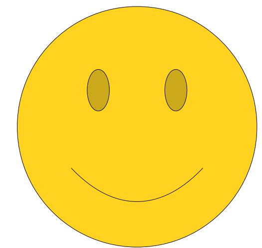
We're going to start off by assuming that $A_{TM}$ is decidable.



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

Contradiction!

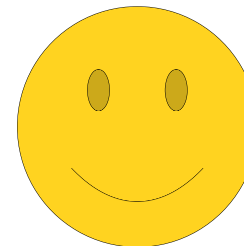
Somehow, we're going to try to use this to get to a contradiction.



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

Contradiction!

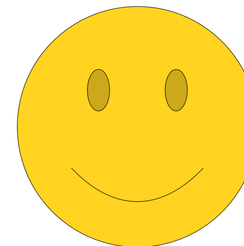
If we can get a contradiction – any contradiction – we'll see that our assumption was wrong.



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

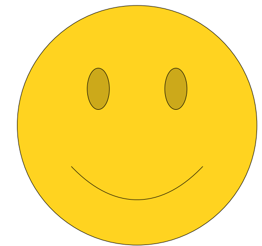
Contradiction!

The challenge is figuring out exactly how to go and do this.



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

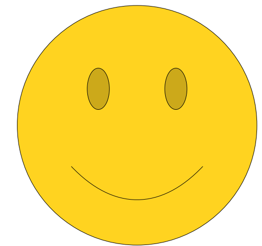
Rather than just jumping all the way to the end, let's see what our initial assumption tells us.



Contradiction!

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

We're assuming that $A_{TM}$ is decidable. What does that mean?

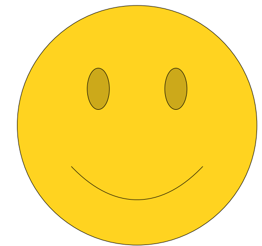


Contradiction!

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

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*.

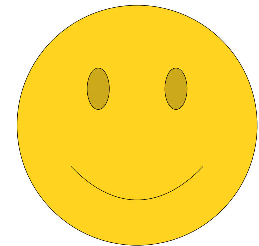


Contradiction!

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

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

What might this decider look like?



Contradiction!

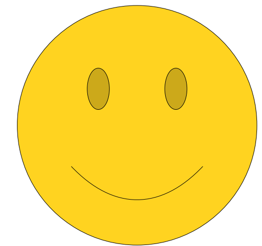
$A_{TM} \in \mathbf{R}$

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

Contradiction!

Decider $D$ for $A_{TM}$

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



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

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

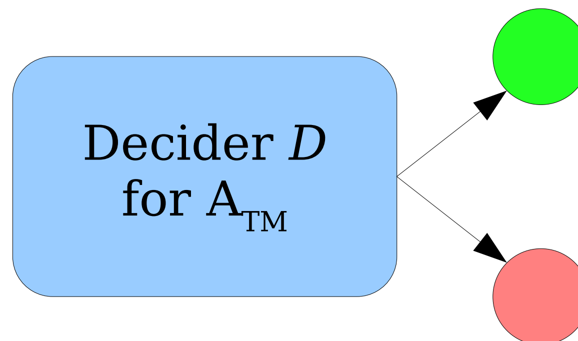
Contradiction!

Decider $D$ for $A_{TM}$

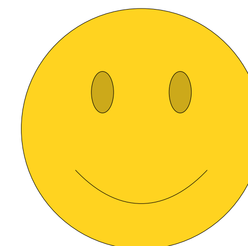
First, it has to always halt.

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

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



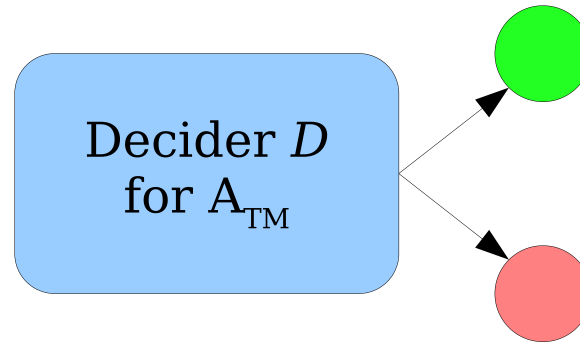

That means that if you give it any input, it has to either accept or reject it. We'll visualize this with these two possible outputs.



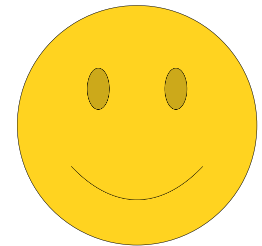
Contradiction!

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

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




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

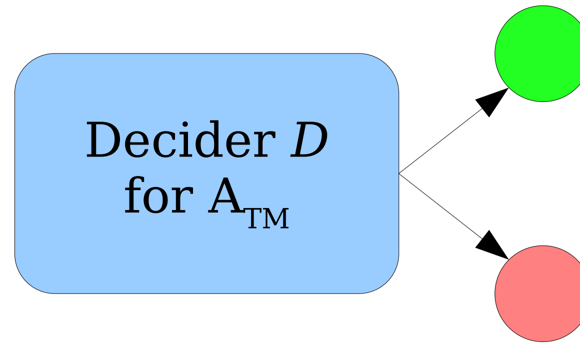


Contradiction!

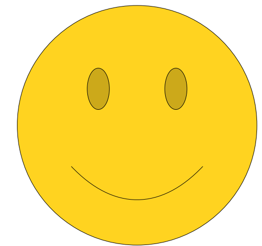
$A_{TM} \in \mathbf{R}$

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

Contradiction!

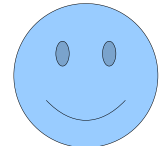



Next, the decider has to tell us something about $A_{TM}$.



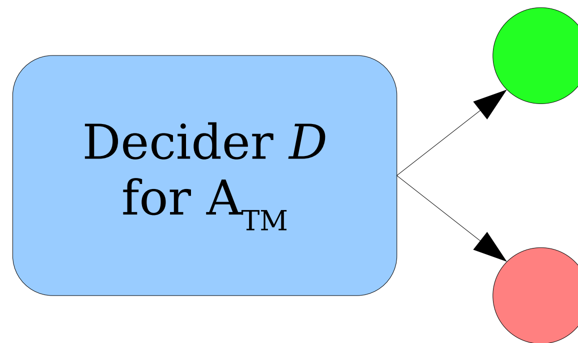
As a reminder, $A_{TM}$ is the language

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

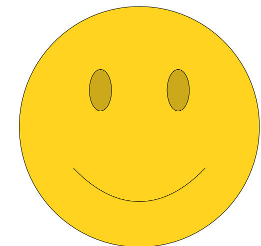


$A_{TM} \in \mathbf{R}$

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

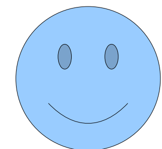



Specifically, the decider $D$ needs to take in an input and tell us whether that input is in $A_{TM}$.



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

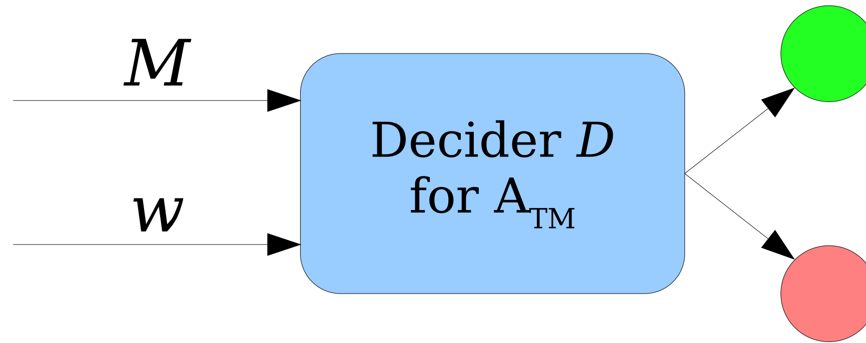
$\{ \langle M, w \rangle \mid M$ is a TM and $M$ accepts $w \}$



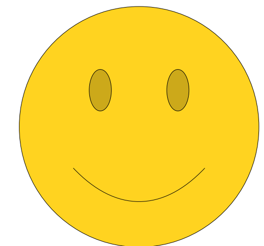
Contradiction!

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

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

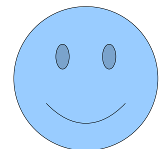



$A_{TM}$ is a language of pairs of TMs and strings, so $D$ will take in two inputs, a machine $M$ and a string $w$.



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

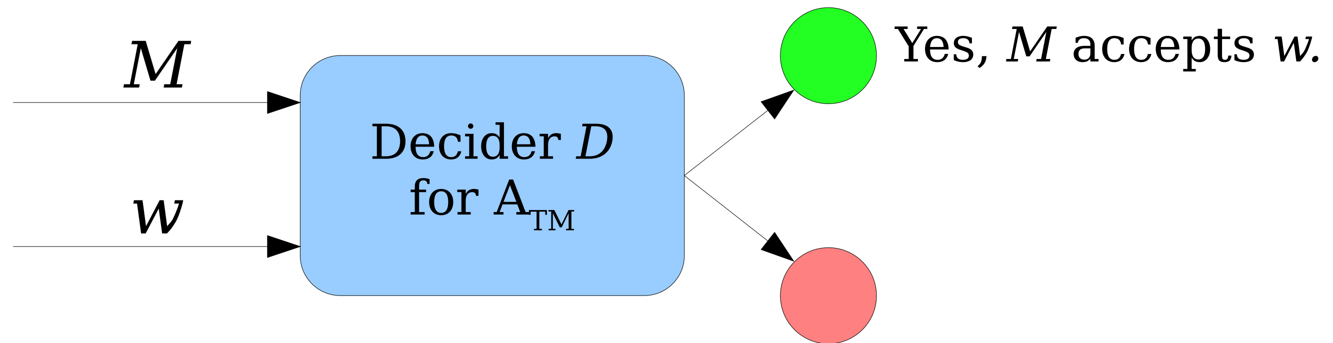
$\{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$



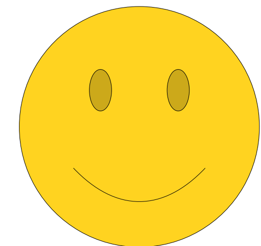
Contradiction!

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

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

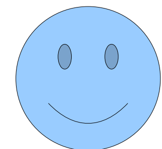



If $D$ accepts its input ⟨M, w⟩, it means that ⟨M, w⟩ ∈ $A_{TM}$, and so $M$ accepts $w$.



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

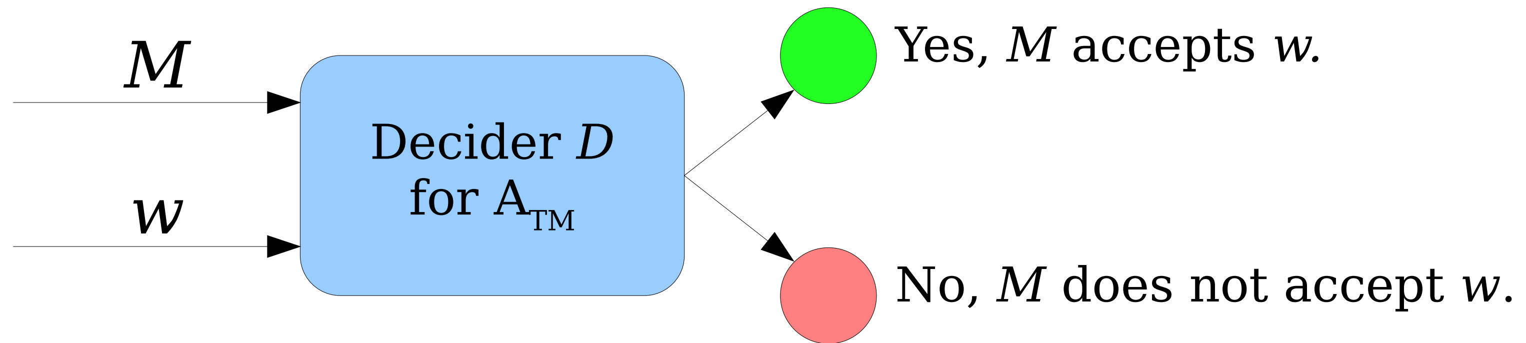
{ ⟨$M$, $w$⟩ | $M$ is a TM and $M$ accepts $w$ }



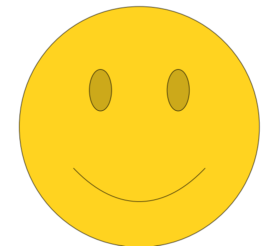
Contradiction!

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

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

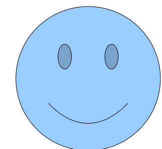



Otherwise, if $D$ rejects its input, it means that $M$ doesn't accept $w$.



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

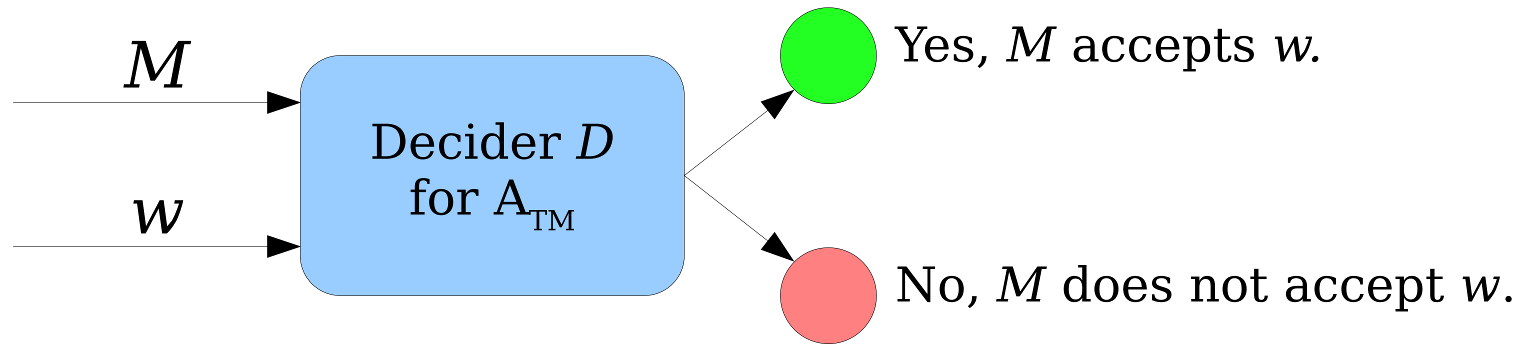
$\{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$



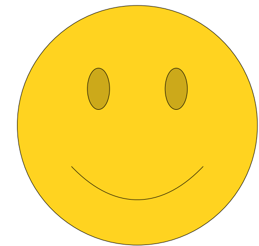
Contradiction!

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

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




So now we've got this TM D lying around. What can we do with it?



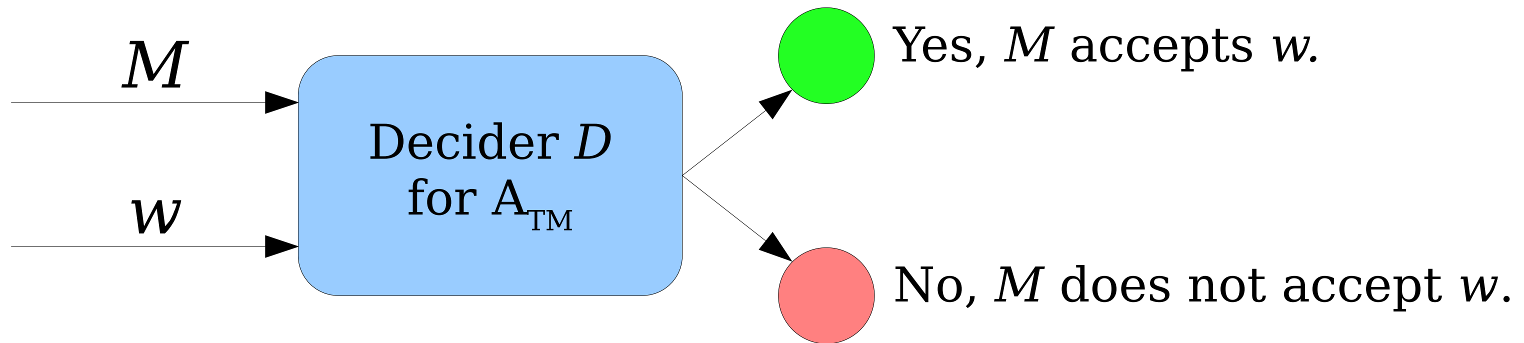
Contradiction!

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

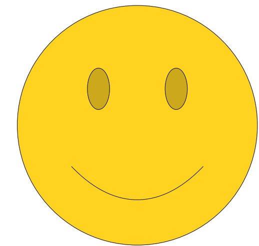
There is a decider $D$ for $A_{TM}$

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

Contradiction!




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

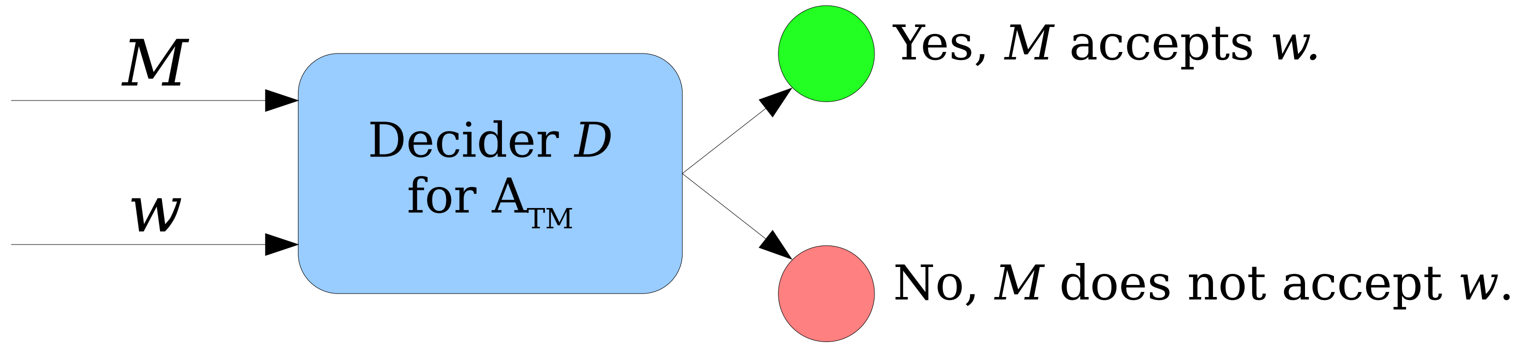


$A_{TM} \in \mathbf{R}$

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

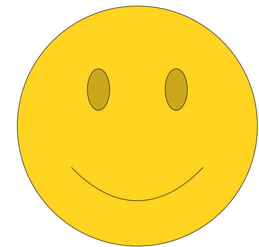
We can write programs that use $D$ as a helper function

Contradiction!




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

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

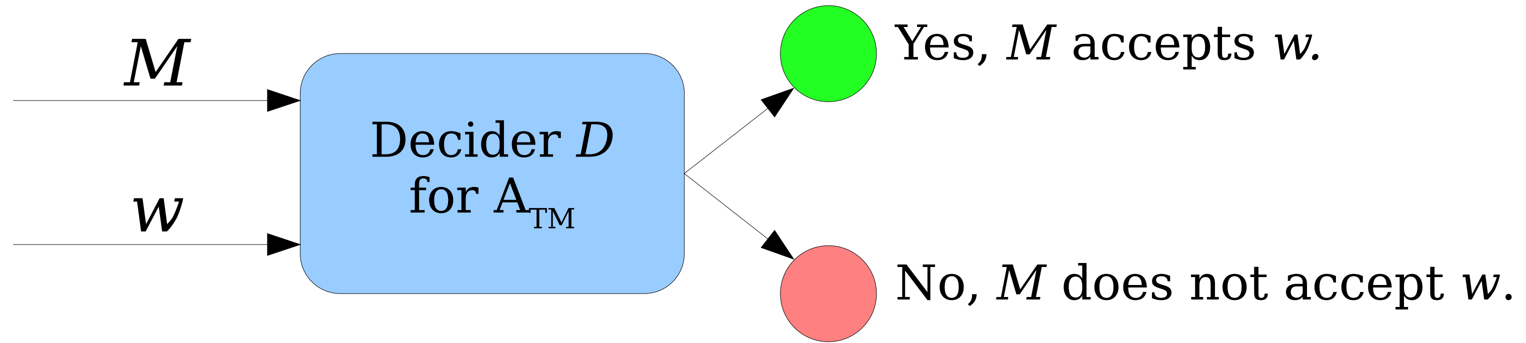


$A_{TM} \in \mathbf{R}$

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

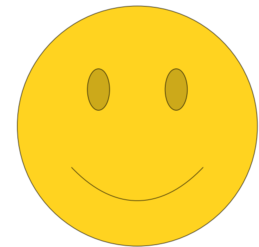
We can write programs that use $D$ as a helper function

Contradiction!




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

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

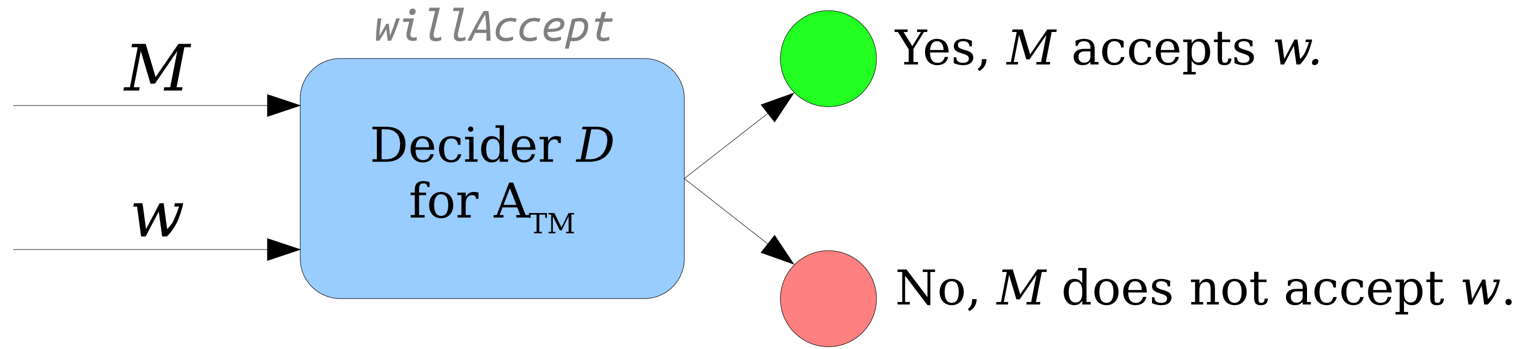


$A_{TM} \in \mathbf{R}$

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

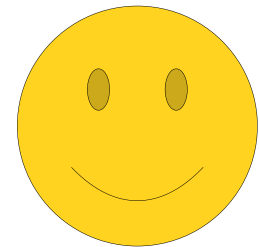
We can write programs that use $D$ as a helper function

Contradiction!




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

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

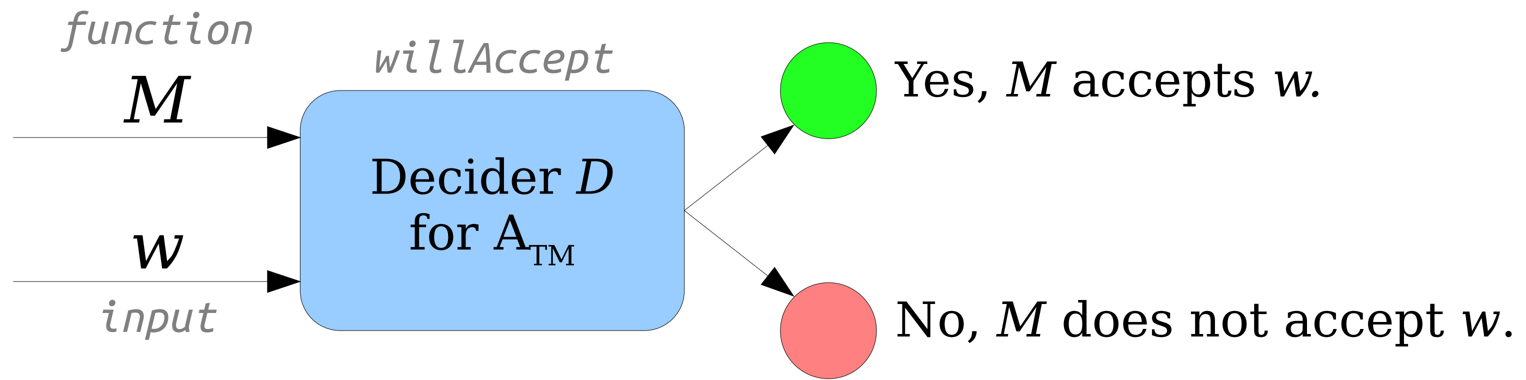


$A_{TM} \in \mathbf{R}$

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

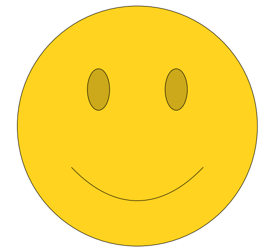
We can write programs that use $D$ as a helper function

Contradiction!




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

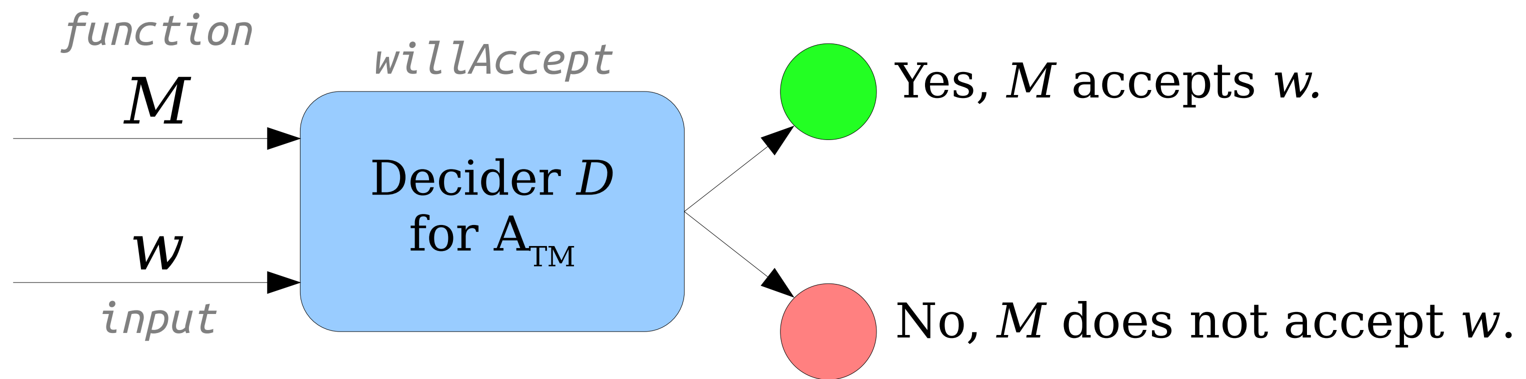
The two arguments to willAccept then correspond to the inputs to the decider $D$.



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

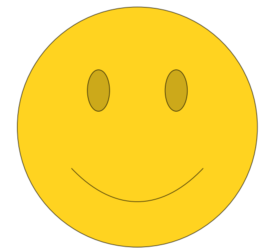
There is a decider *D* for $A_{TM}$

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




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

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



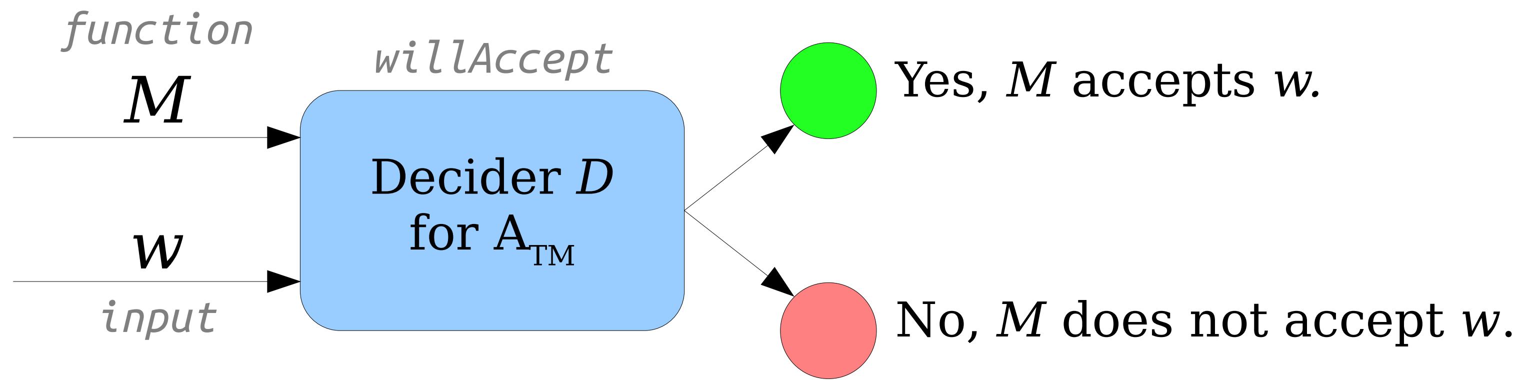
Contradiction!

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

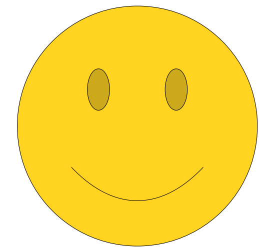
There is a decider $D$ for $A_{TM}$

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

Contradiction!




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

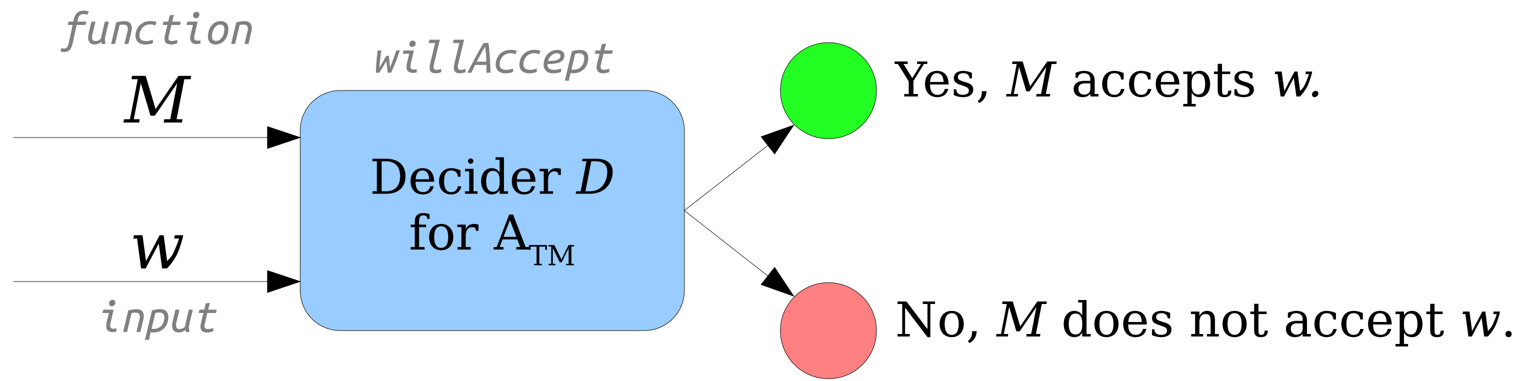


$A_{TM} \in \mathbf{R}$

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

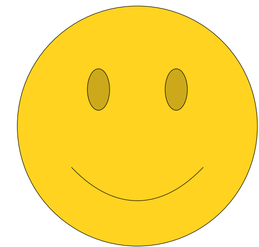
We can write programs that use *D* as a helper function

Contradiction!




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

Ultimately, we're trying to get a contradiction.



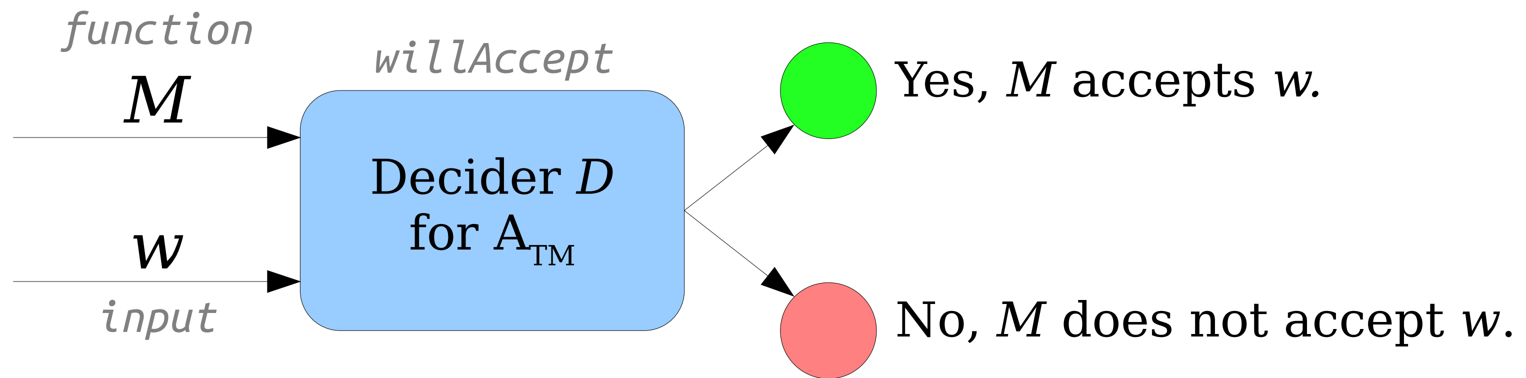
$A_{TM} \in \mathbf{R}$

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

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

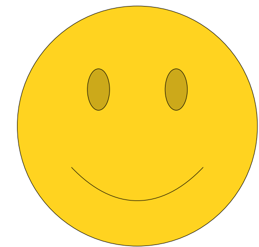
`trickster` accepts its input if and only if `trickster` does not accept its input

Contradiction!




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

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!



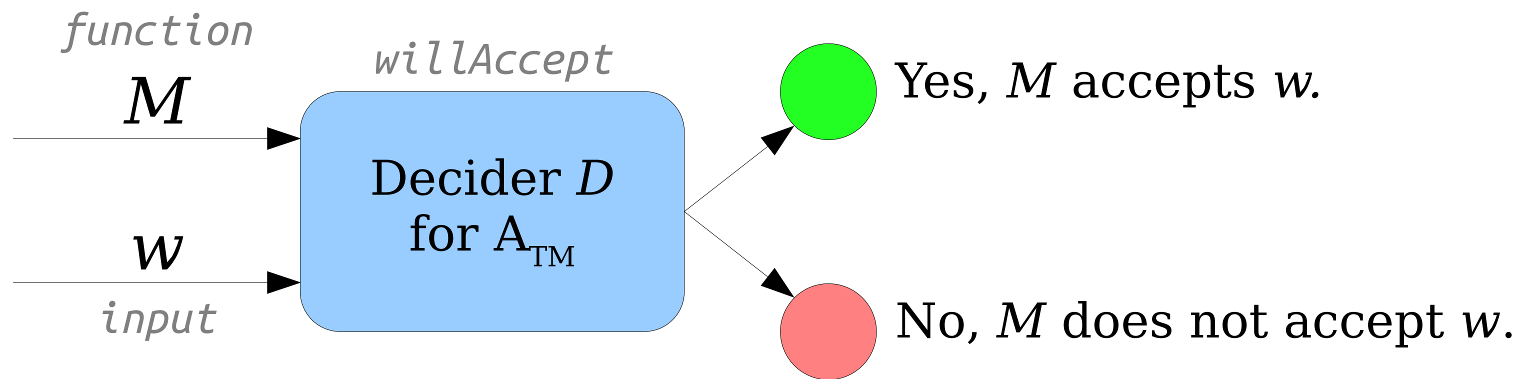
$A_{TM} \in \mathbf{R}$

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

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

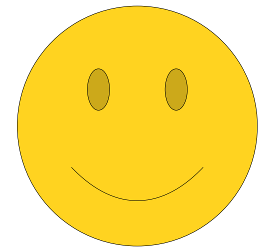
`trickster` accepts its input if and only if `trickster` does not accept its input

Contradiction!




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

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.



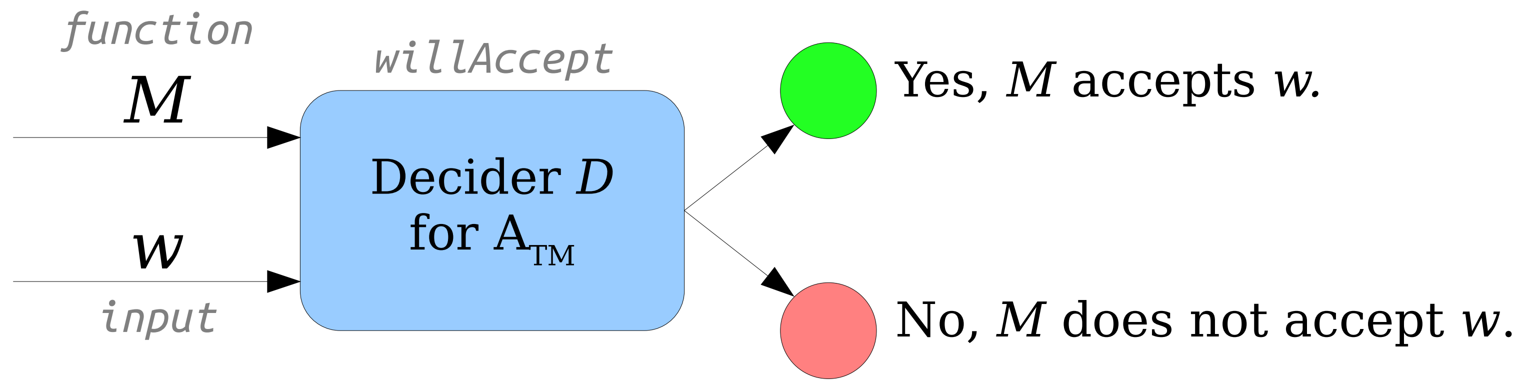
$A_{TM} \in \mathbf{R}$

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

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

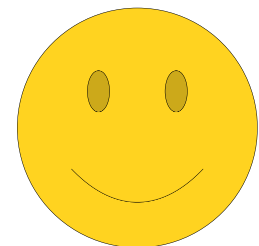
`trickster` accepts its input if and only if `trickster` does not accept its input

Contradiction!




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

Now, we haven't actually written this `trickster` function yet. That's the next step.



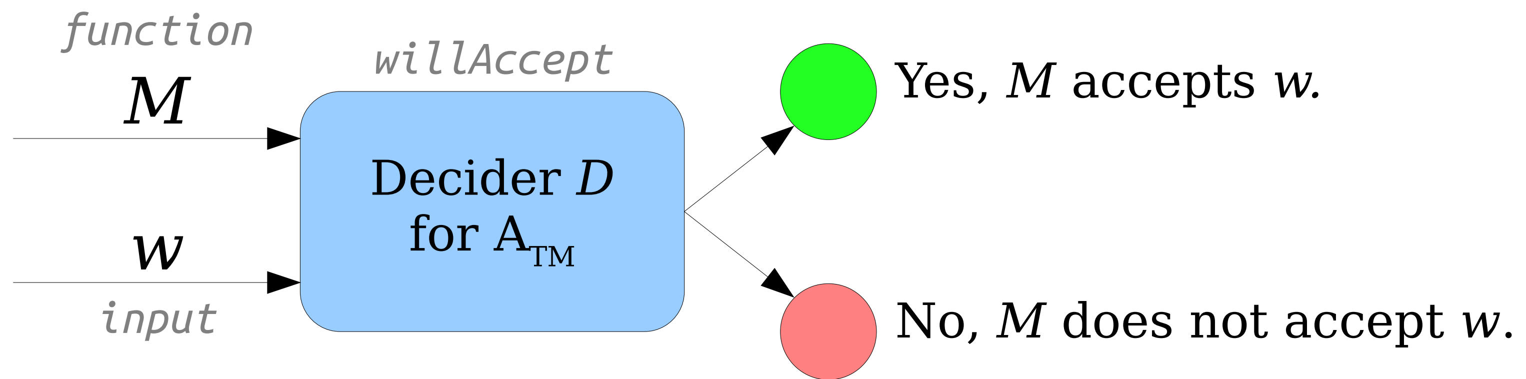
$A_{TM} \in \mathbf{R}$

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

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

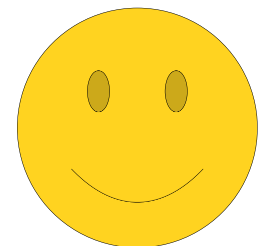
`trickster` accepts its input if and only if `trickster` does not accept its input

Contradiction!




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

If you look at what we've said, right now we have a goal of what `trickster` <u>should</u> do, not how `trickster` actually does that.



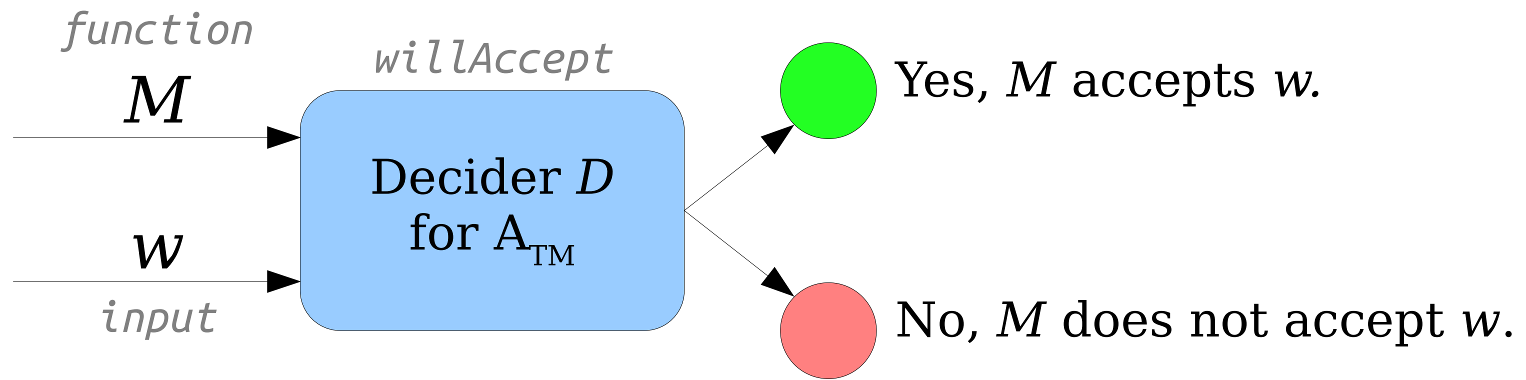
$A_{TM} \in \mathbf{R}$

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

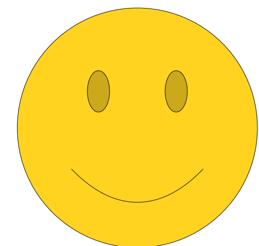
We can write programs that use $D$ as a helper function

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

Contradiction!




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



You can think of this requirement as a sort of "design specification."

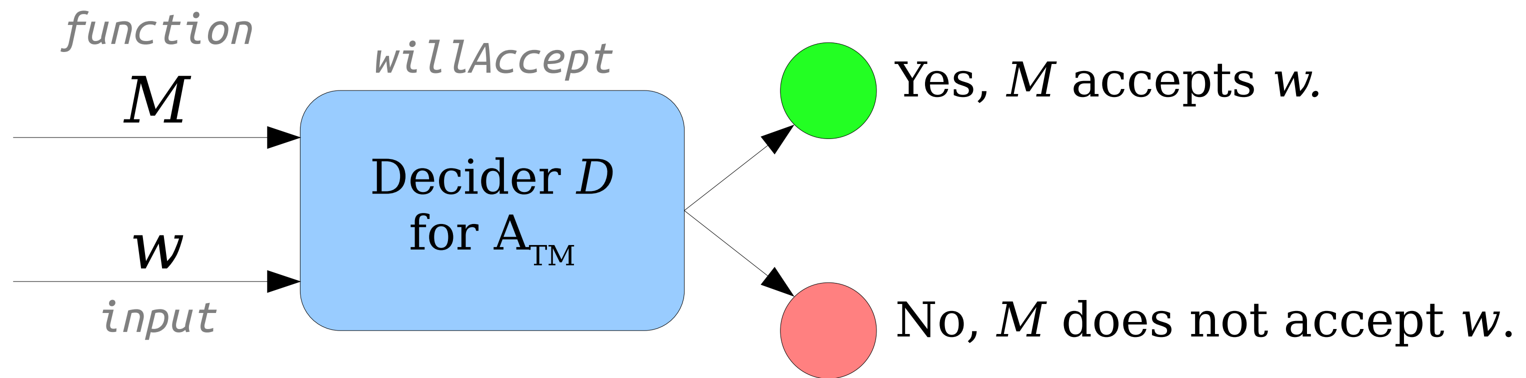
$A_{TM} \in \mathbf{R}$

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

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

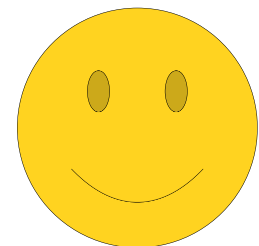
`trickster` accepts its input if and only if `trickster` does not accept its input

Contradiction!




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

`trickster` design specification:



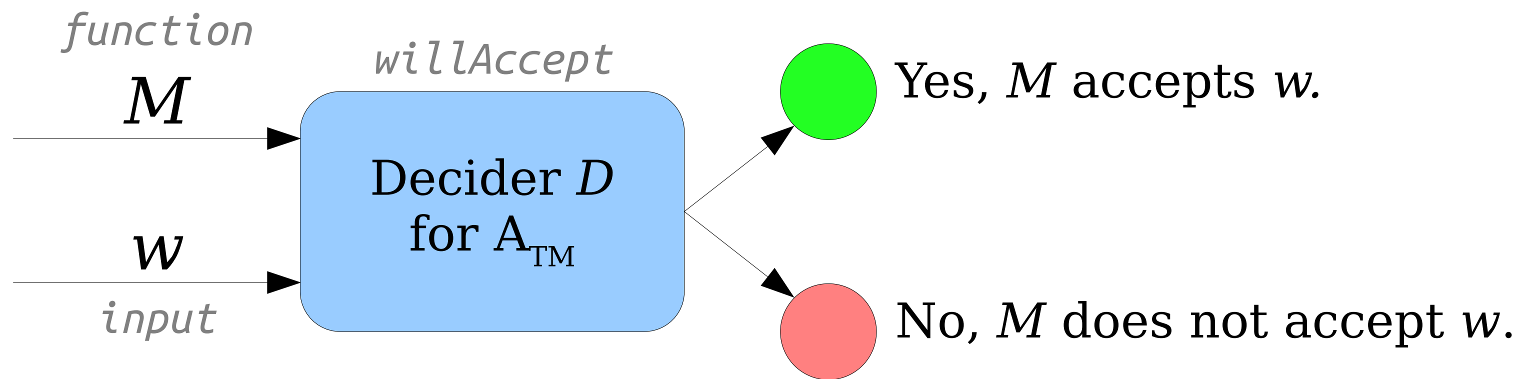
$A_{TM} \in \mathbf{R}$

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

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

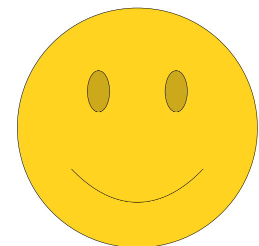
`trickster` accepts its input if and only if `trickster` does not accept its input

Contradiction!




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

`trickster` design specification:



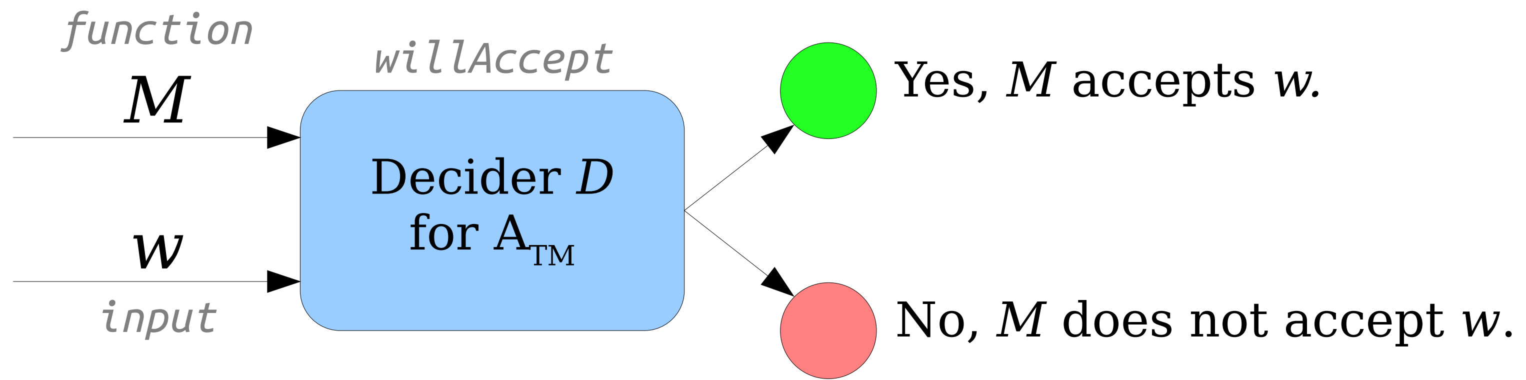
$A_{TM} \in \mathbf{R}$

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.

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

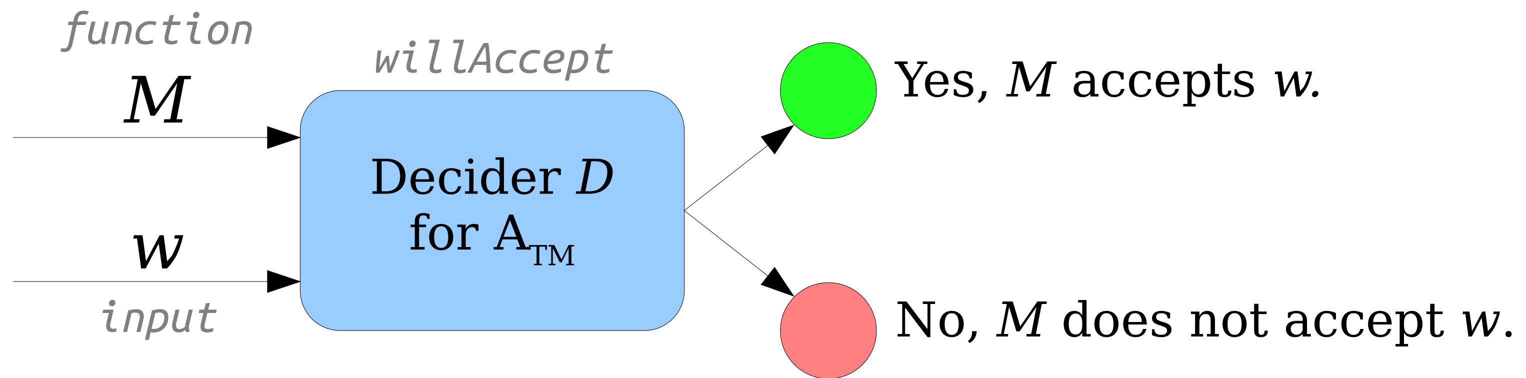
$A_{TM} \in \mathbf{R}$

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!

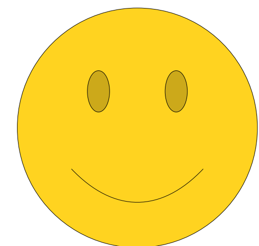



```
bool willAccept(string function, string input)
```

`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.

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



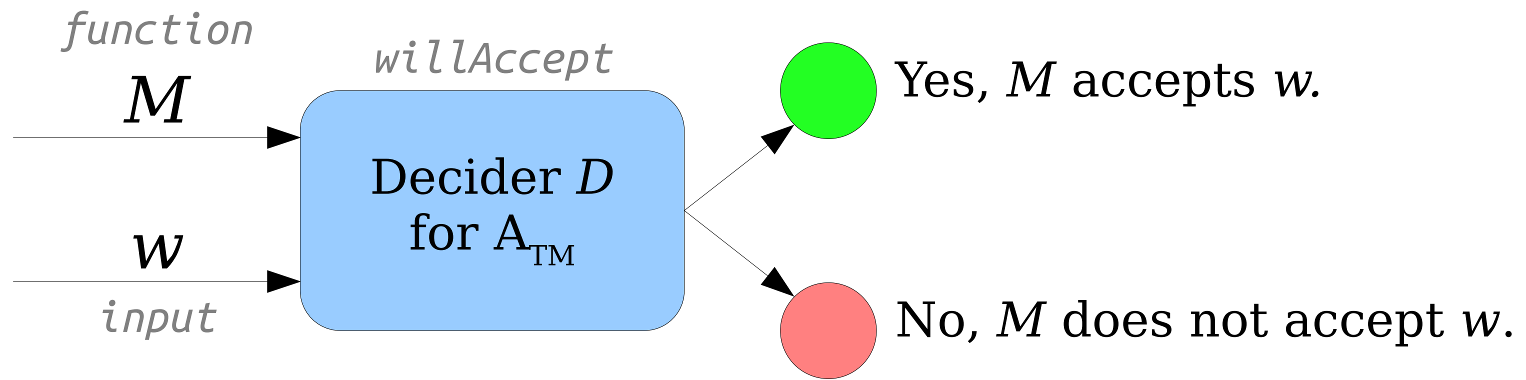
$A_{TM} \in \mathbf{R}$

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!

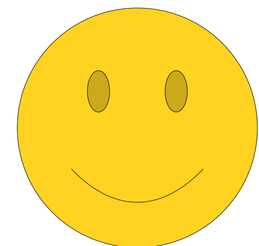



```
bool willAccept(string function, string input)
```

`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.

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



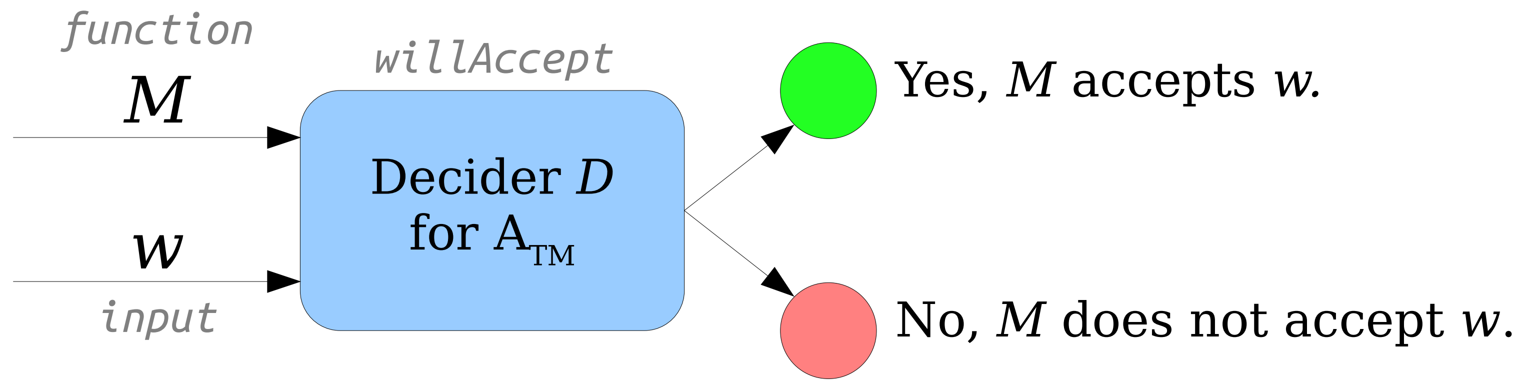
$A_{TM} \in \mathbf{R}$

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.

We'll write it in the space over to the left.

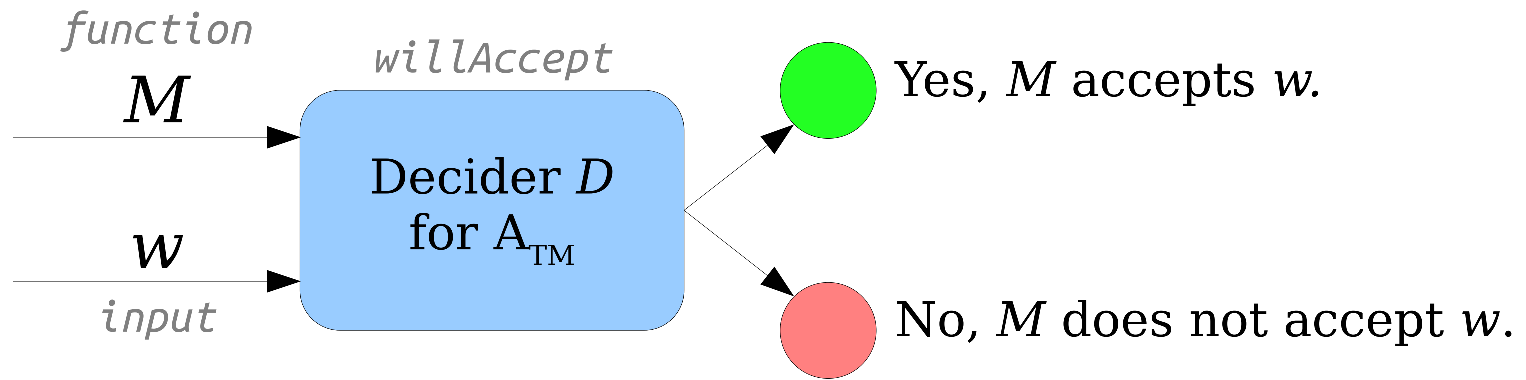
$A_{TM} \in \mathbf{R}$

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) {

}
```

This function will take in a single input, then return a boolean.

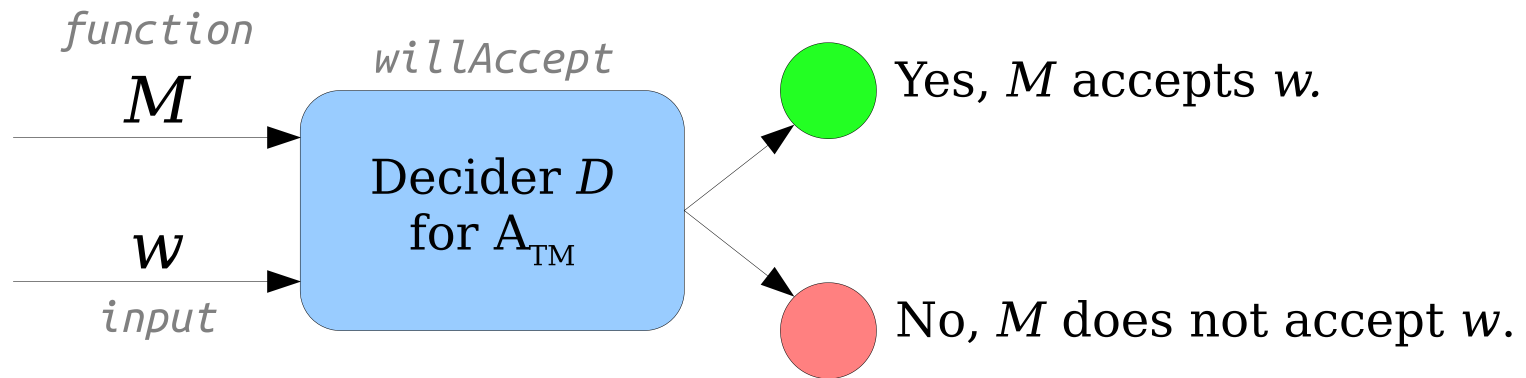
$A_{TM} \in \mathbf{R}$

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) {

}
```

Now, we somehow need to meet the design spec given above.

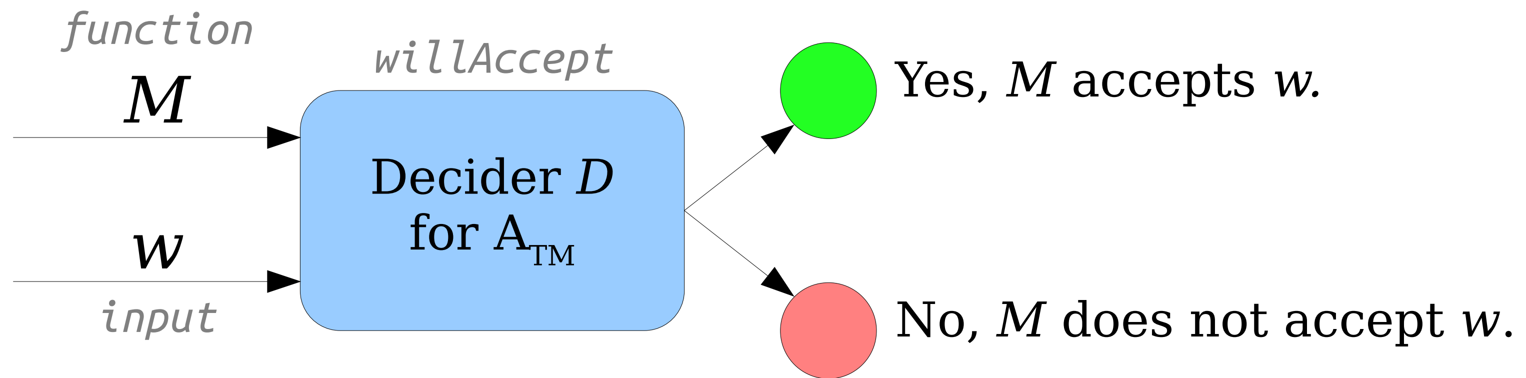
$A_{TM} \in \mathbf{R}$

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) {

}
```

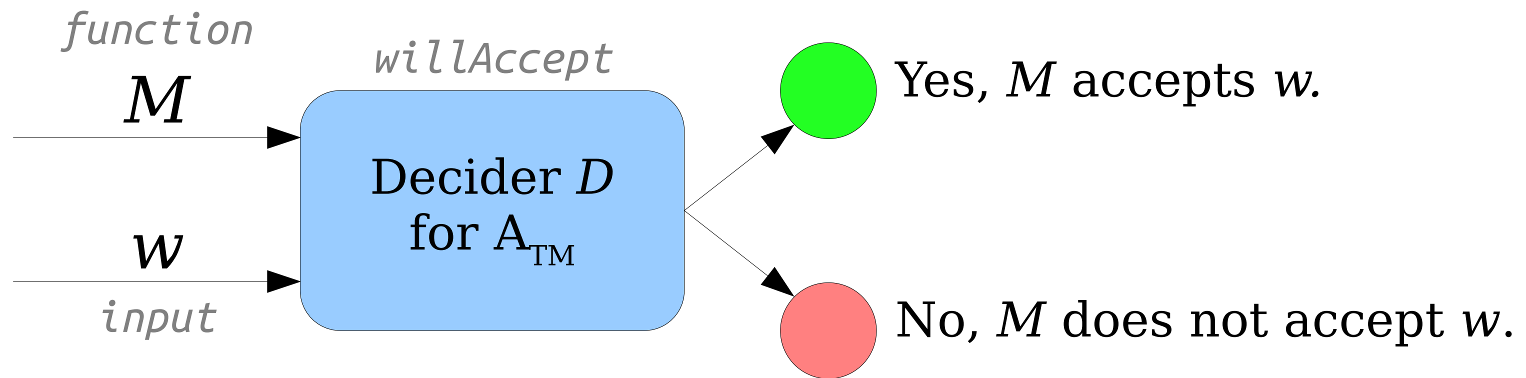
$A_{TM} \in \mathbf{R}$

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) {


}
```

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

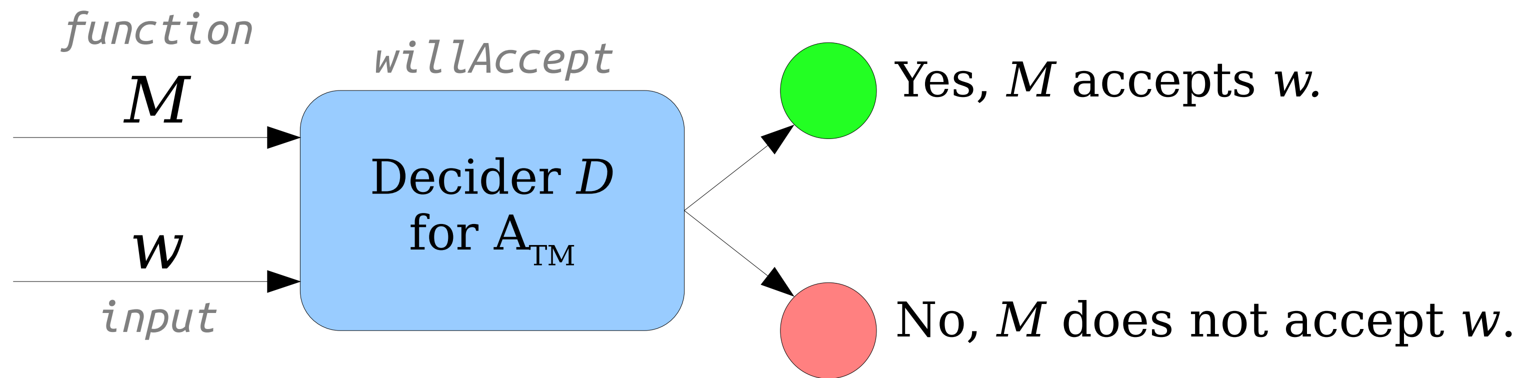
$A_{TM} \in \mathbf{R}$

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) {

}
```

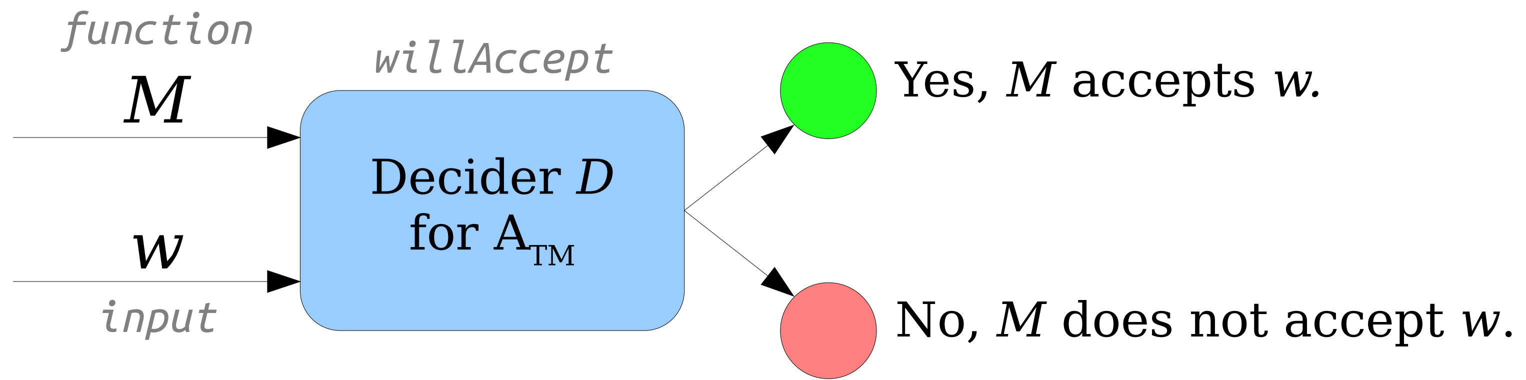
$A_{TM} \in \mathbf{R}$

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) {

}
```

Crazy as it seems, that's something we can actually do!

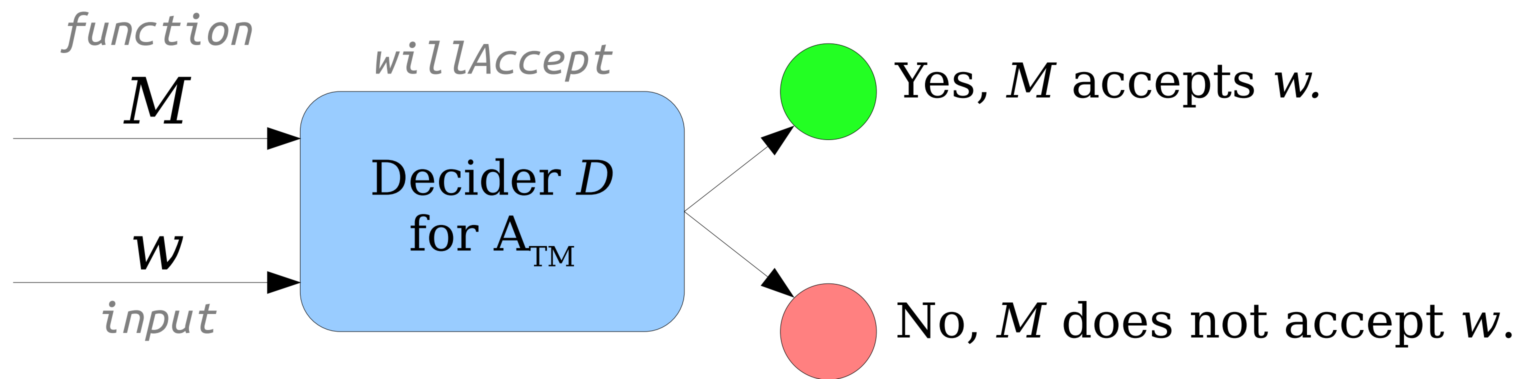
$A_{TM} \in \mathbf{R}$

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
               */;

}
```

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

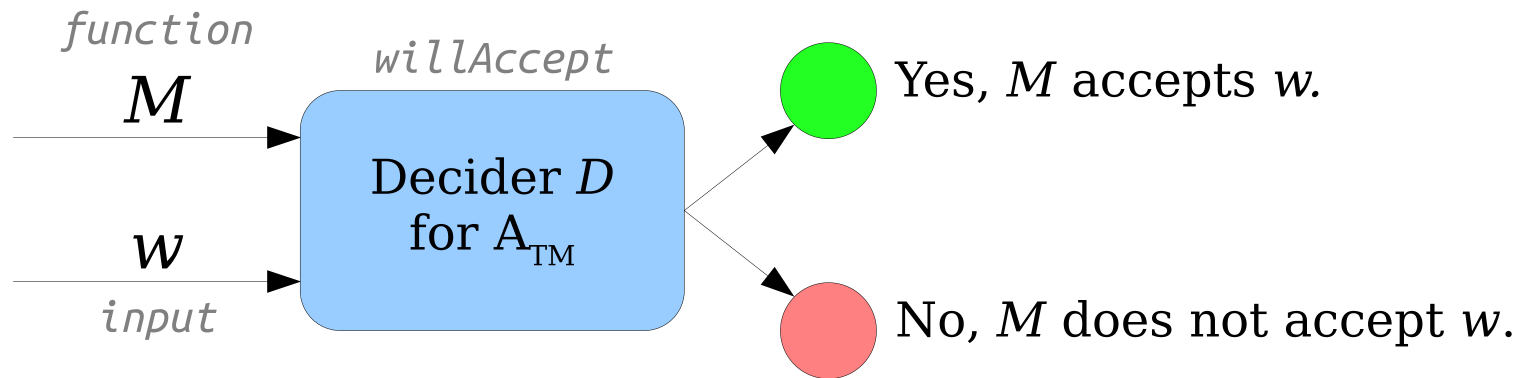
$A_{TM} \in \mathbf{R}$

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)) {

  } else {

  }
}
```

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

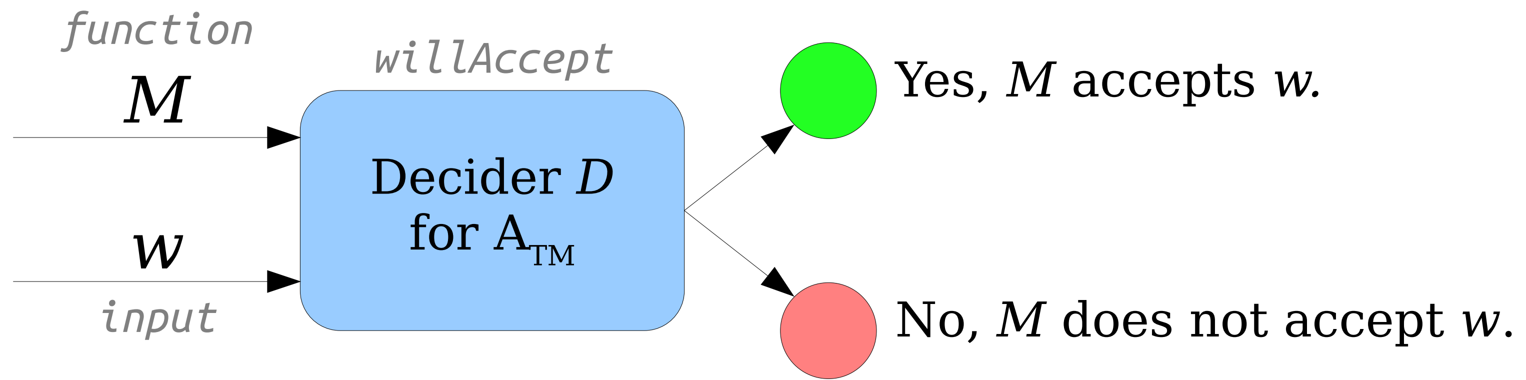
$A_{TM} \in \mathbf{R}$

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)) {

  } else {

  }
}
```

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

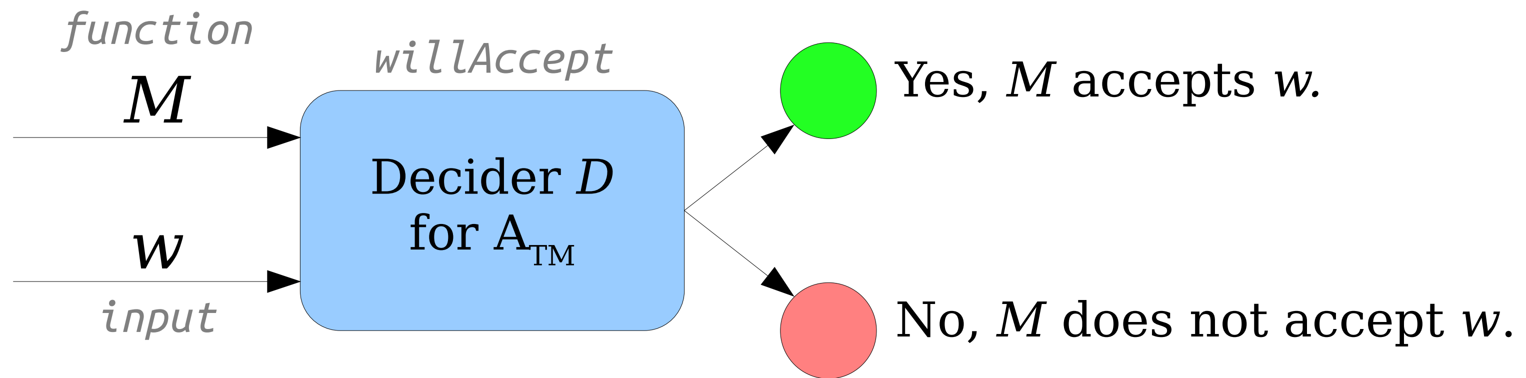
$A_{TM} \in \mathbf{R}$

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)) {

  } else {

  }
}
```

Our specification says that, if `trickster` is supposed to accept its input, then it needs to not accept its input.

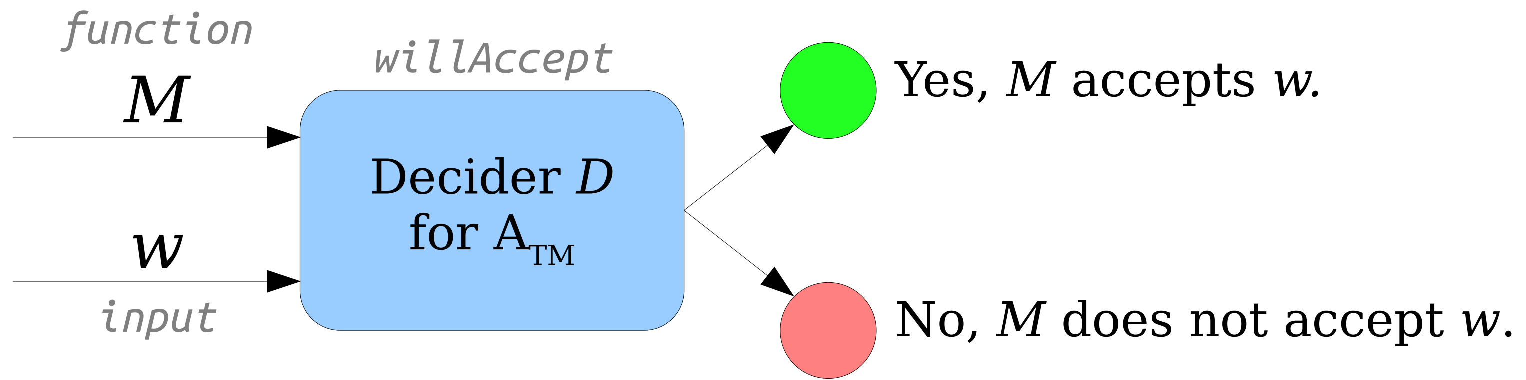
$A_{TM} \in \mathbf{R}$

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)) {

  } else {

  }
}
```

What's something we can do to not accept our input?

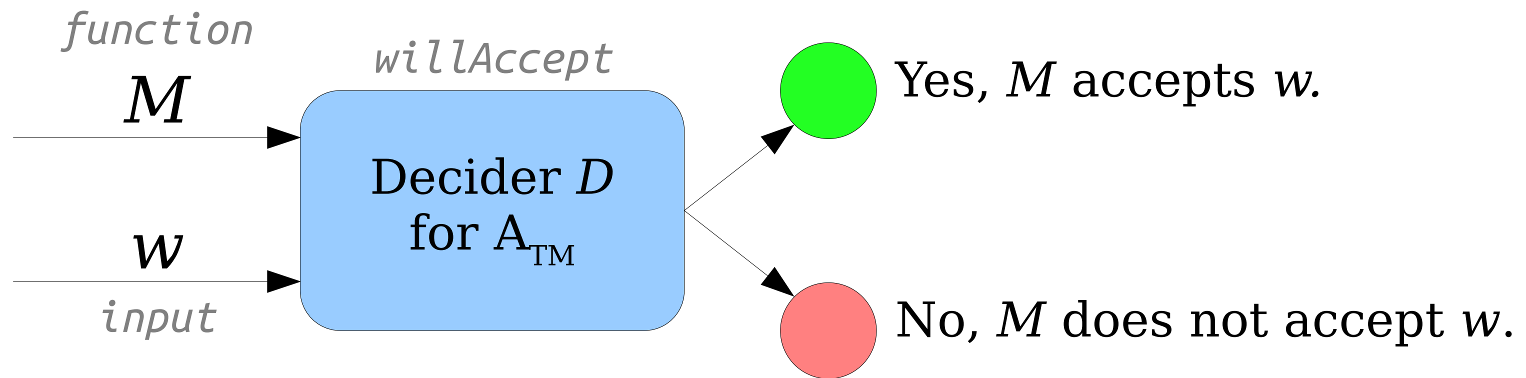
$A_{TM} \in \mathbf{R}$

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 {

  }
}
```

There's a couple of options here, actually. One of them is to just go and reject!

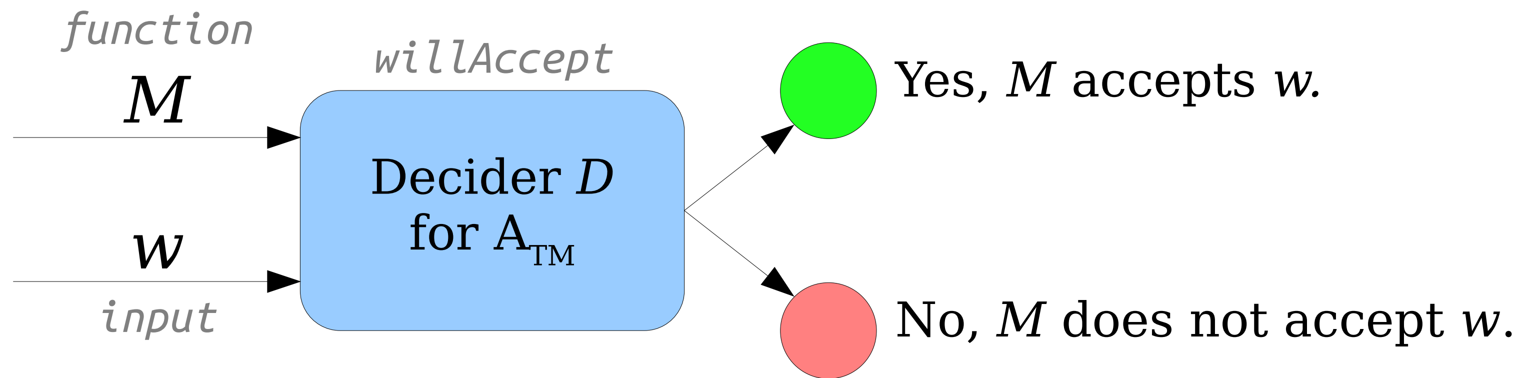
$A_{TM} \in \mathbf{R}$

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 {

  }
}
```

So we've taken care of that part of the design.

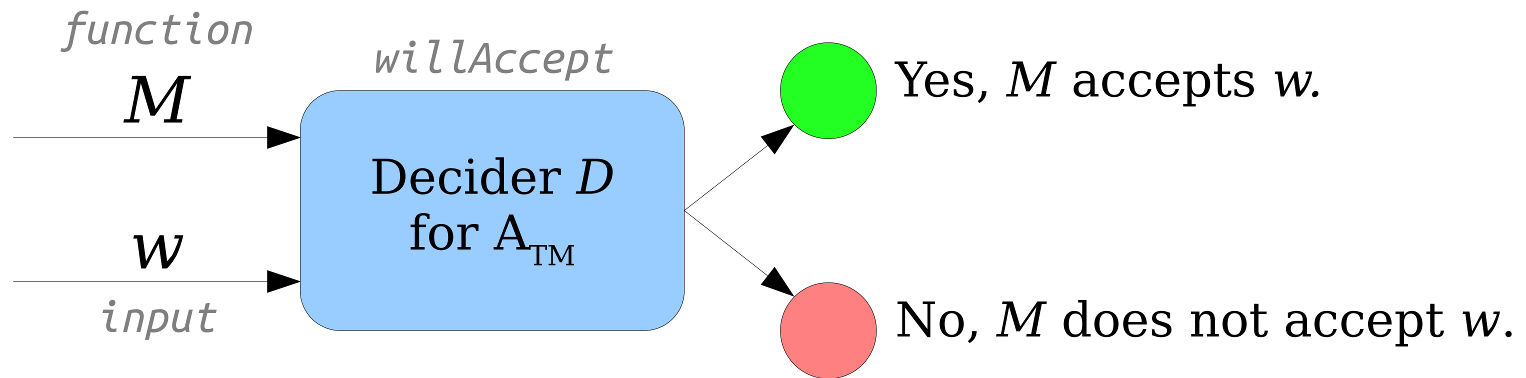
$A_{TM} \in \mathbf{R}$

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 {

  }
}
```

What about this part?

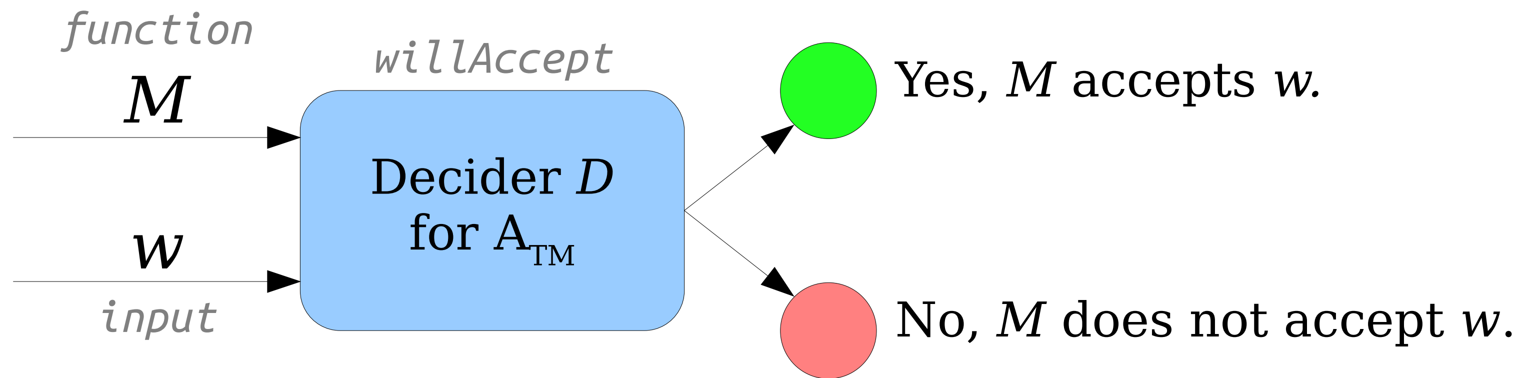
$A_{TM} \in \mathbf{R}$

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 {

  }
}
```

This says that if we aren't supposed to accept the input, then we should accept the input.

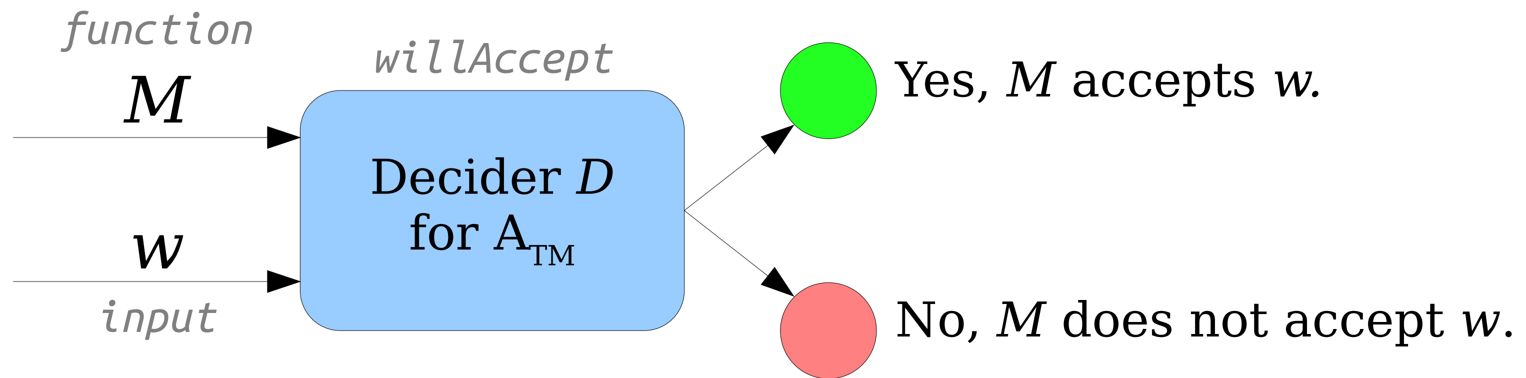
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

So let's go add this line to our program.

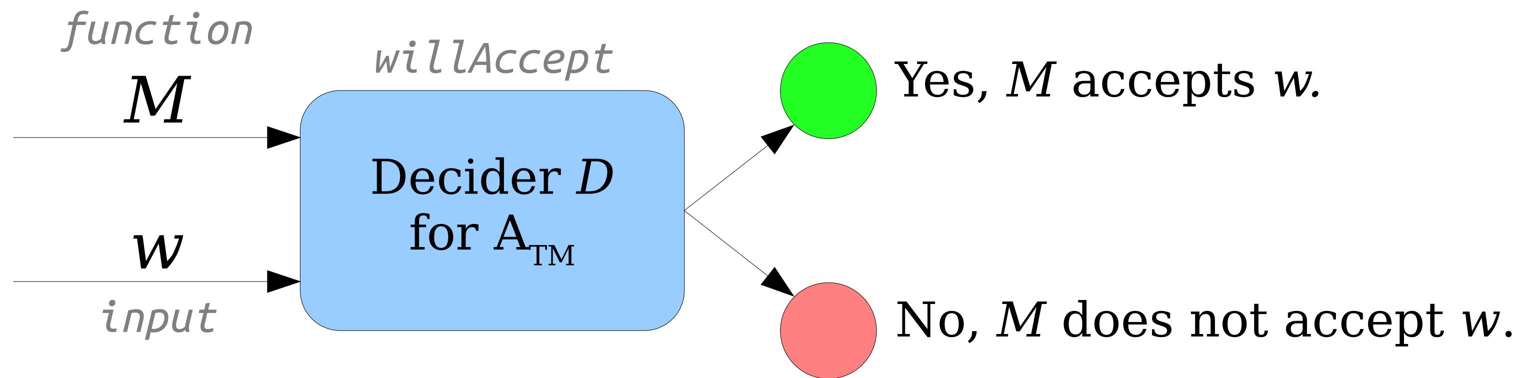
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

And hey! We're done with this part of the design spec.

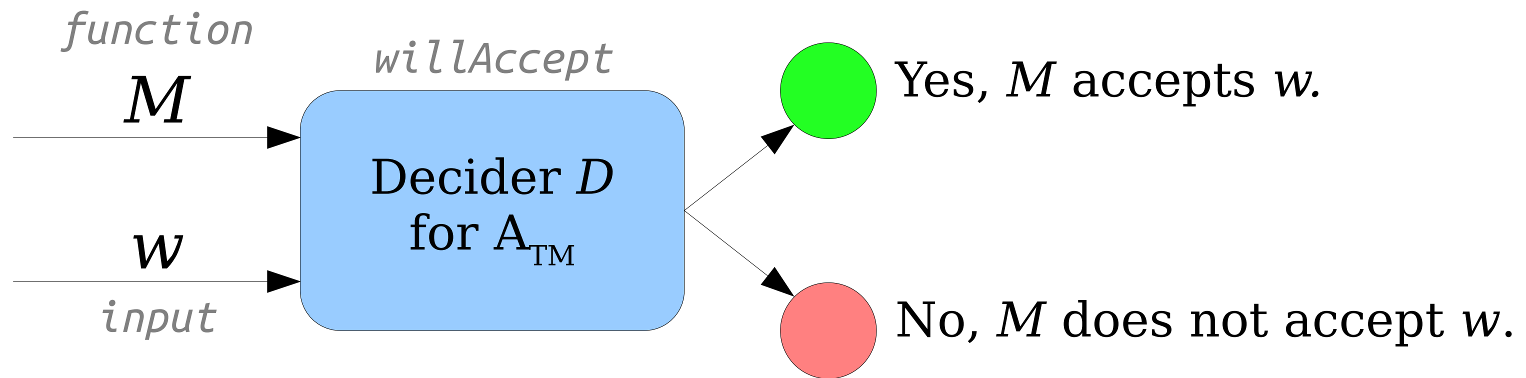
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

Let's take a quick look over our `trickster` function.

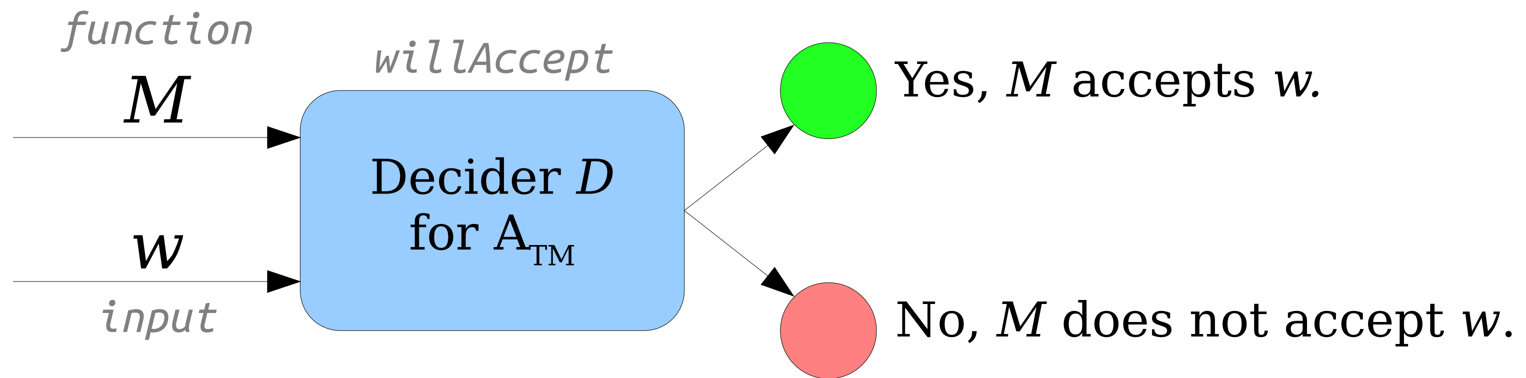
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

This is what we said `trickster` was supposed to do.
And hey! That's what it does.

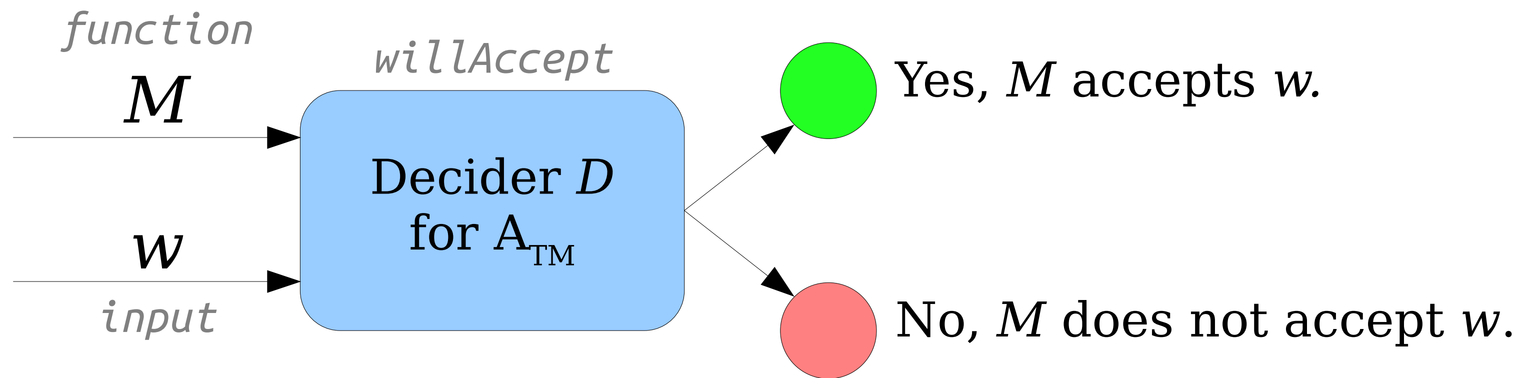
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

The whole point of this exercise was to get a contradiction.

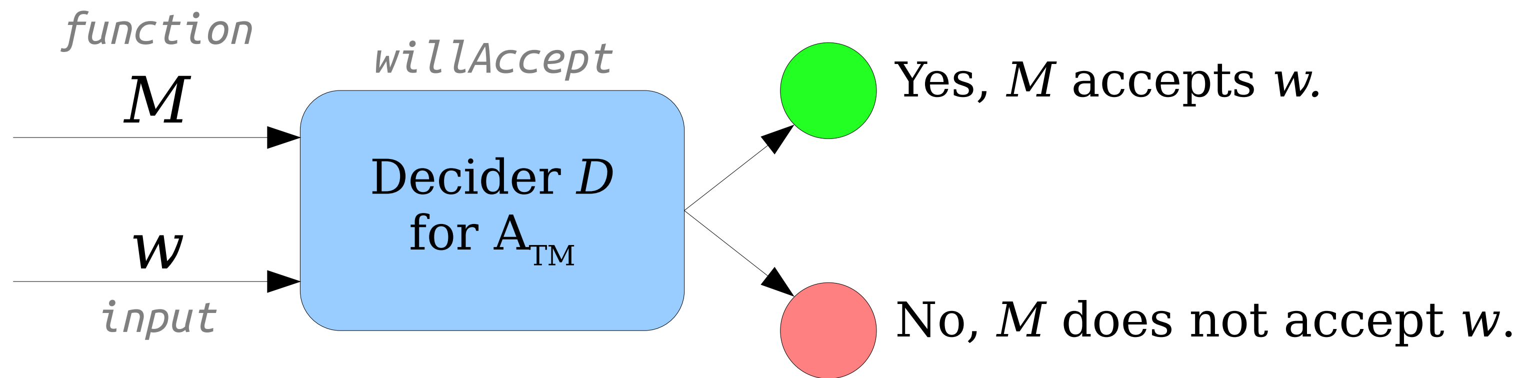
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

And, indeed, that's what we've done! `trickster` accepts if and only if it doesn't accept.

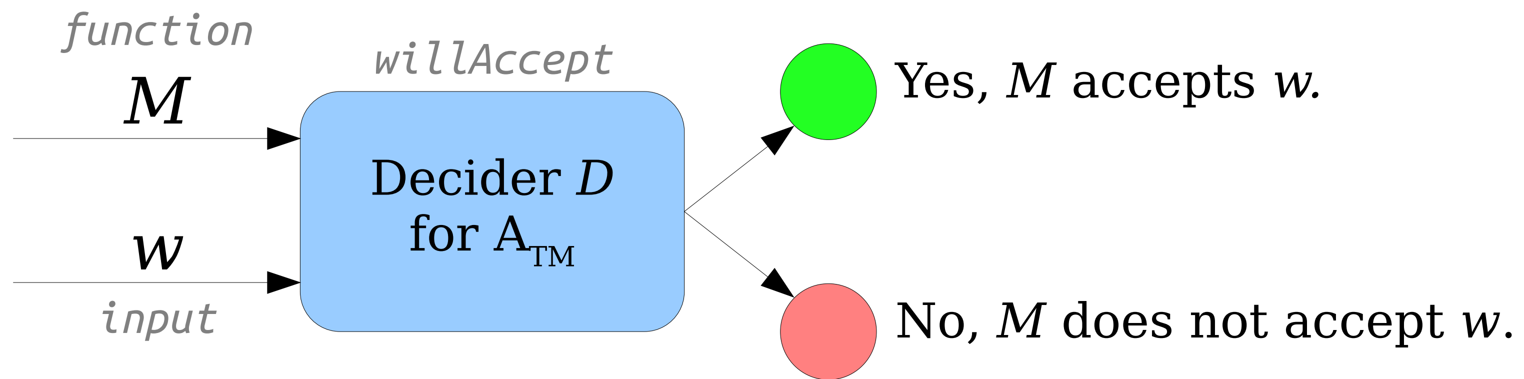
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

So if you trace through the implications here...

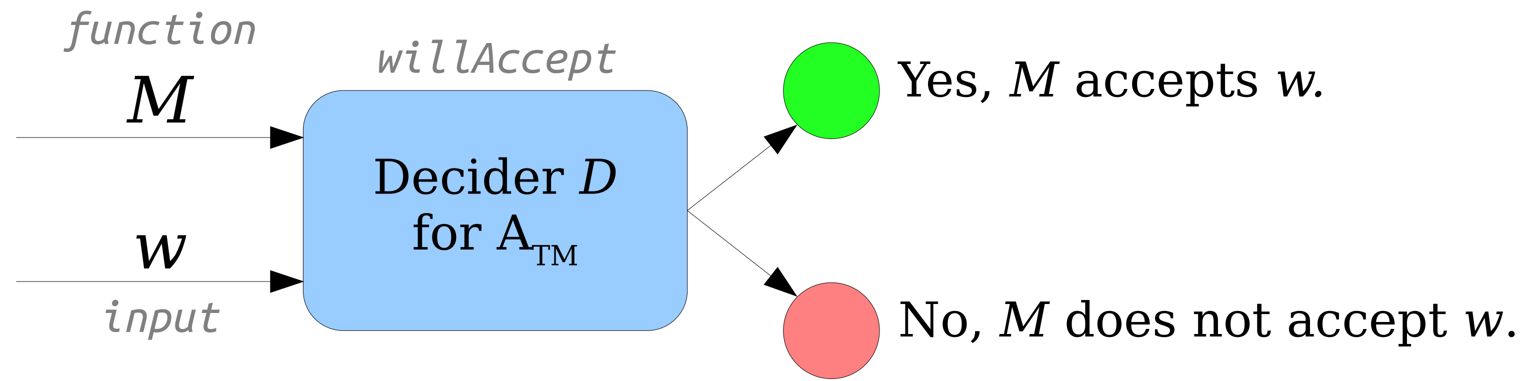
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

So if you trace through the implications here…

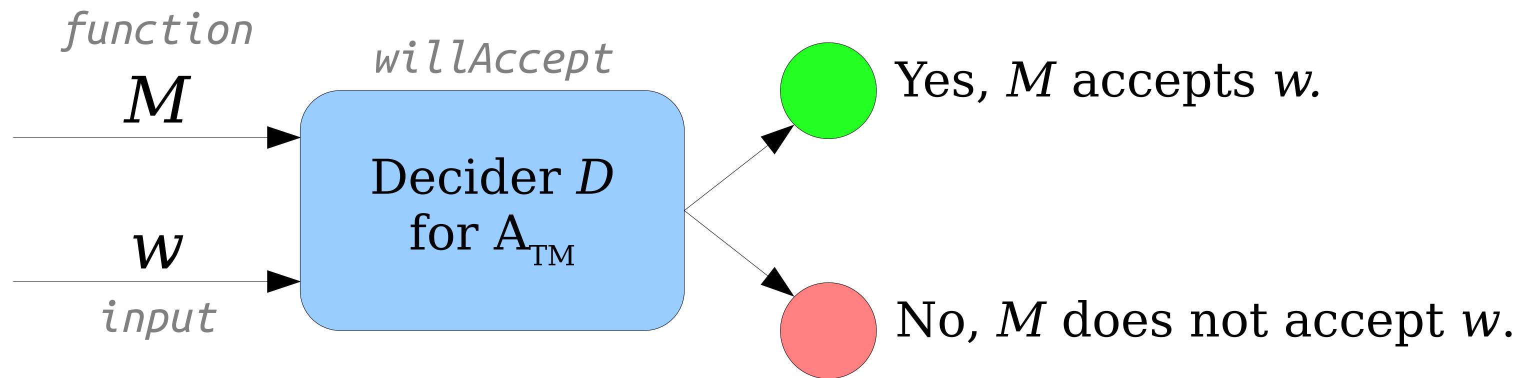
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

So if you trace through the implications here…

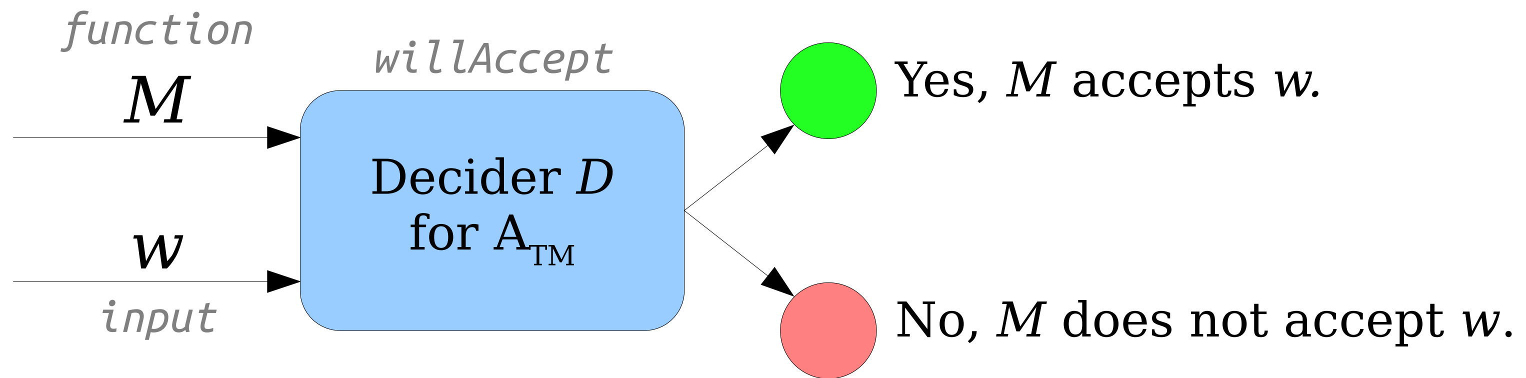
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

So if you trace through the implications here…

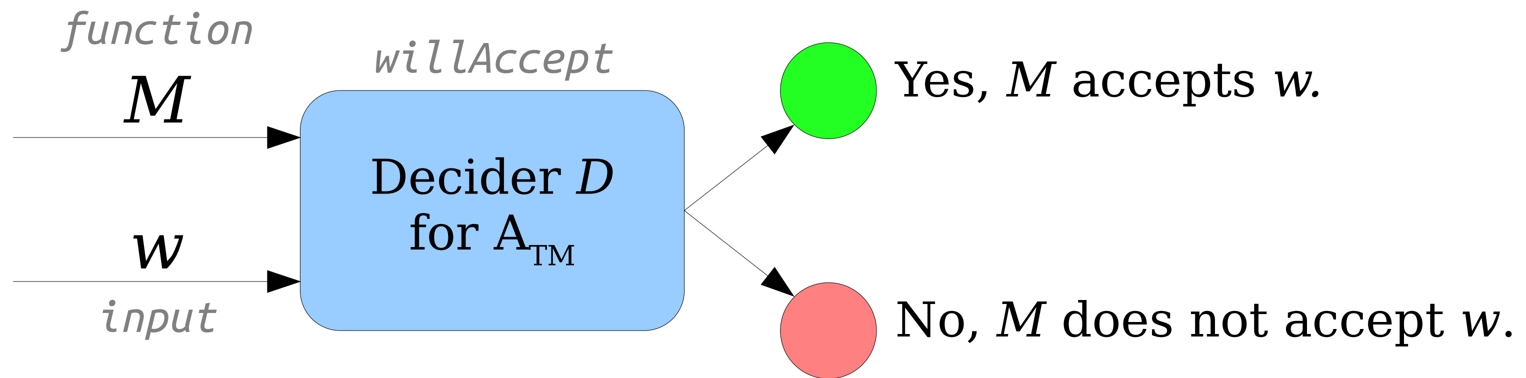
$A_{TM} \in \mathbf{R}$

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.

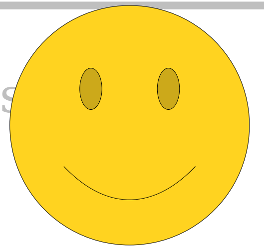
```
bool trickster(string input) {
  string me = /* source
               * code of
               * trickster
               */;
  if (willAccept(me, input)) {
    return false;
  } else {
    return true;
  }
}
```

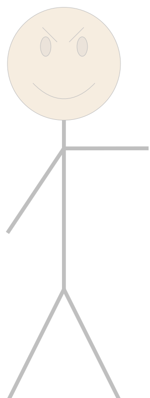
You can see the starting assumption that $A_{TM}$ is decidable leads to a contradiction – we're done!

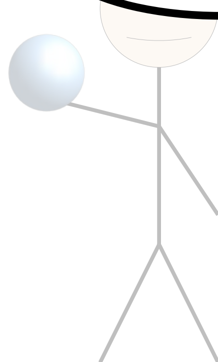
```
bool willAccept(string function, string input) {
    // Returns true if function(input) returns true.
    // Returns false otherwise.
}

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

Here's that initial lecture slide again.







trickster willAccept

↔

willAccept(me, input) returns true

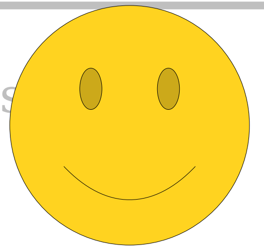
↔

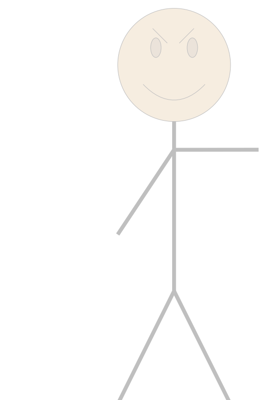
trickster(input) returns false

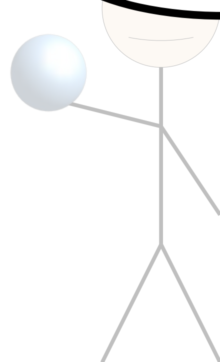
```
bool willAccept(string function, string input) {
   // Returns true if function(input) returns true.
   // Returns false otherwise.
}

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

Take a look at it more closely.







↔

willAccept(me, input) returns true

↔

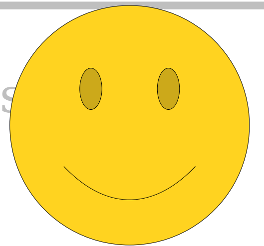
trickster(input) returns false

trickster willAccept

```
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);
}
```

Do you better understand where the **trickster** function comes from?

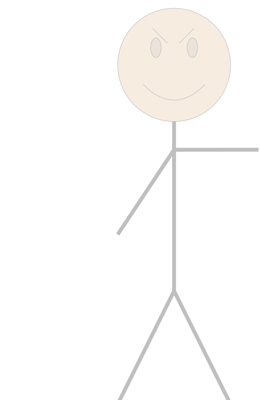


↔

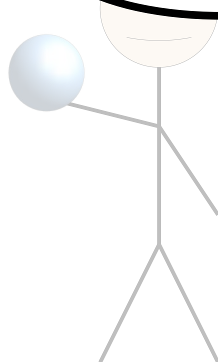
willAccept(me, input) returns true

↔

trickster(input) returns false







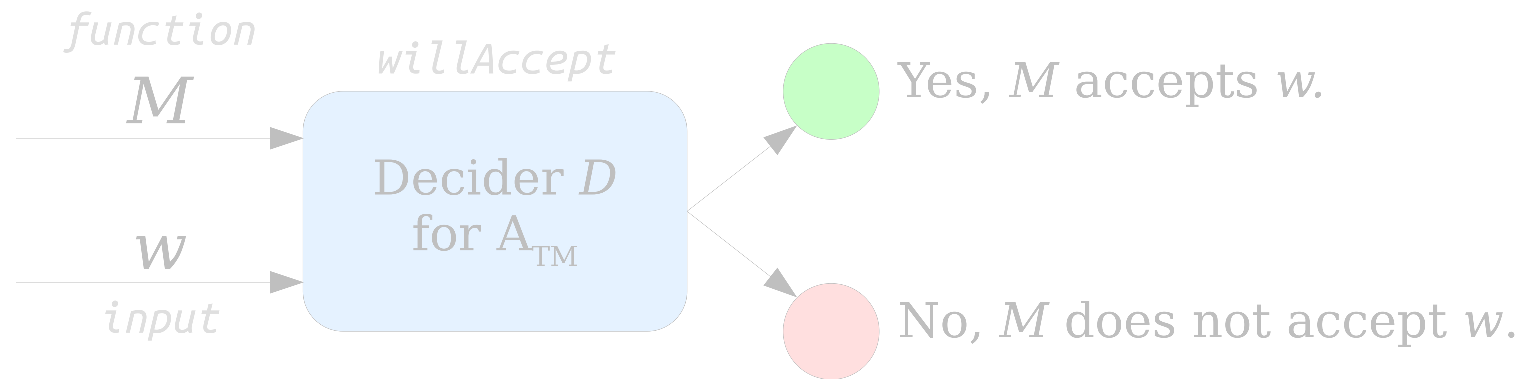
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

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

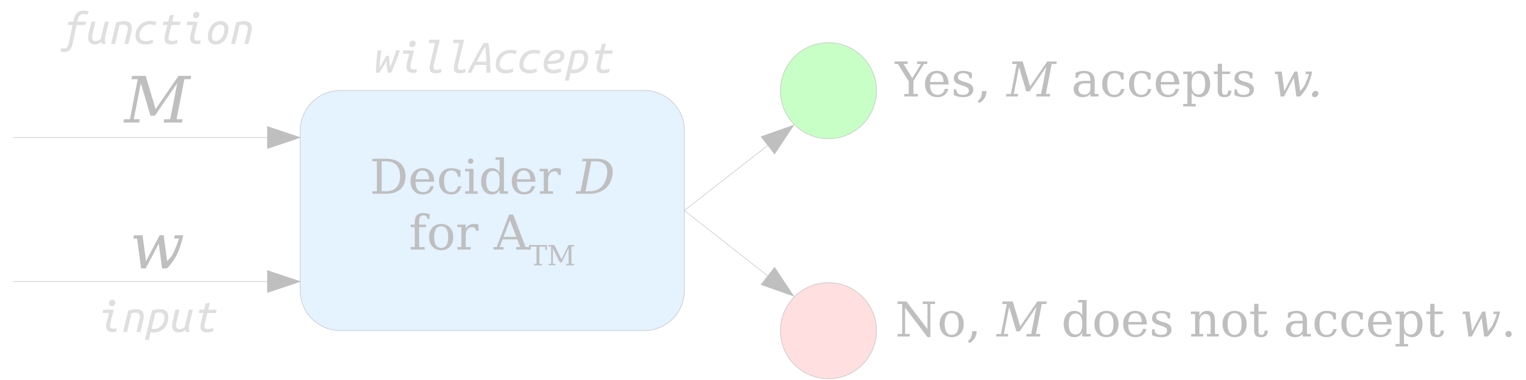
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

This progression comes up in all the self-reference proofs we've done this quarter.

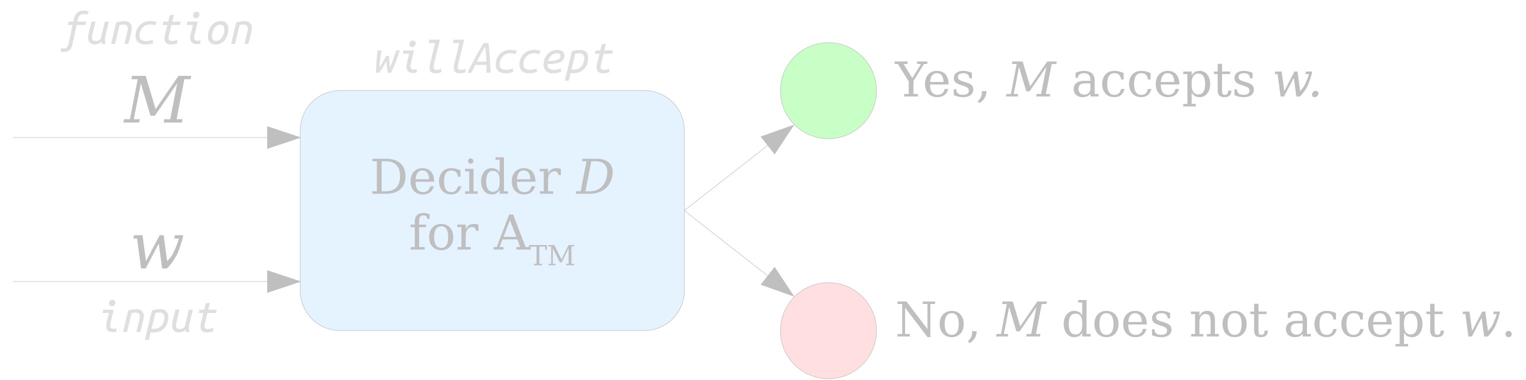
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

We'll do another example of this in a little bit.

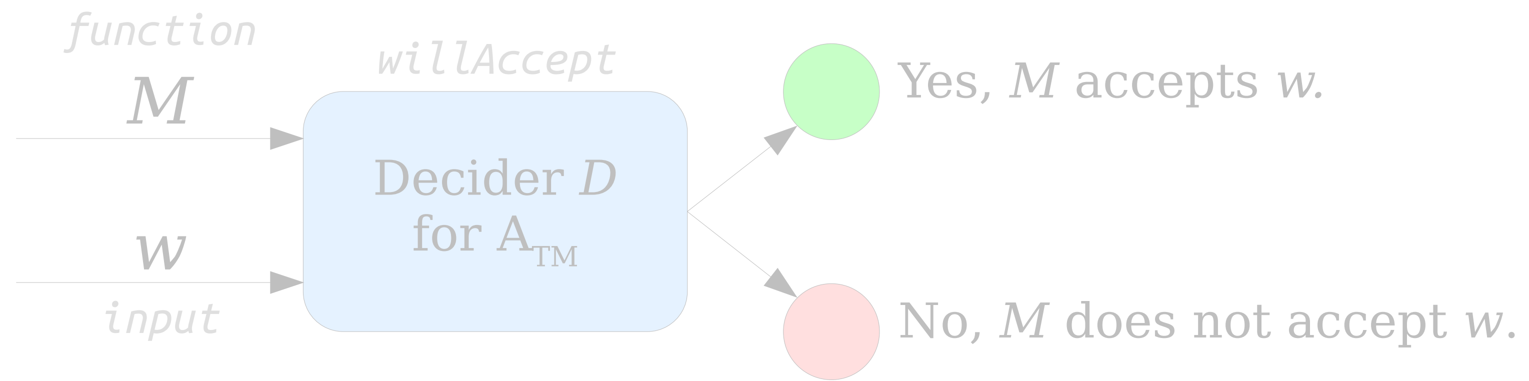
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

Before we move on, though, I thought I'd take a minute to talk about a few common questions we get.

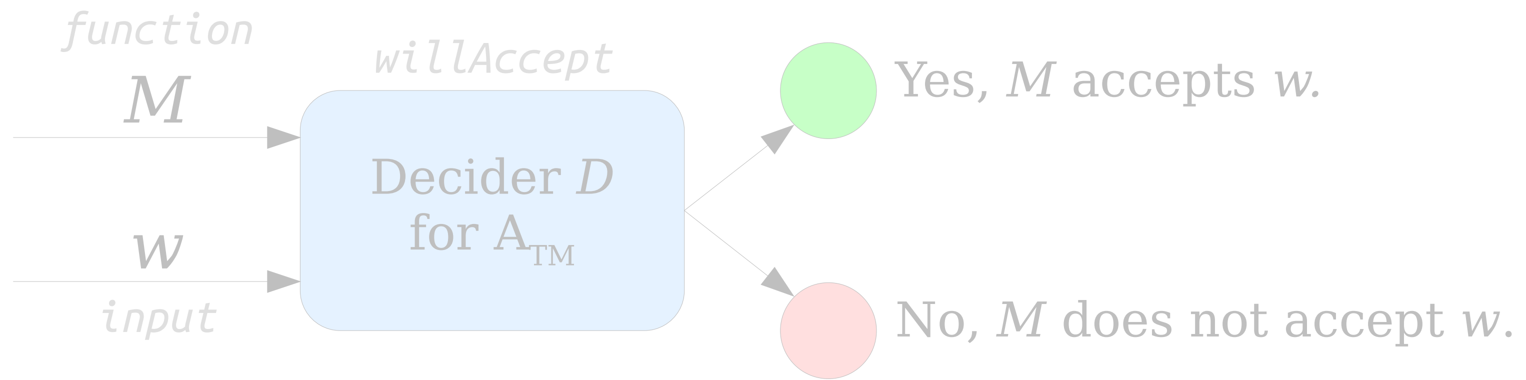
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

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

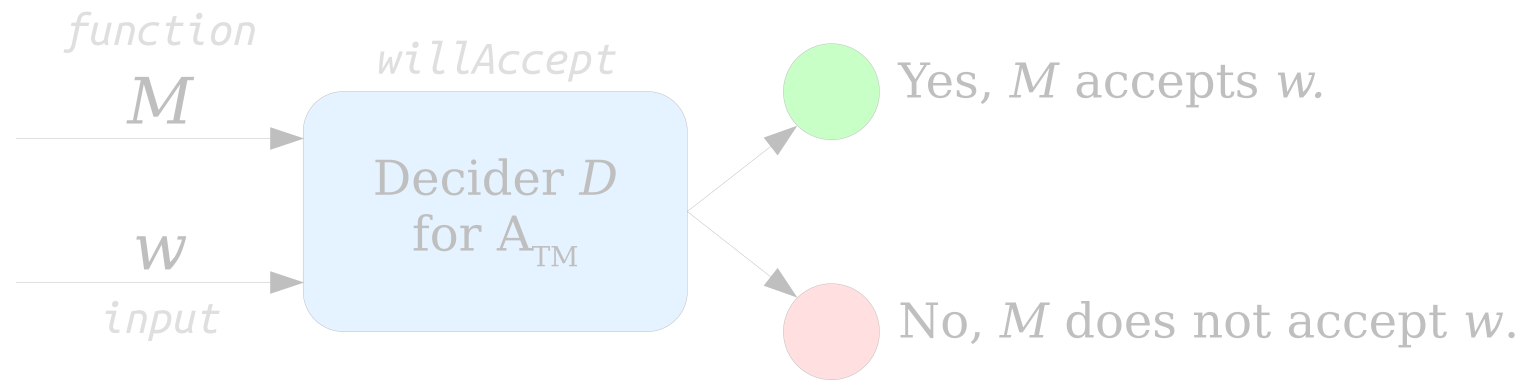
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

This is the case where trickster is supposed to accept. We need to program it so that it doesn't.

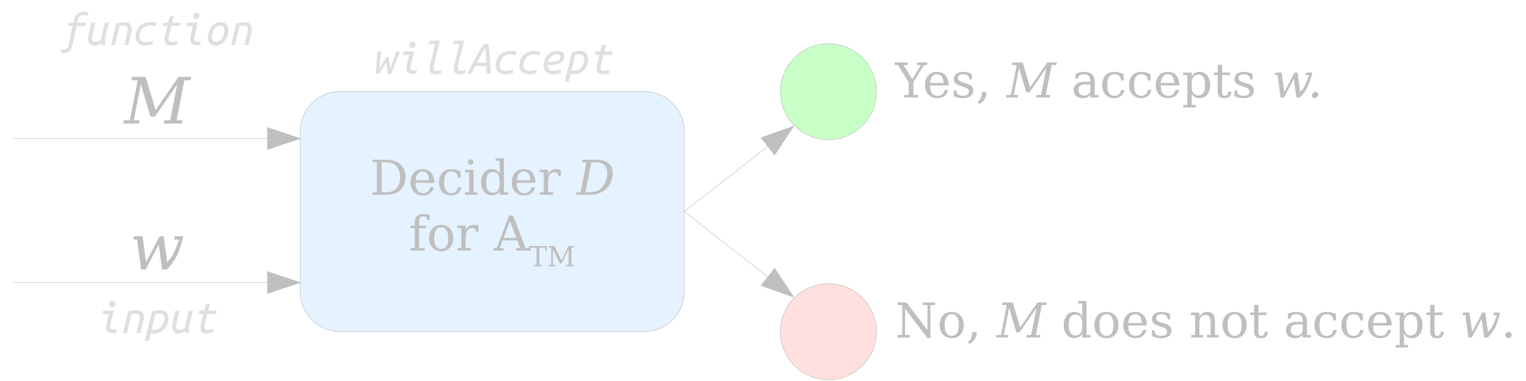
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

Here, the way we ended up doing that was by having `trickster` reject its input.

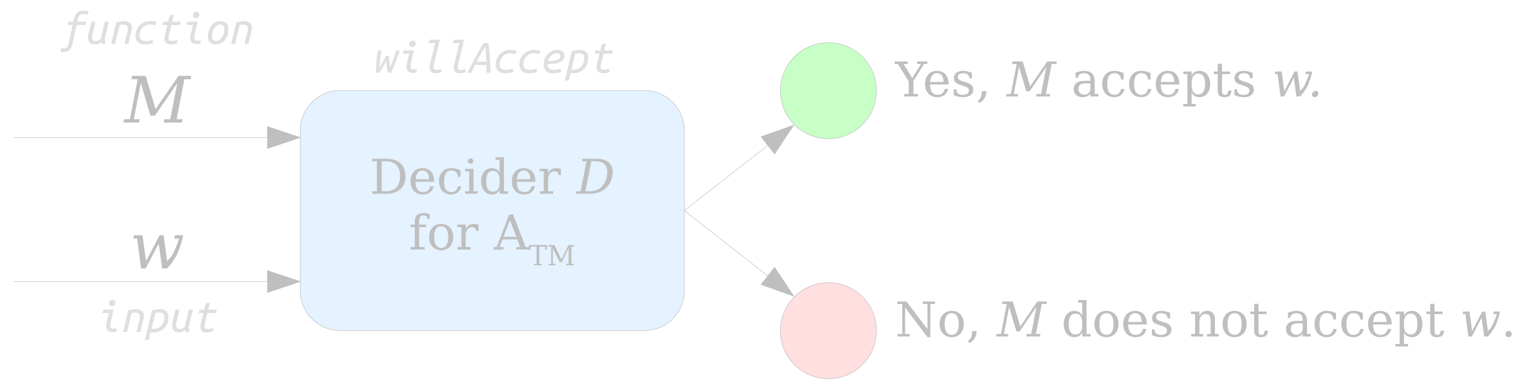
$A_{TM} \in \mathbf{R}$

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; // <--- here
  } else {
    return true;
  }
}
```

I mentioned that there were other things we could do here as well.

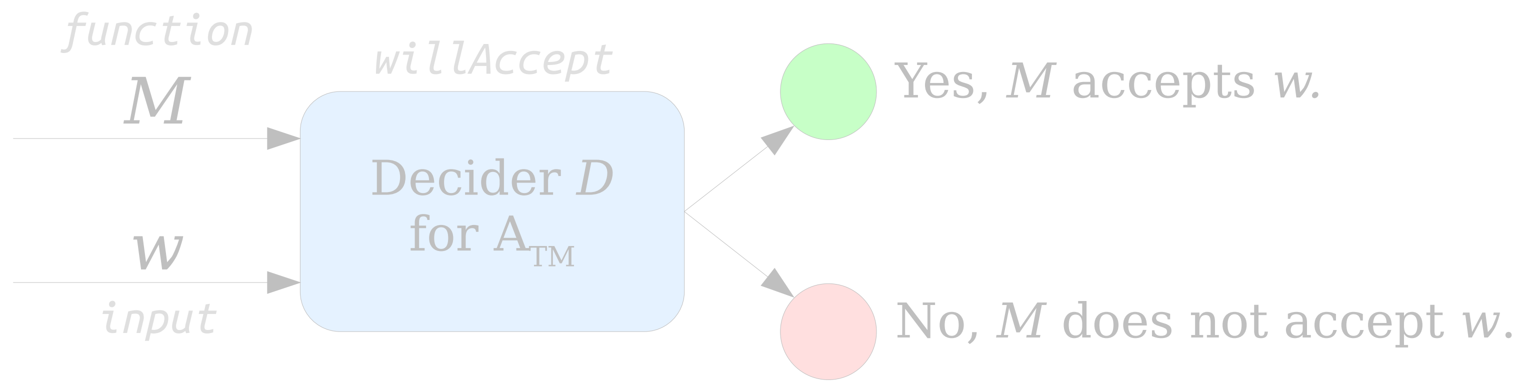
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

Here's another option. We could have `trickster` go into an infinite loop in this case.

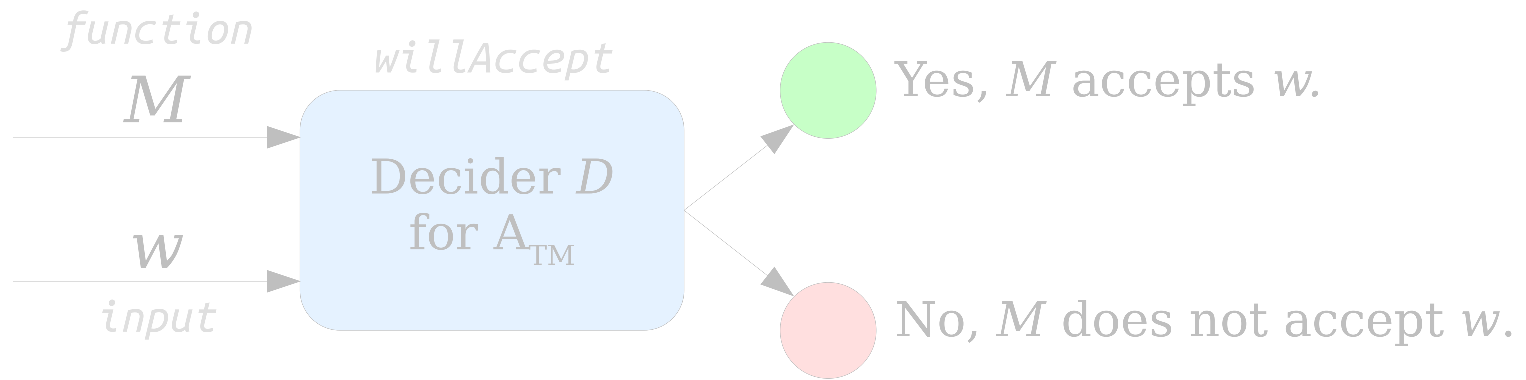
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

The design spec here says trickster needs to not accept in this case, and indeed, that's what happens!

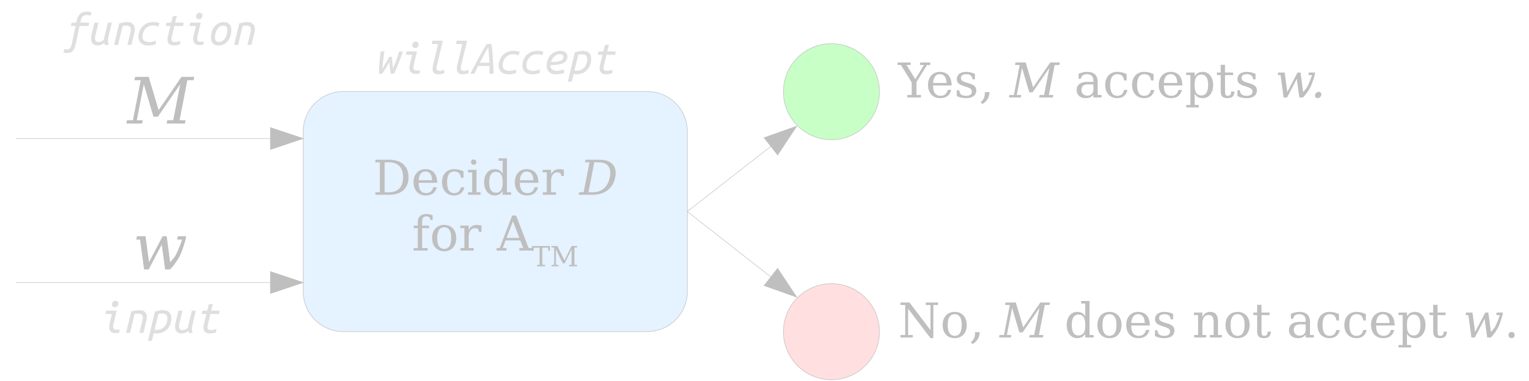
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

A lot of people ask whether this is allowed, since we were assuming we had a decider and deciders can't loop.

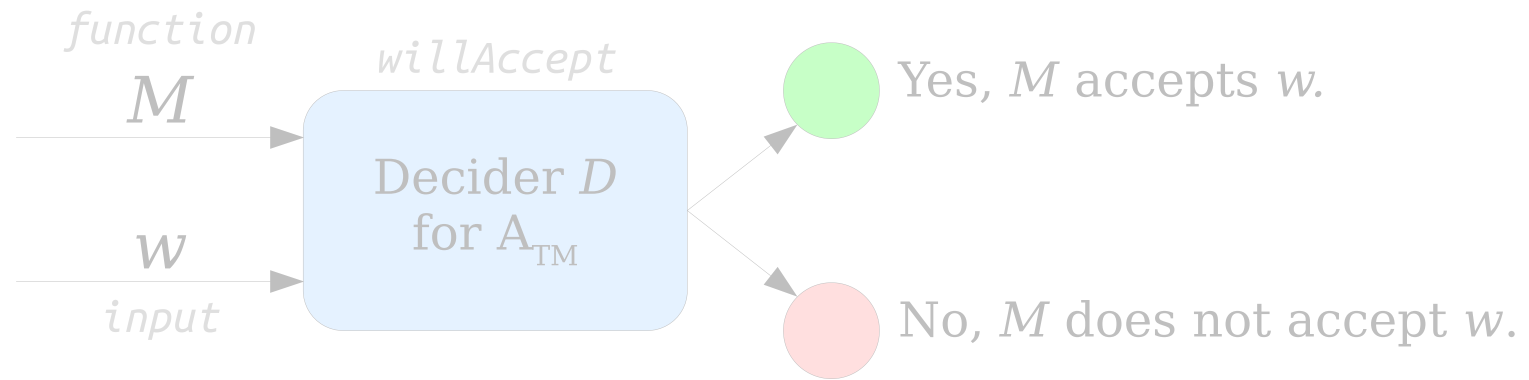
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

Turns out, this is totally fine!

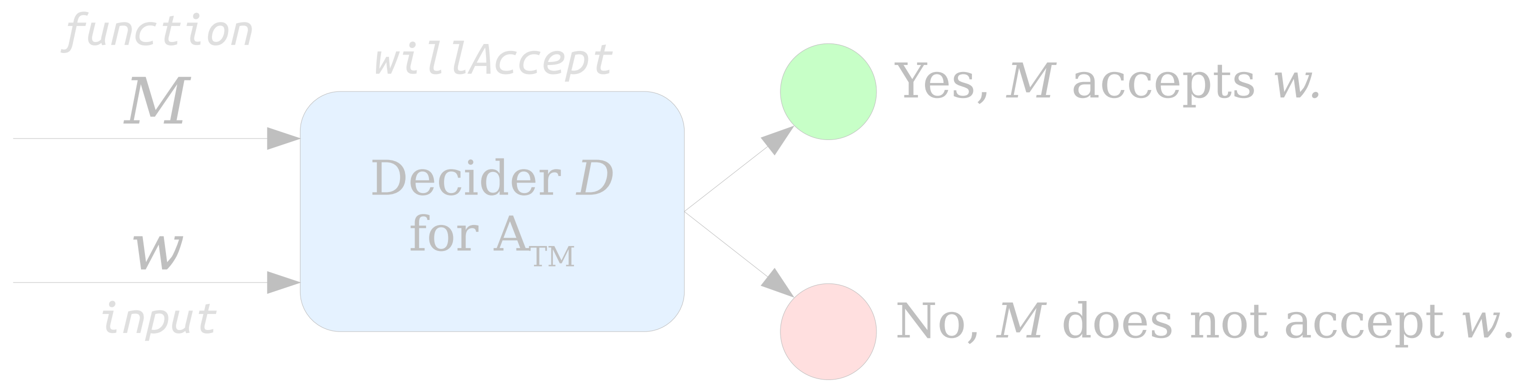
$A_{TM} \in \mathbf{R}$

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 are two different functions here.

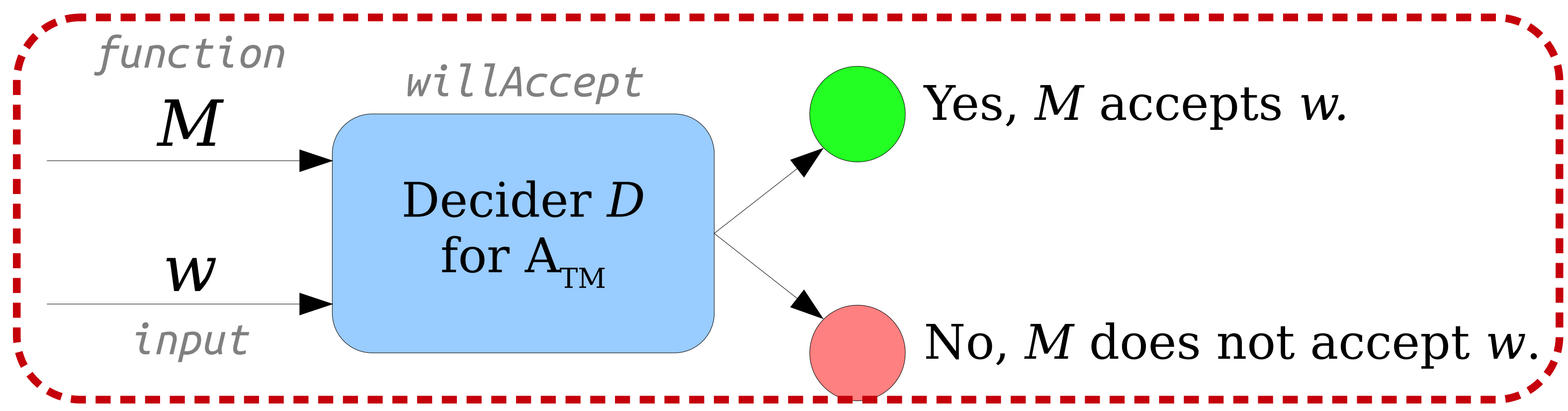
$A_{TM} \in \mathbf{R}$

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)
```

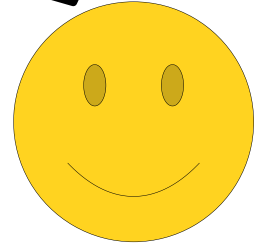
`trickster` design specification:

√ 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;
  }
}
```

First, there's this decider D. D is a decider, so it's required to halt on all inputs.



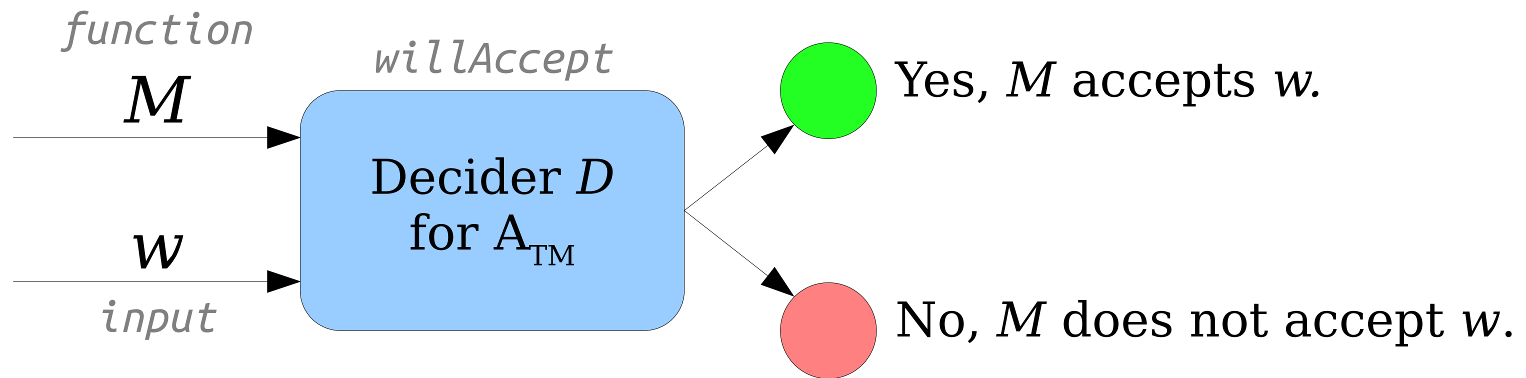
$A_{TM} \in \mathbf{R}$

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

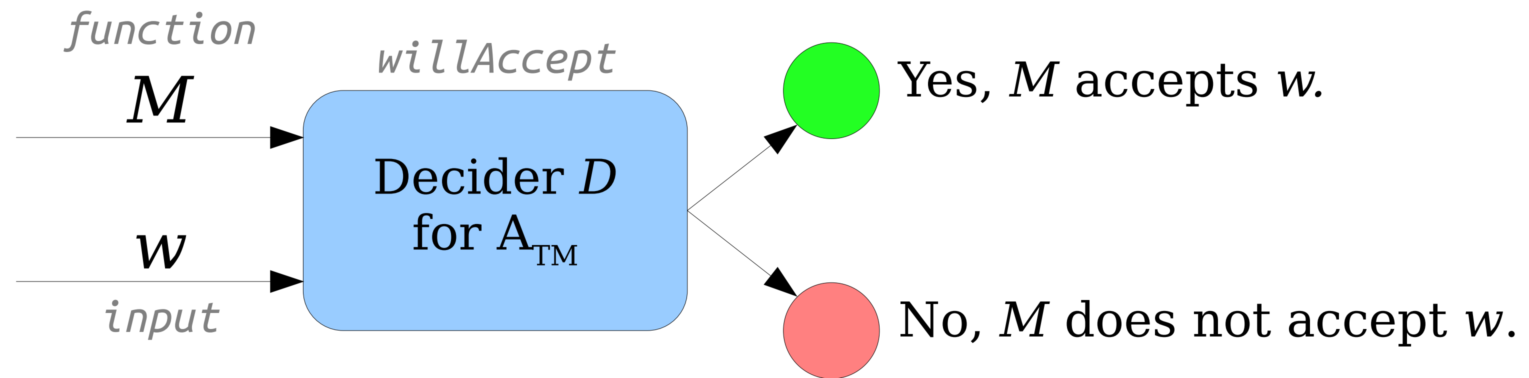
$A_{TM} \in \mathbf{R}$

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)
```

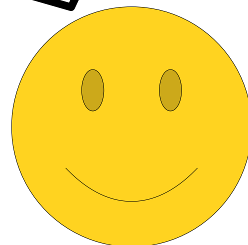
trickster design specification:

√ 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;
  }
}
```

These proofs involve two different programs: the decider for the language, and the self-referential program.



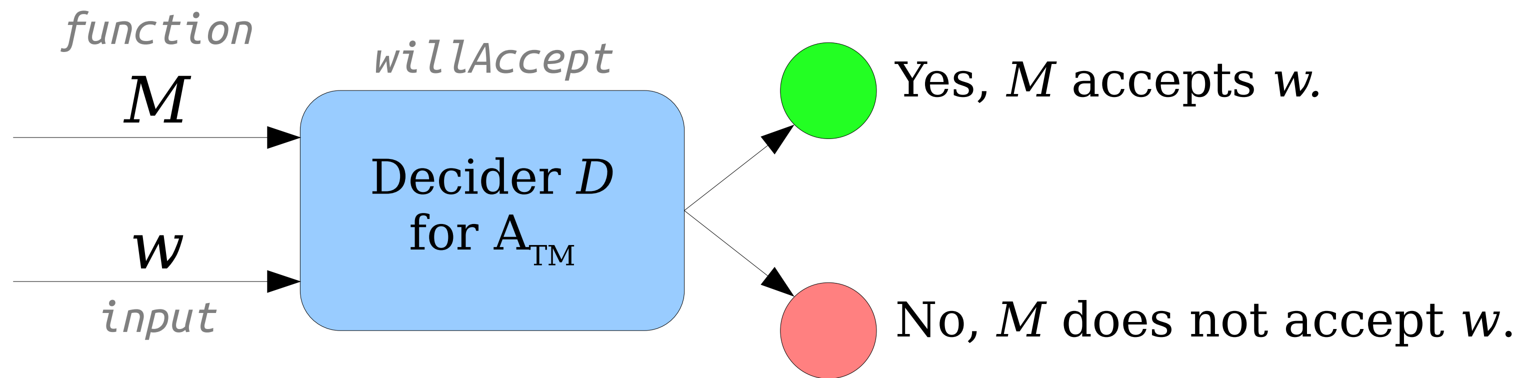
$A_{TM} \in \mathbf{R}$

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)
```

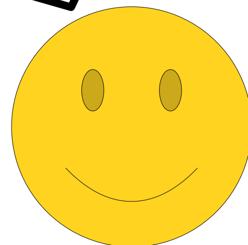
`trickster` design specification:

√ 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;
  }
}
```

The decider is always required to halt, but `trickster` is not.



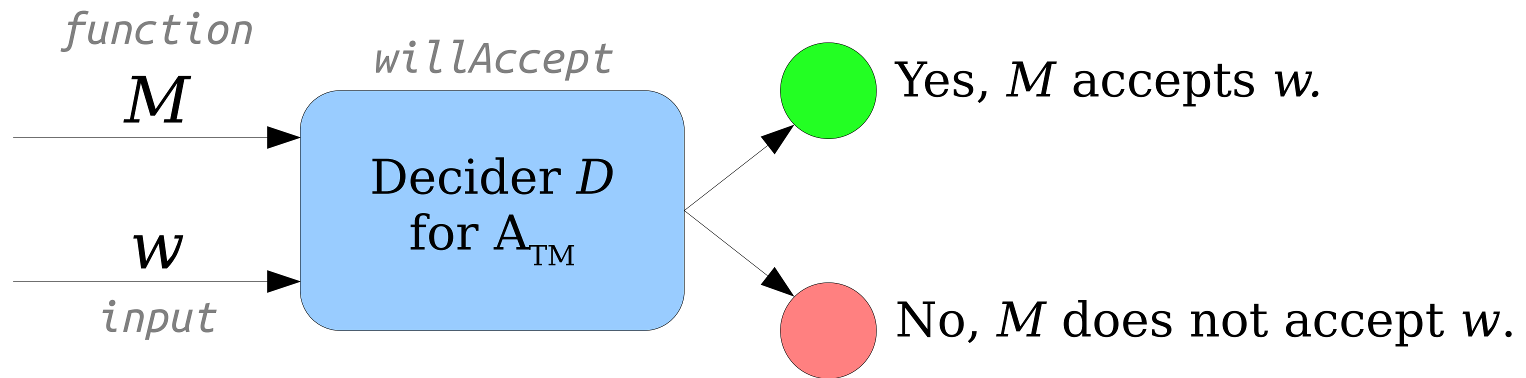
$A_{TM} \in \mathbf{R}$

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)
```

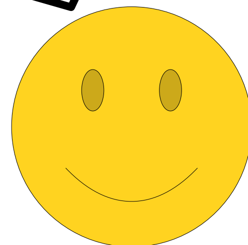
`trickster` design specification:

√ 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;
  }
}
```

Let's undo all these changes so that we can talk about the next common question.



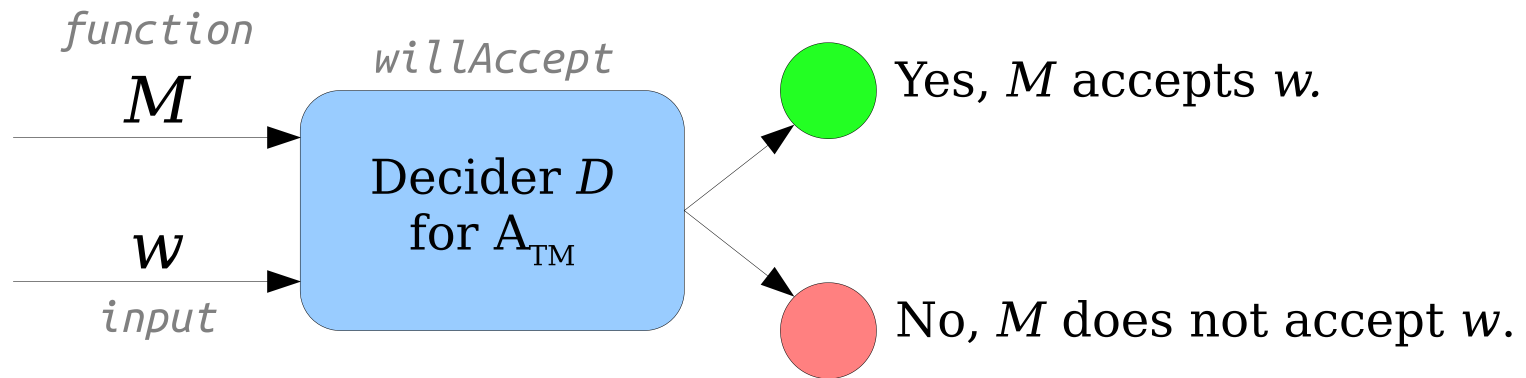
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

Much better!

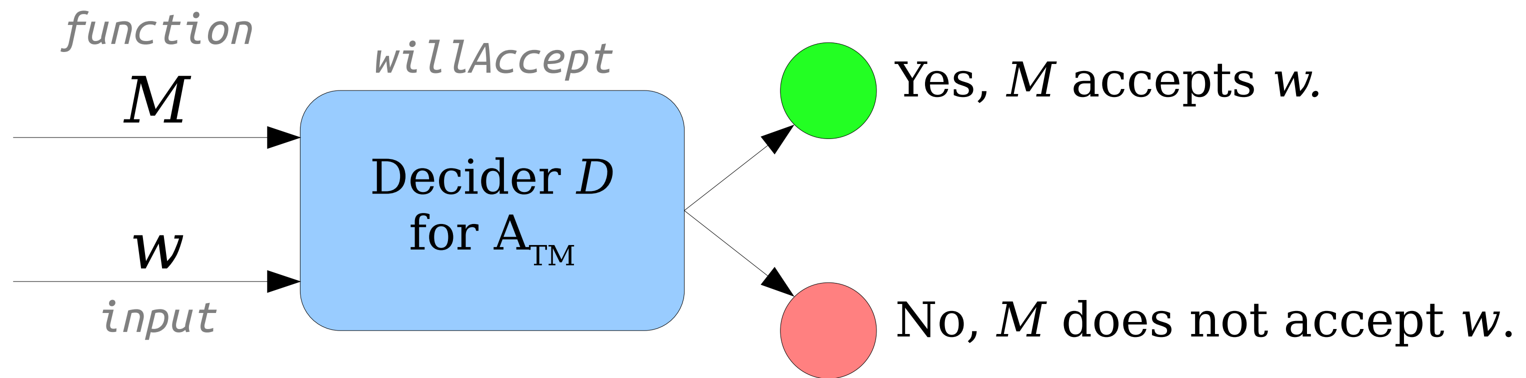
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

On to the next question.

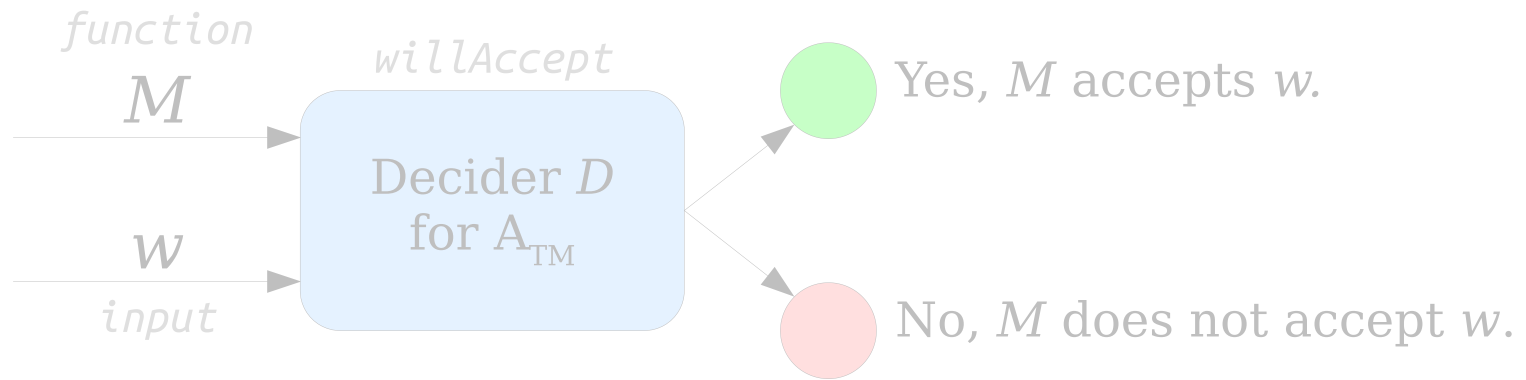
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

A lot of people take a look at the program we've written…

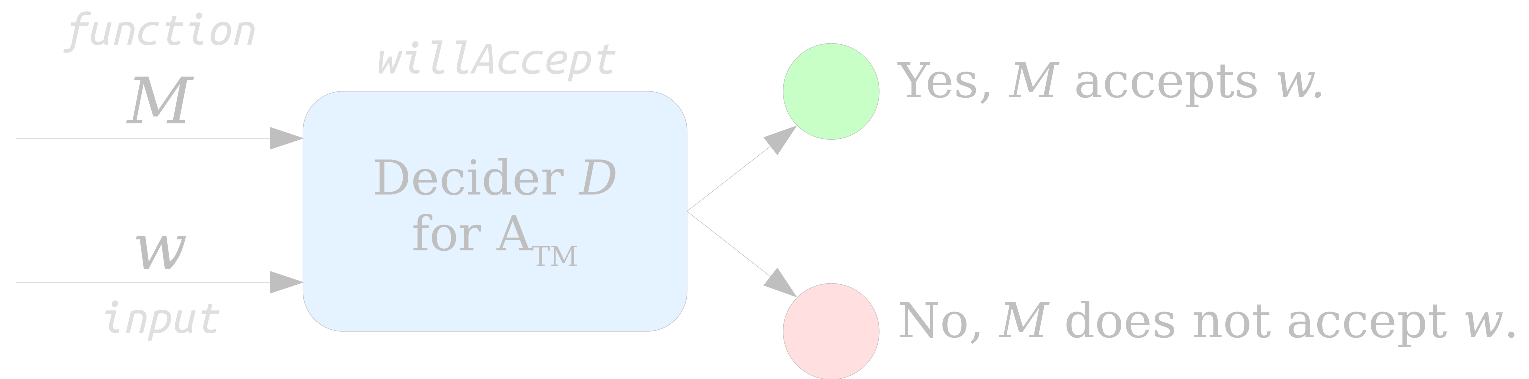
$A_{TM} \in \mathbf{R}$

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; // <-- here
  } else {
    return true;  // <-- here
  }
}
```

… and ask what happens if we take these two lines…

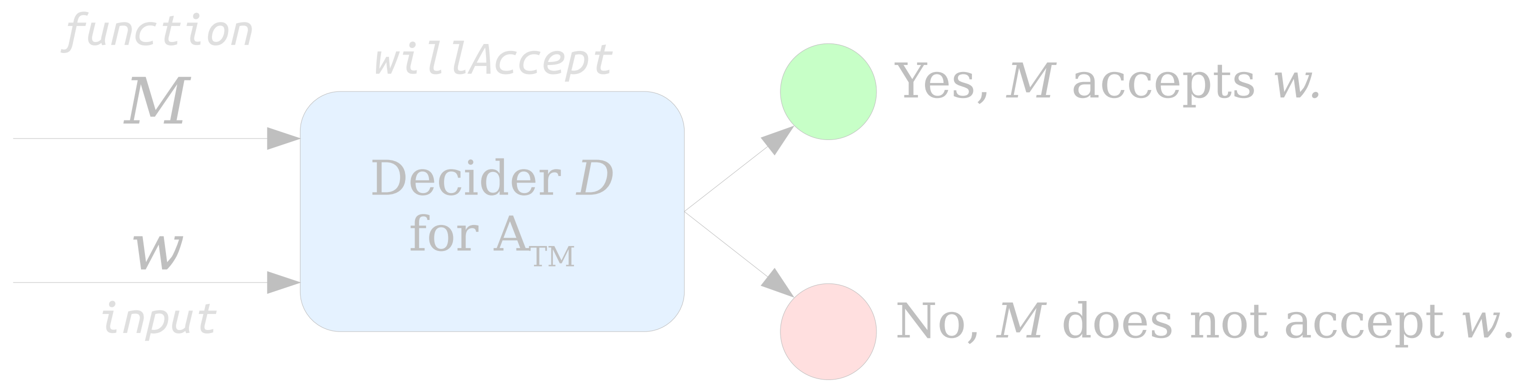
$A_{TM} \in \mathbf{R}$

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;  // <-- swap!
  } else {
    return false; // <-- swap!
  }
}
```

… and swap them like this.

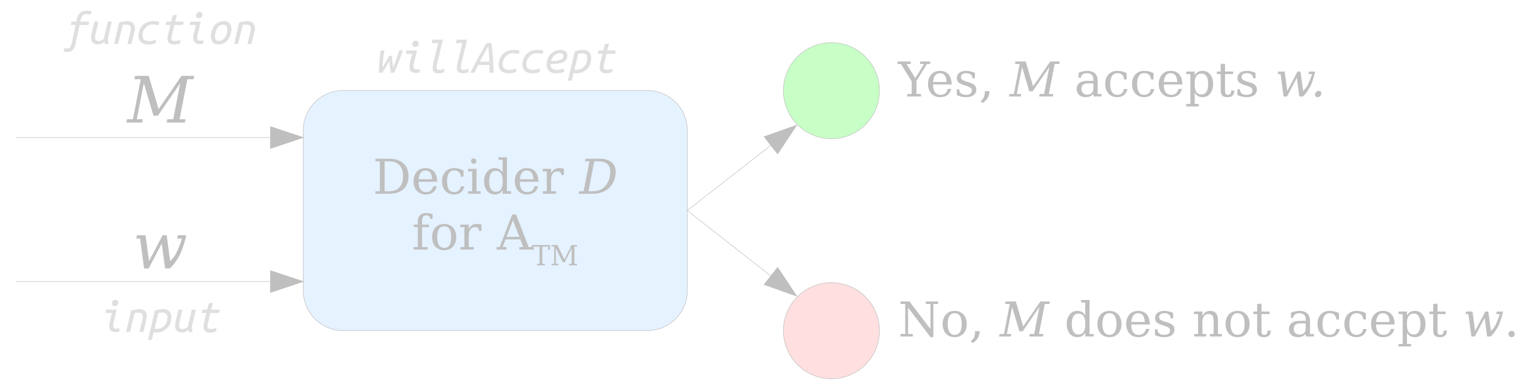
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

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

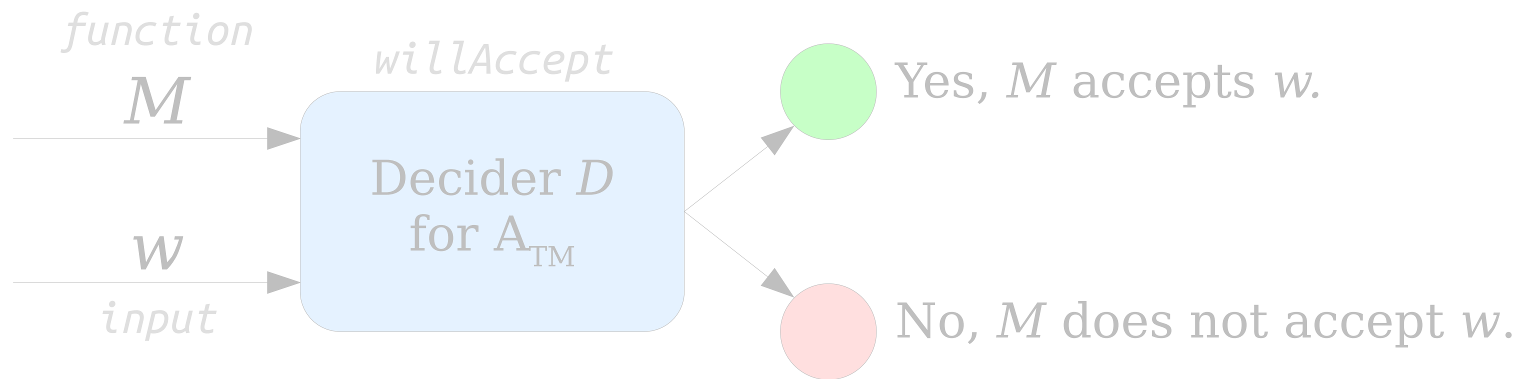
$A_{TM} \in \mathbf{R}$

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.

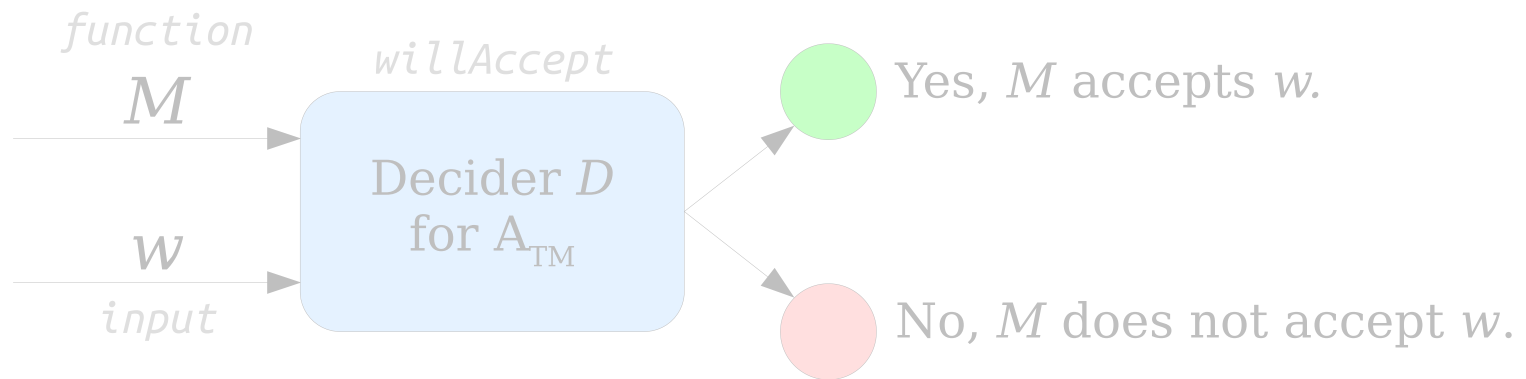
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

Notice that this trickster doesn't have the behavior given over here.

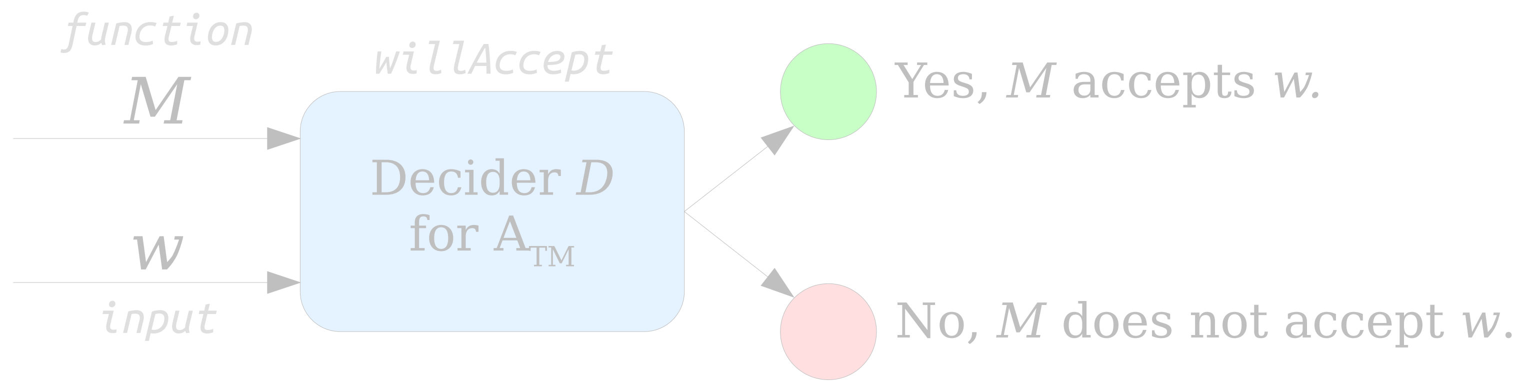
$A_{TM} \in \mathbf{R}$

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

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;
  }
}
```

If you think about the behavior it **does** have, it looks more like this.

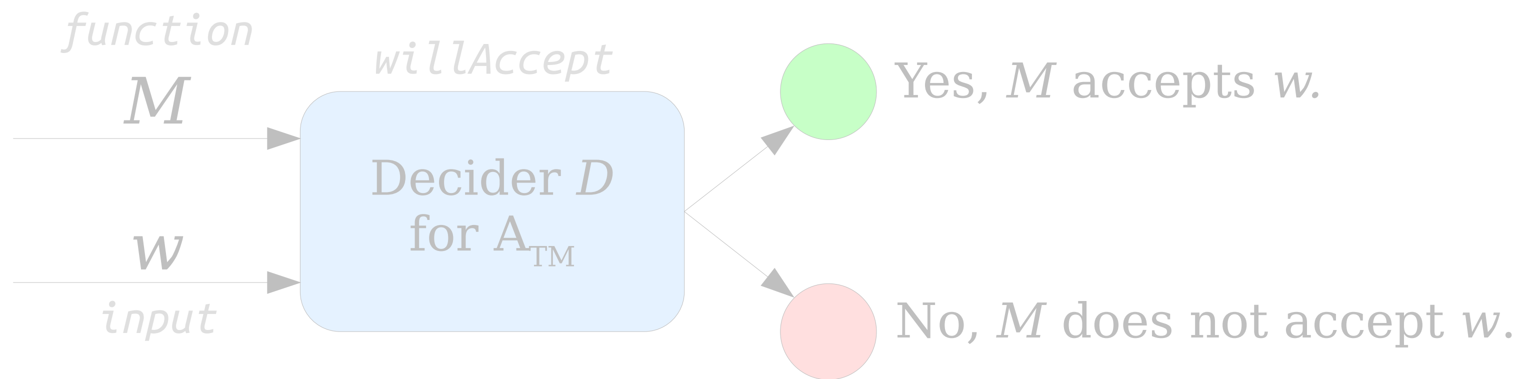
$A_{TM} \in \mathbf{R}$

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

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;
  }
}
```

Notice that this is a true statement.

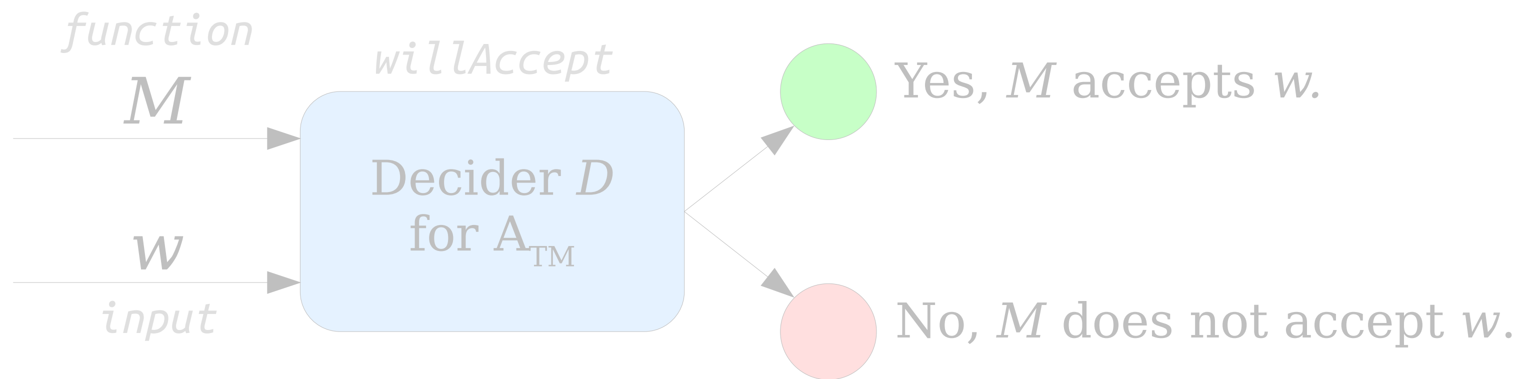
$A_{TM} \in \mathbf{R}$

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

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;
  }
}
```

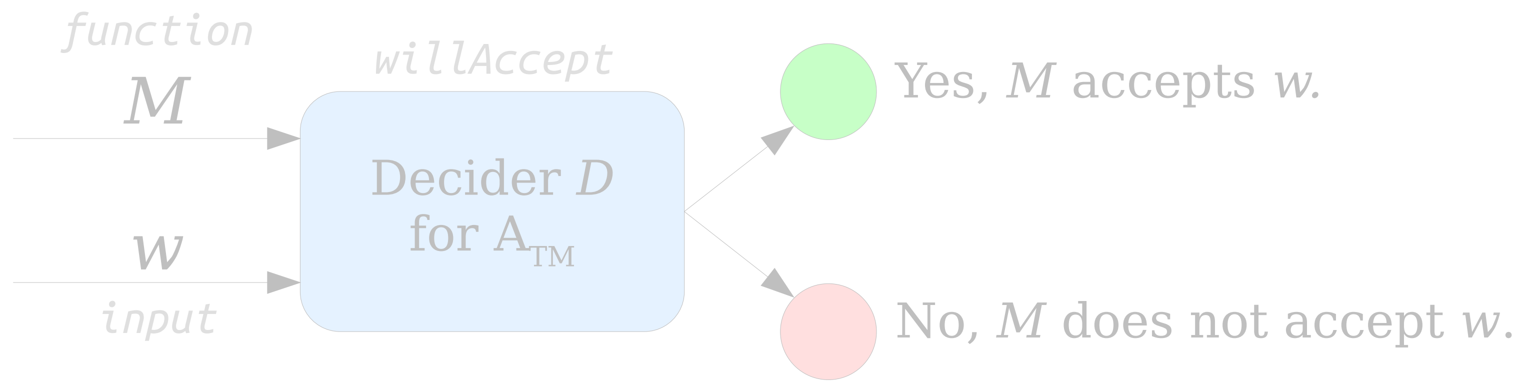
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

Instead, we've shown that we end up at a true statement.

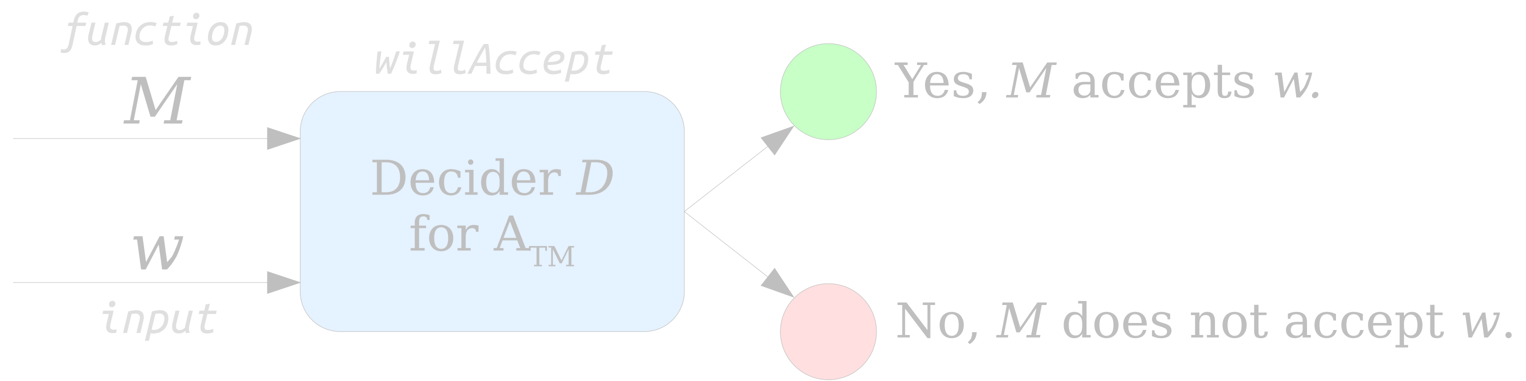
$A_{TM} \in \mathbf{R}$

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

$\top$




```
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;
  }
}
```

However, take a minute to look at the giant implication given here.

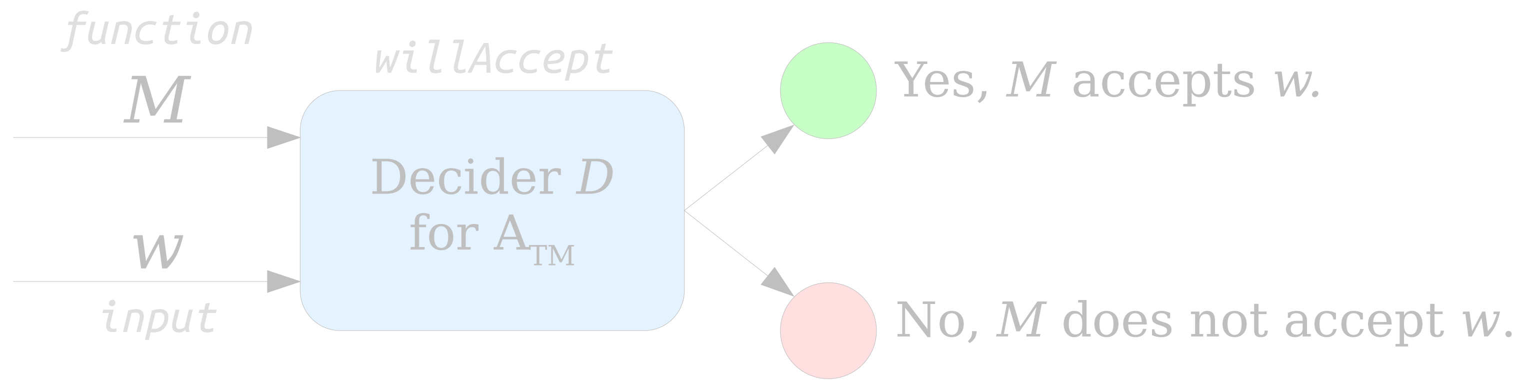
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

Overall, this shows that

$A_{TM} \in \mathbf{R} \rightarrow \top$

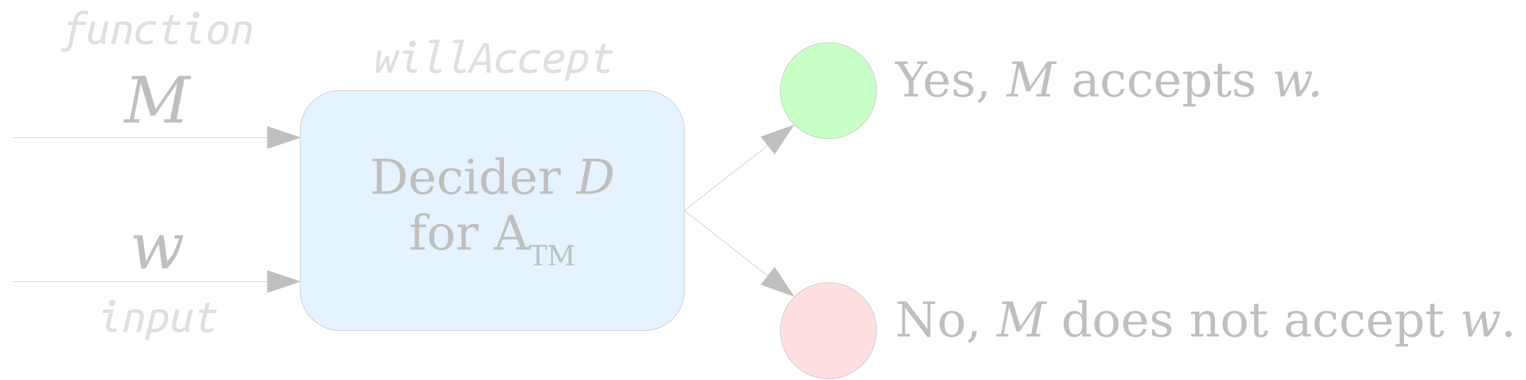
$A_{TM} \in \mathbf{R}$

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

$\bot$




```
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;
  }
}
```

Does this statement say anything about whether $A_{TM}$ is decidable?

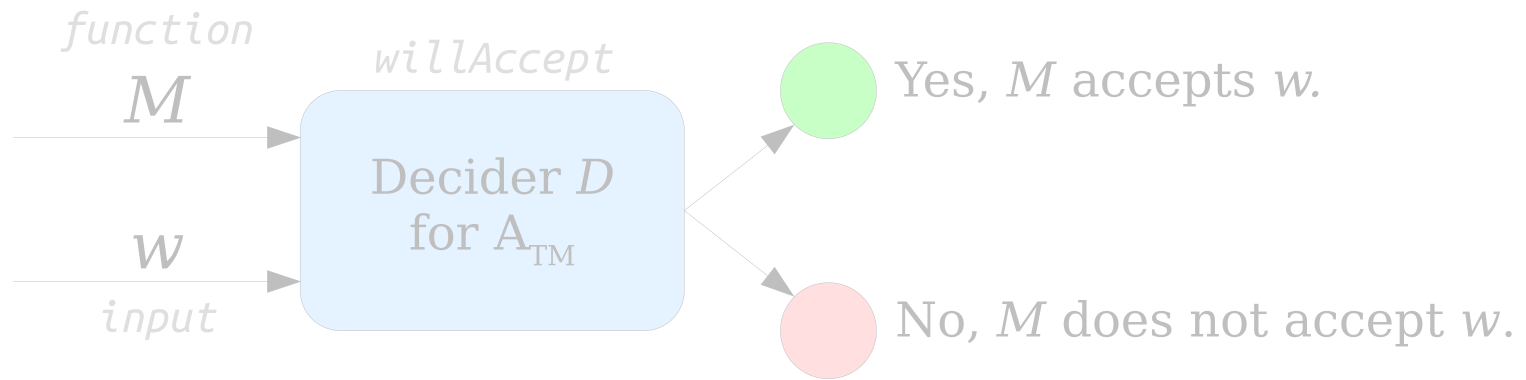
$A_{TM} \in \mathbf{R}$

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

$\top$




```
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;
  }
}
```

Nope! Remember, *anything* implies a true statement.

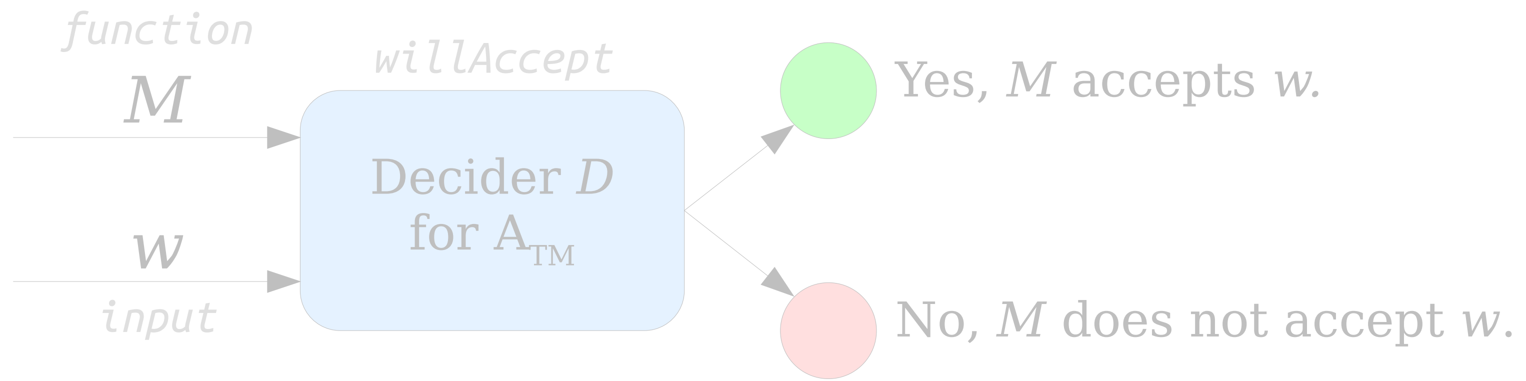
$A_{TM} \in \mathbf{R}$

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

$\perp$




```
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;
  }
}
```

We have no way of knowing whether $A_{TM} \in \mathbf{R}$ or not just by looking at this statement.

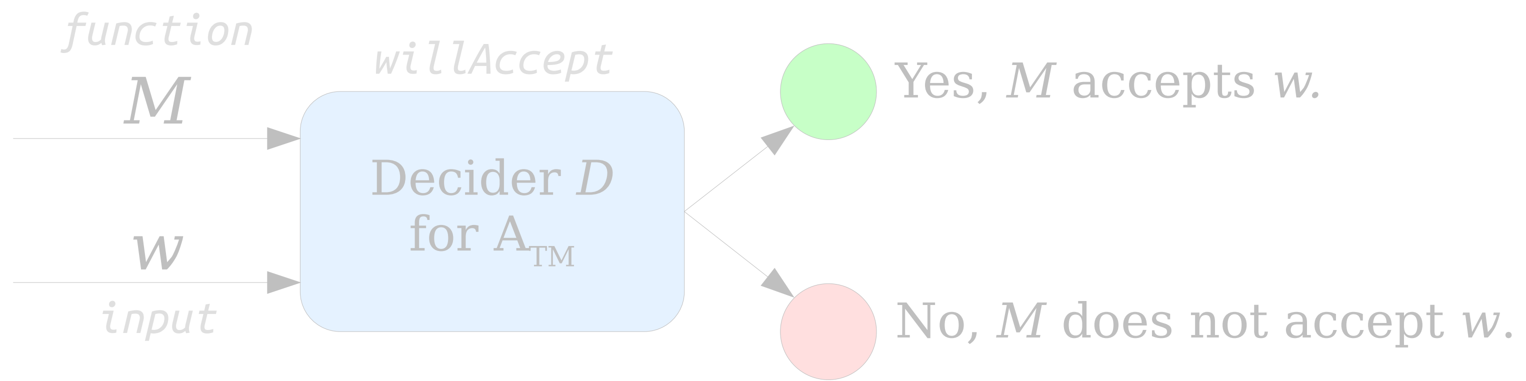
$A_{TM} \in \mathbf{R}$

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

$\perp$




```
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;
  }
}
```

The fact that we didn't get a contradiction doesn't mean that $A_{TM}$ is decidable.

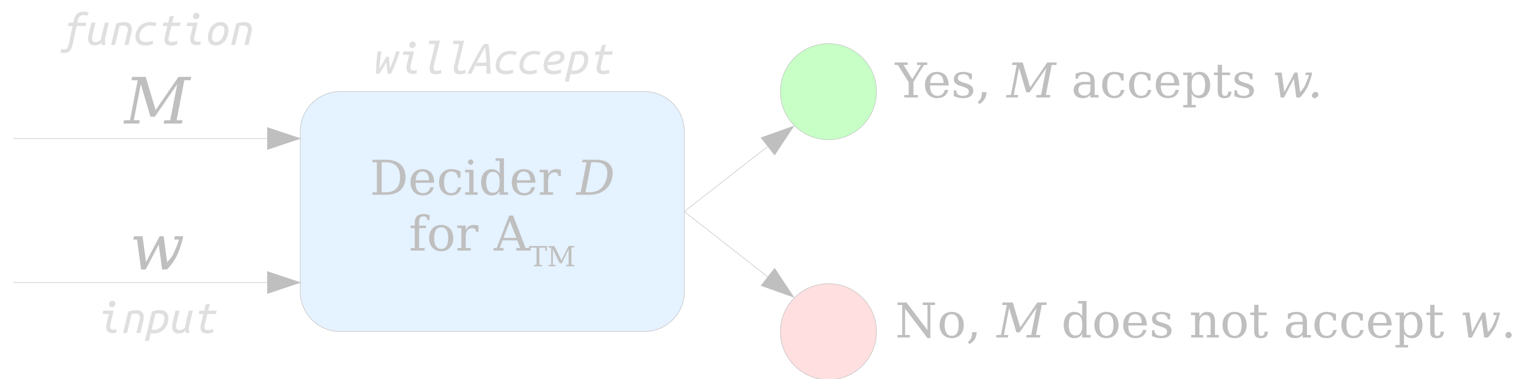
$A_{TM} \in \mathbf{R}$

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

$\bot$




```
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;
  }
}
```

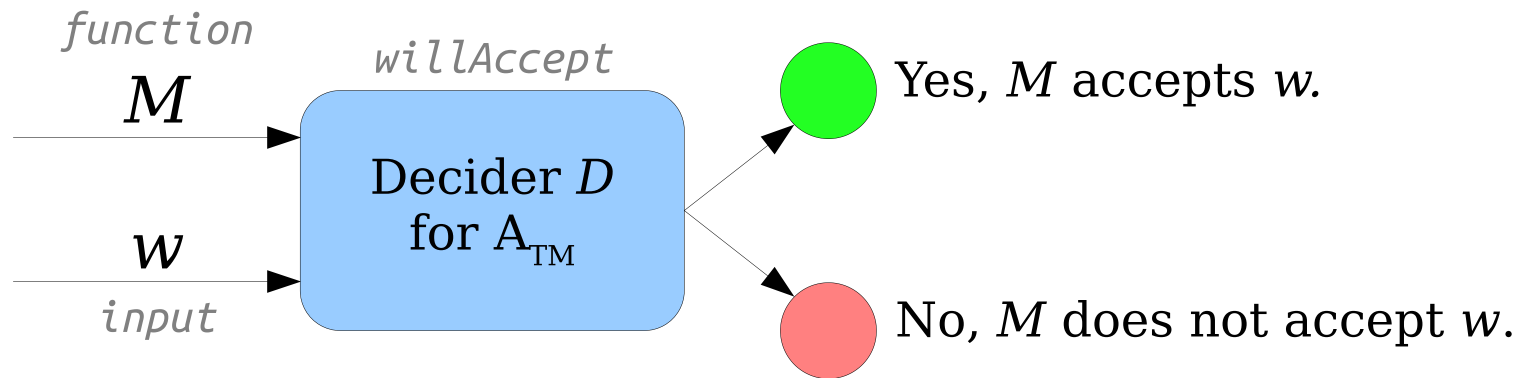
$A_{TM} \in \mathbf{R}$

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;
  }
}
```

Much better!

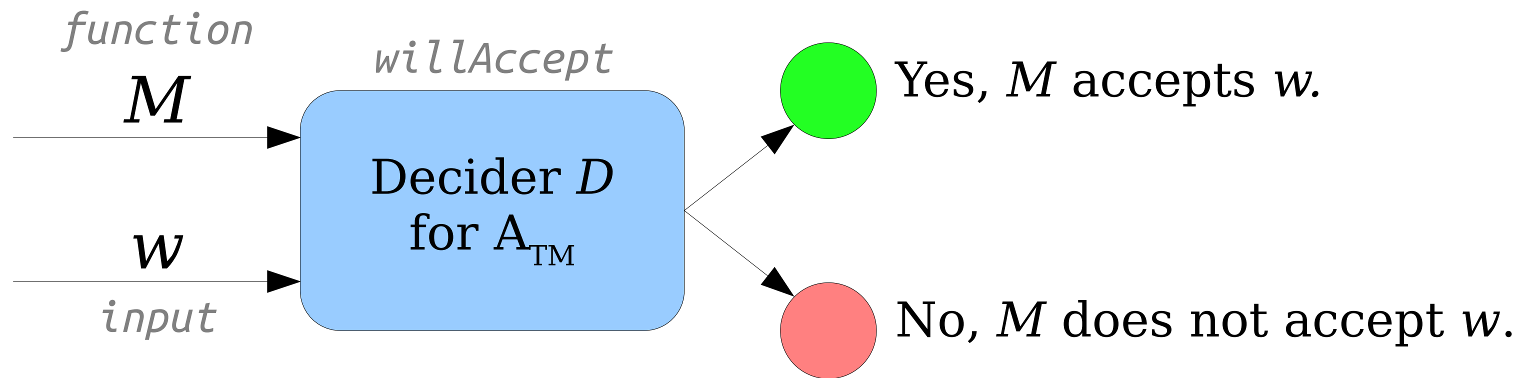
$A_{TM} \in \mathbf{R}$

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)
```

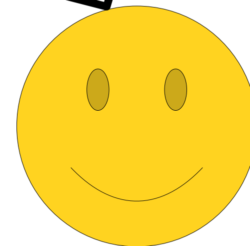
trickster design specification:

- √ 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;
  }
}
```

Take a look at the general structure of how we got here. Then, let's go do another example.

Do you remember the secure voting problem from lecture?

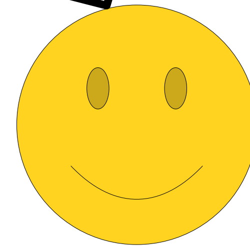


$M$ is a secure voting machine

if and only if

$M$ accepts its input precisely if it has more **r**'s than **d**'s.

We said that a TM $M$ is a secure voting machine if it obeys the above rule.

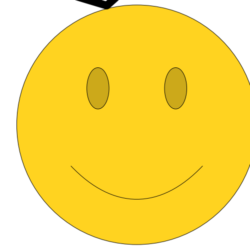


$M$ is a secure voting machine

if and only if

$M$ 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.

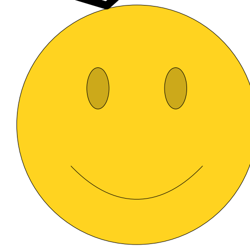


$M$ is a secure voting machine

if and only if

$M$ accepts its input precisely if it has more `r`'s than `d`'s.

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.

```
bool countVotes(string input) {
  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.

```
bool countVotes(string input) {
  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.

Here's a program where *no one* knows whether it's a secure voting machine.

```
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.

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

```
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!

```
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.

So, to recap:
Building a secure voting machine isn't hard.
Checking whether an arbitrary program is a secure voting machine is really hard.

```
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);
}
```

Our goal is to show that the secure voting problem – the problem of checking whether a program is a secure voting machine – is undecidable.

The secure voting problem is decidable.

Following our pattern from before, we'll assume that the secure voting problem is decidable.

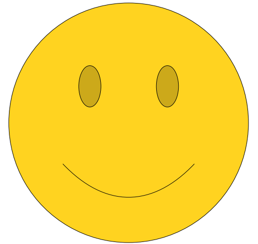
The secure voting problem is decidable.

Contradiction!

We're ultimately trying to get some kind of contradiction here.

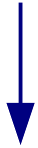
The secure voting problem is decidable.

As before, we'll take it one step at a time.



Contradiction!

The secure voting problem is decidable.



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

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.

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

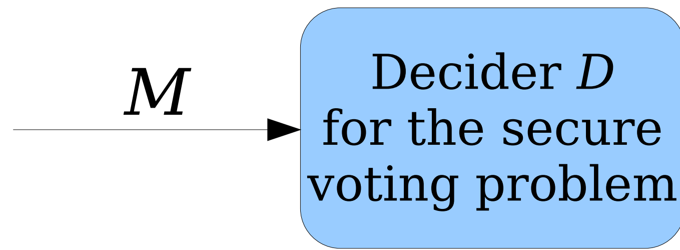
Contradiction!

Decider $D$ for the secure voting problem

So what does that look like?

The secure voting problem is decidable.

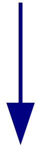
There is a decider $D$ for the secure voting problem



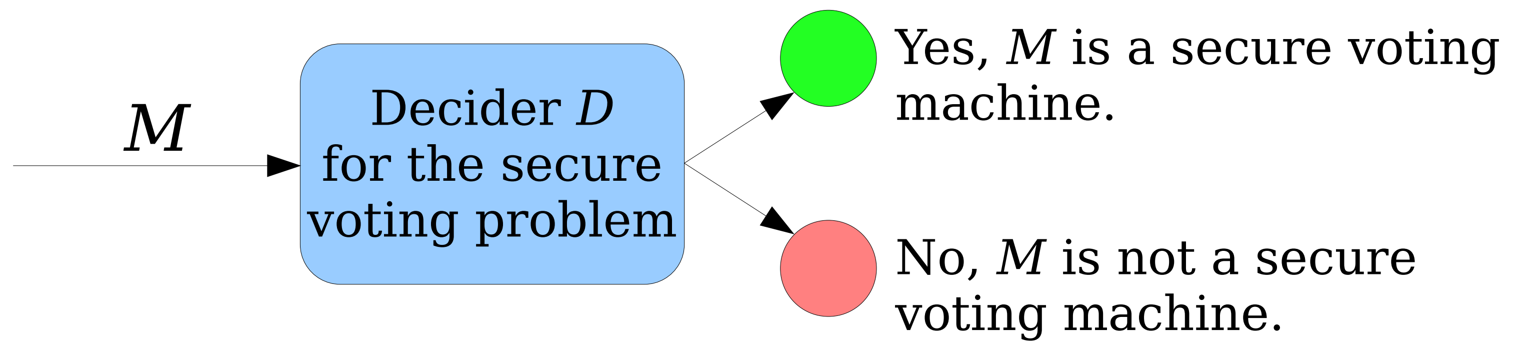

A decider for the secure voting problem will take in some TM M, which is the machine we want to specifically check.

Contradiction!

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.

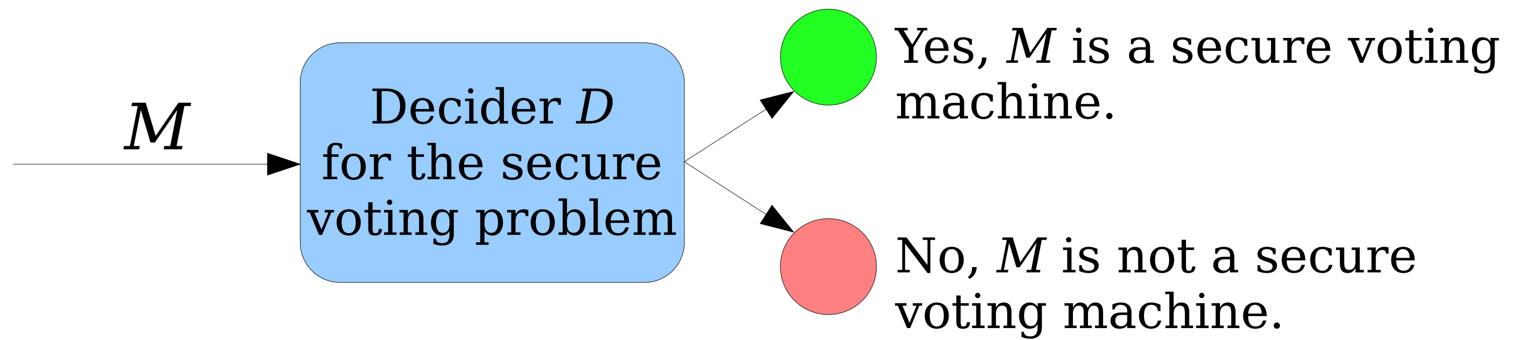
Contradiction!

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!



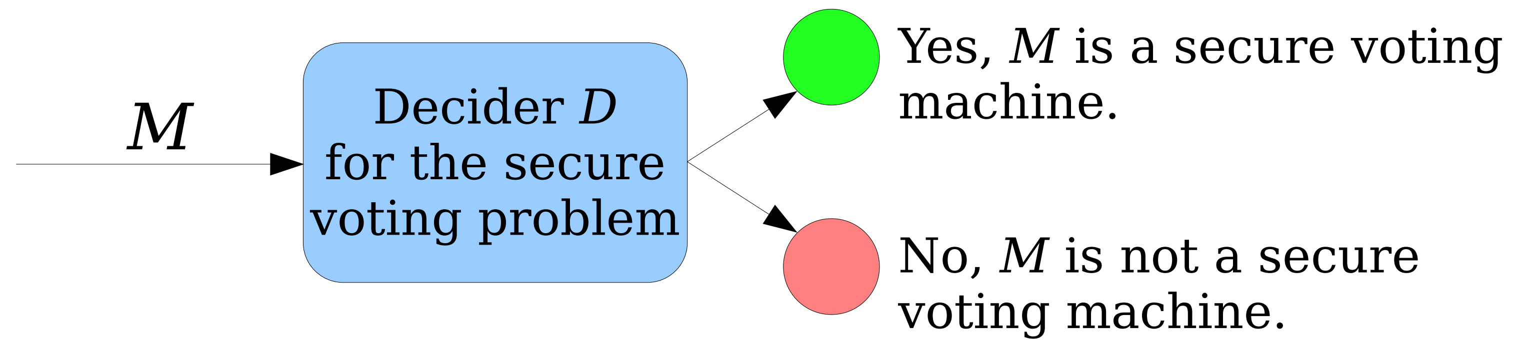

Following our pattern from before, we'll then say that we can use this decider as a subroutine in other TMs.

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!




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

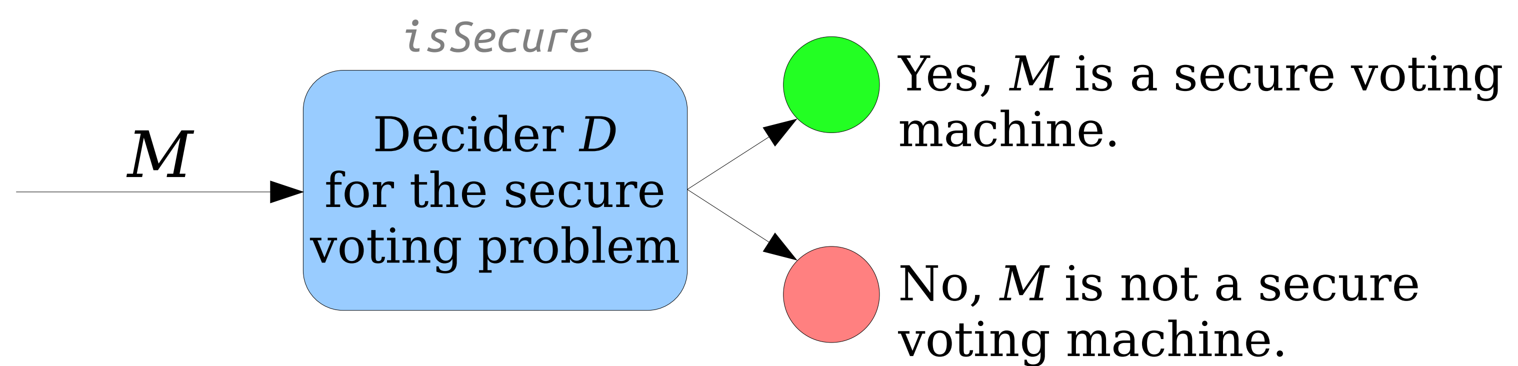
In software, that decider $D$ might look something like what's given above.

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!




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

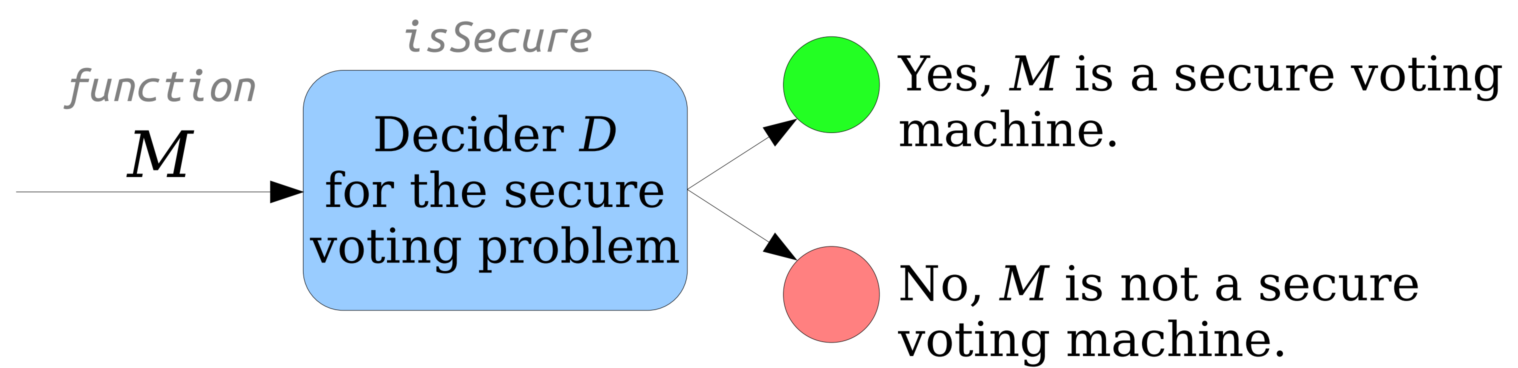
Here, isSecure is just another name for the decider $D$, but with a more descriptive name.

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)
```

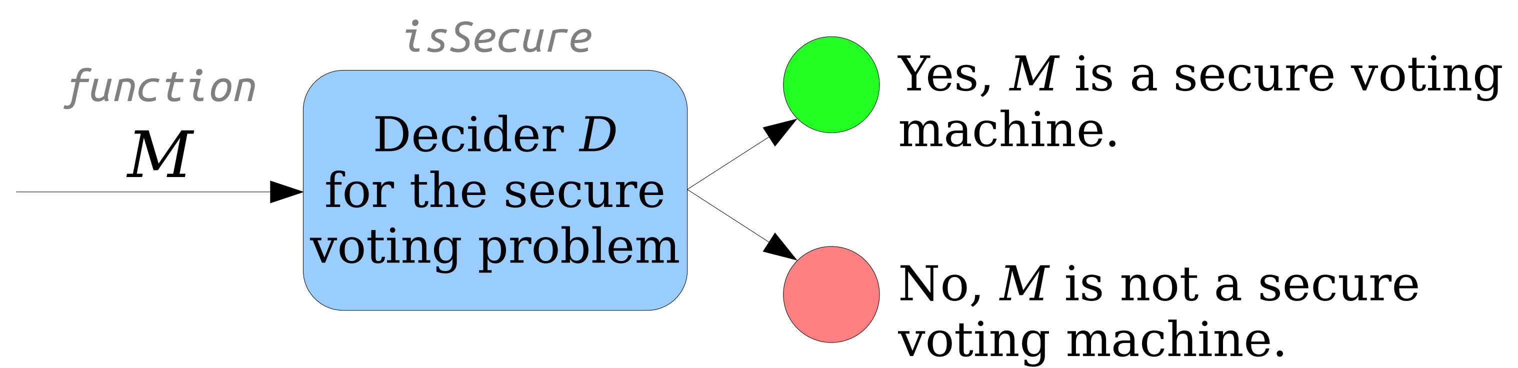
Its argument (**function**) is just a more descriptive name for the TM given as input.

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!




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

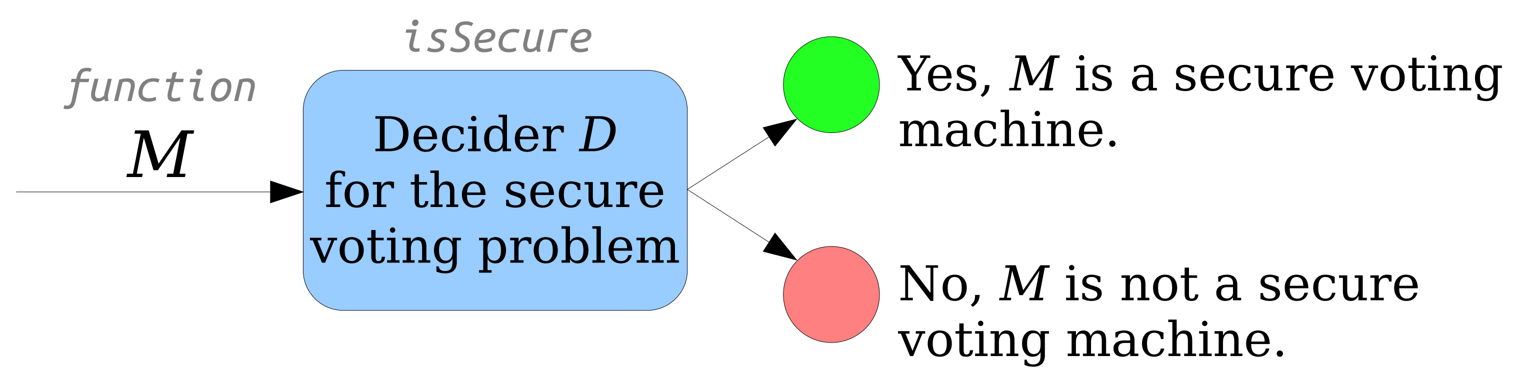
This was the point in the previous proof where we started to write a design spec for some self-referential function **trickster**.

The secure voting problem is decidable.

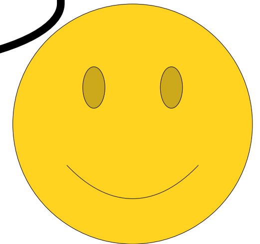
There is a decider $D$ for the secure voting problem

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

Contradiction!




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

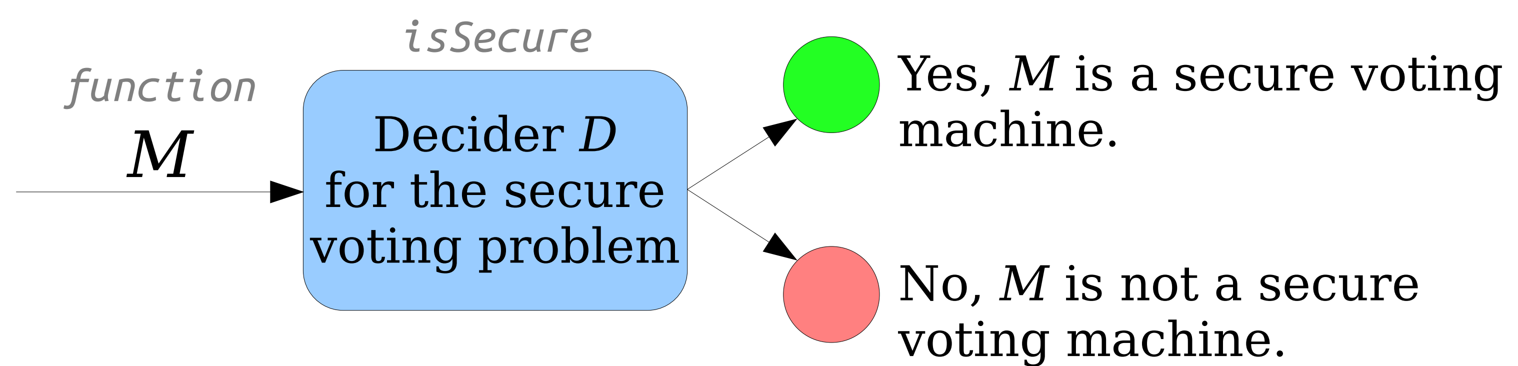


The secure voting problem is decidable.

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)
```

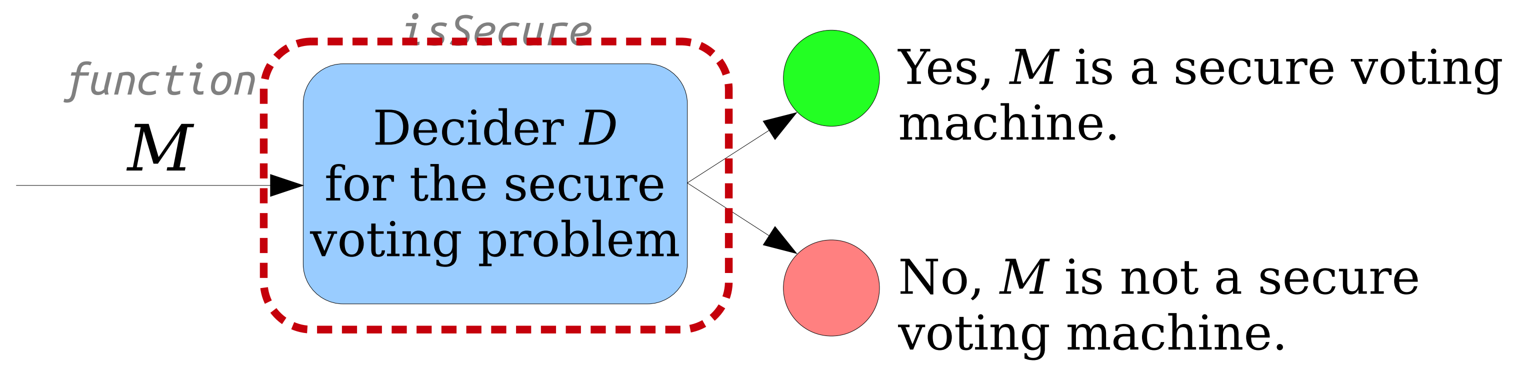
That was a great contradiction to get when we had a decider that would tell us whether a program would accept a given input.

The 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

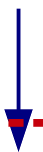
Contradiction!



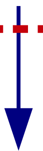

```
bool isSecure(string function)
```

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.

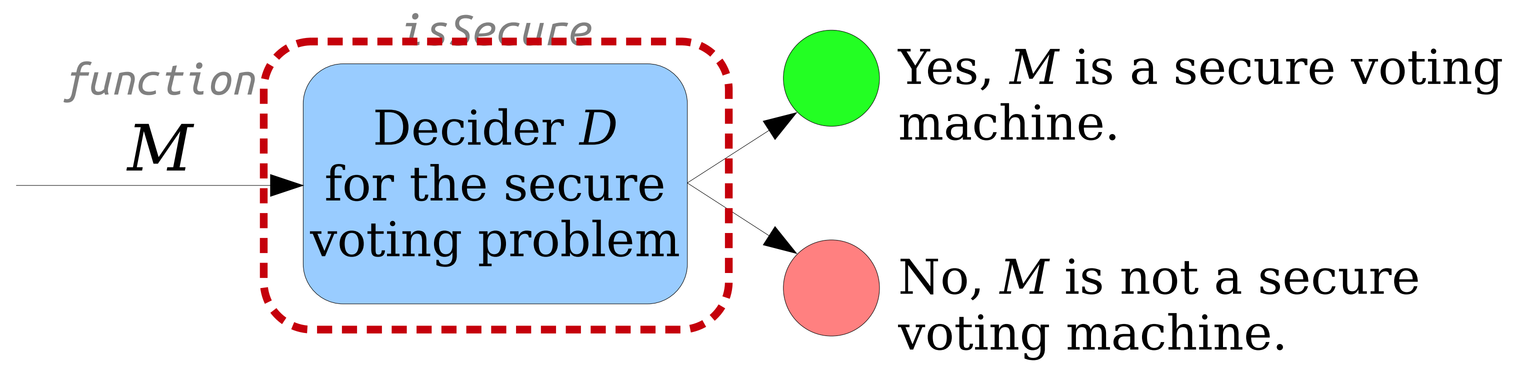


There is a decider $D$ for the secure voting problem

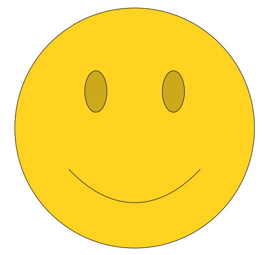


We can write programs that use $D$ as a helper function

Contradiction!




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



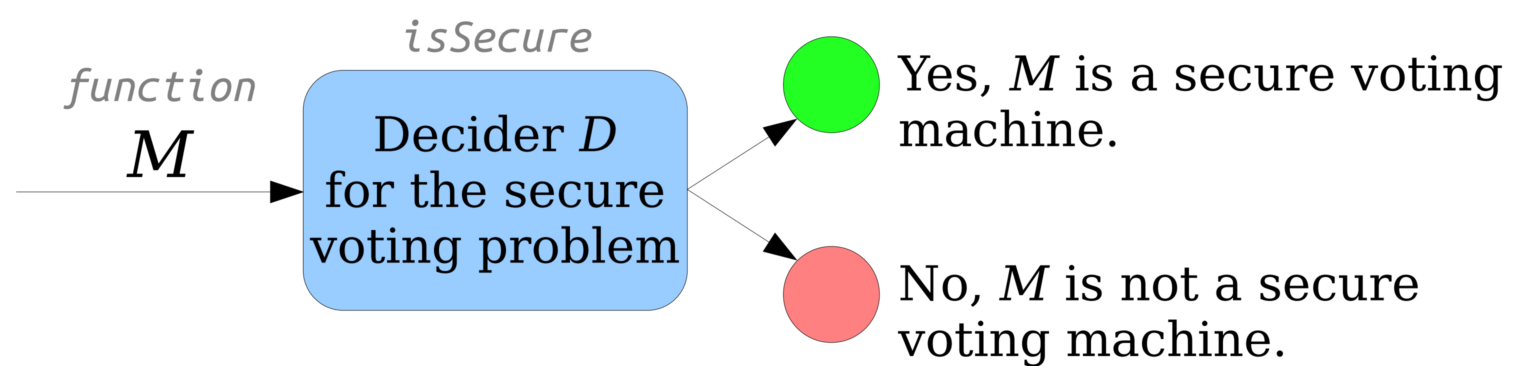
The secure voting problem is decidable.

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

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

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

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.

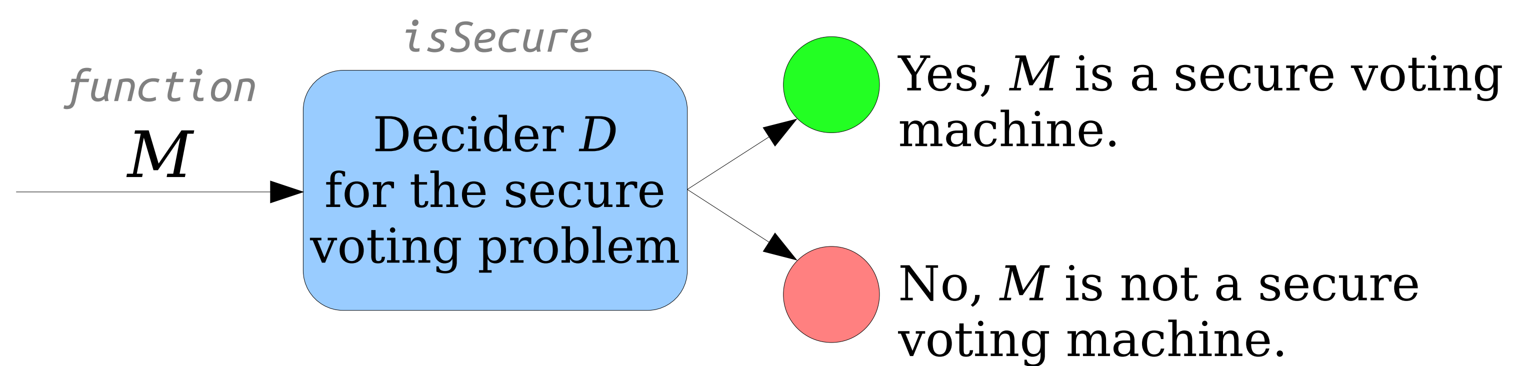
The secure voting problem is decidable.

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

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

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

Contradiction!




```
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.

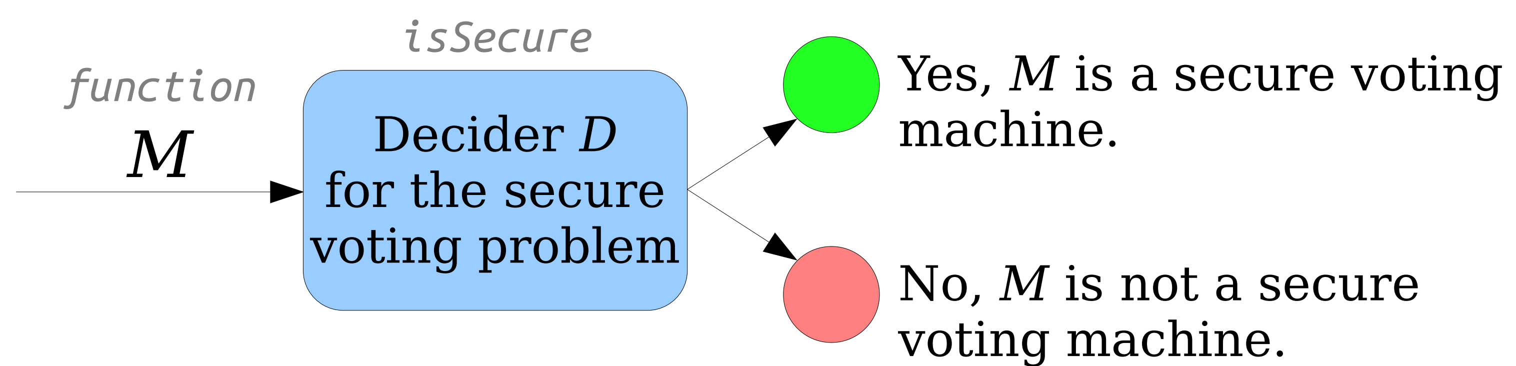
The secure voting problem is decidable.

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.

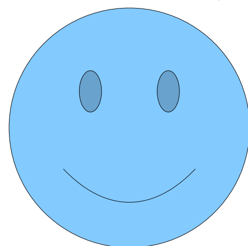
Contradiction!




```
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!



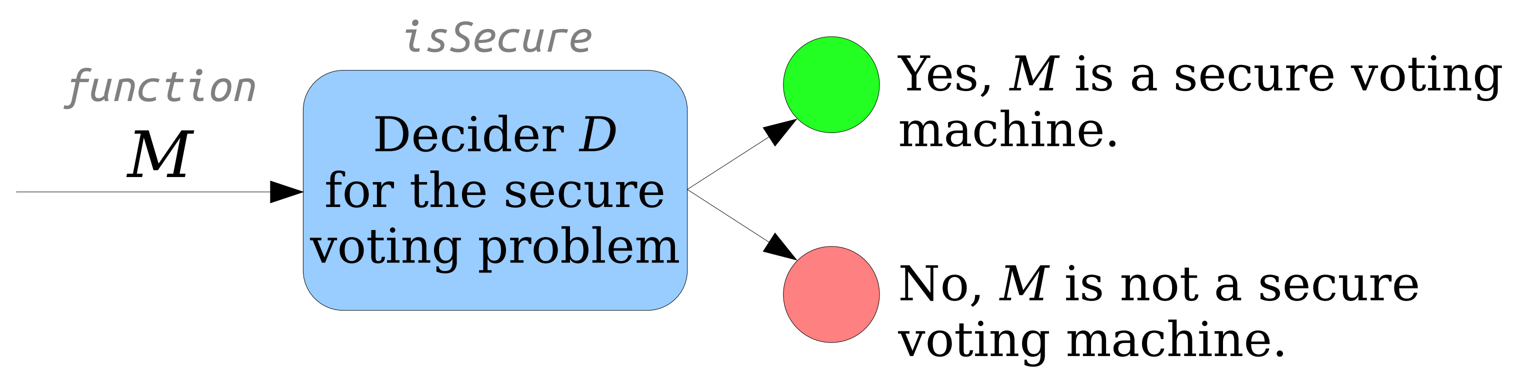
The secure voting problem is decidable.

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!




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

So now we have to figure out how to write this function `trickster`.

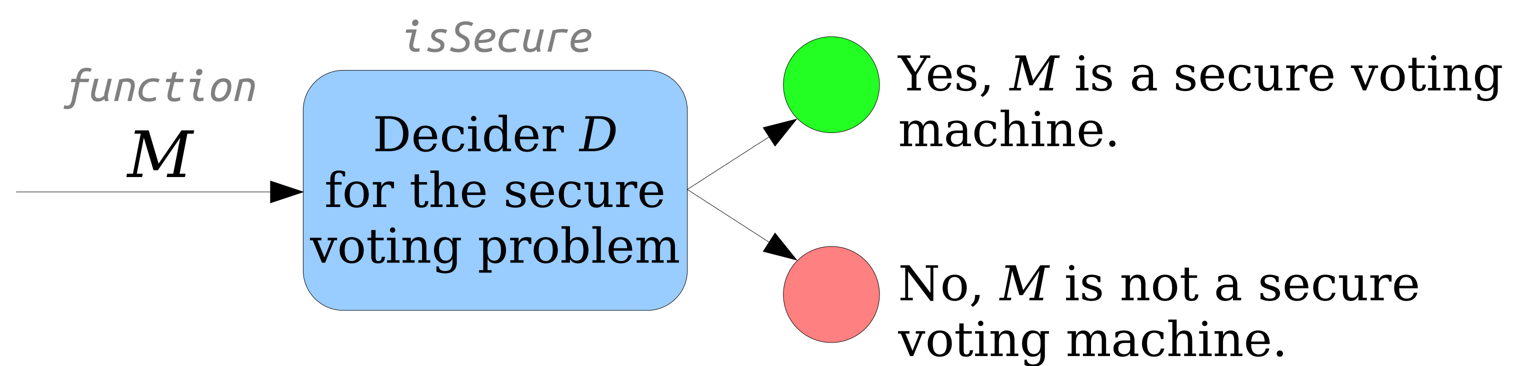
The secure voting problem is decidable.

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

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

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

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.

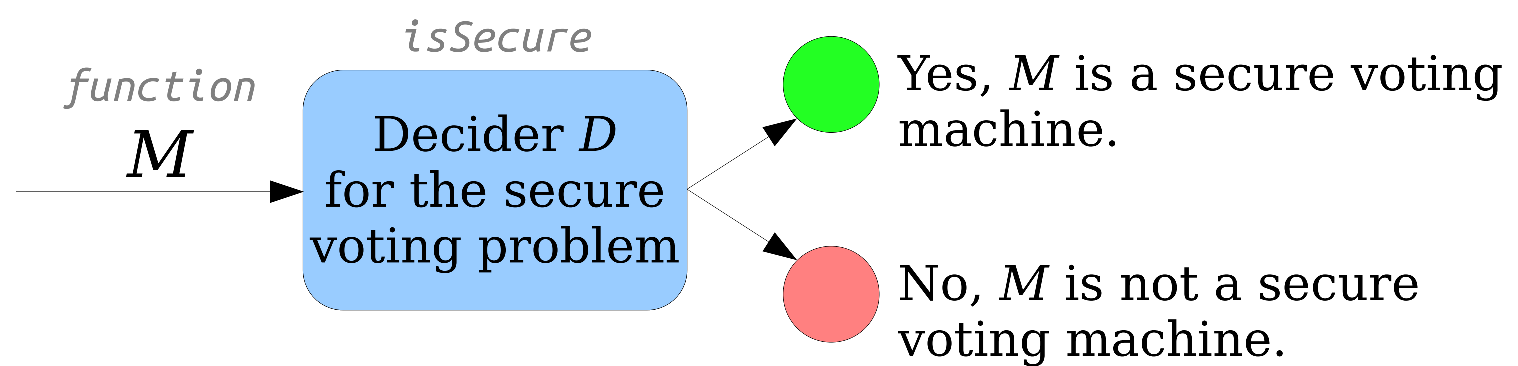
The secure voting problem is decidable.

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

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

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

Contradiction!




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

`trickster` design specification:

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

This first part takes care of the first half of the biconditional.

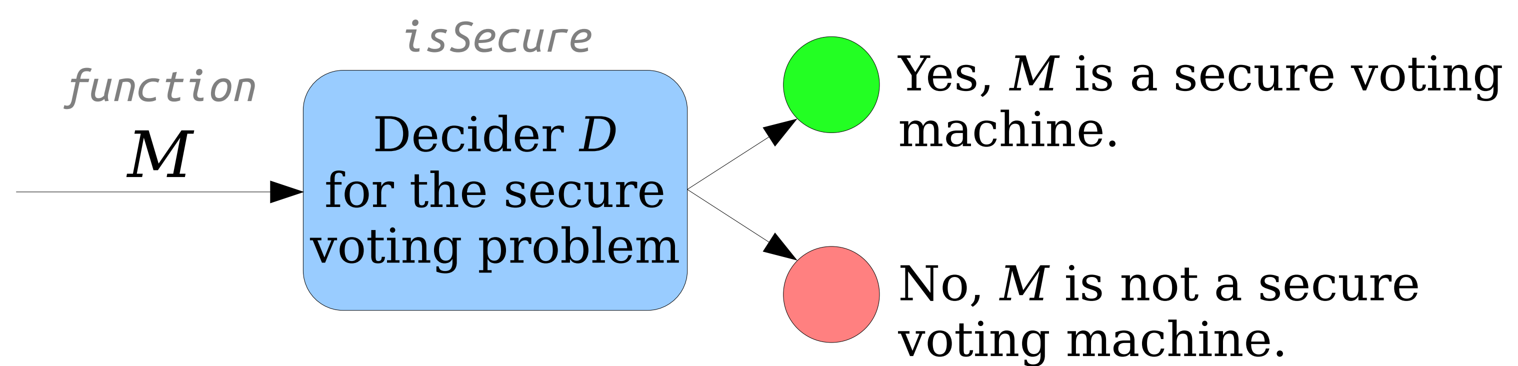
The secure voting problem is decidable.

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

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

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

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.

This second part takes care of the other direction.

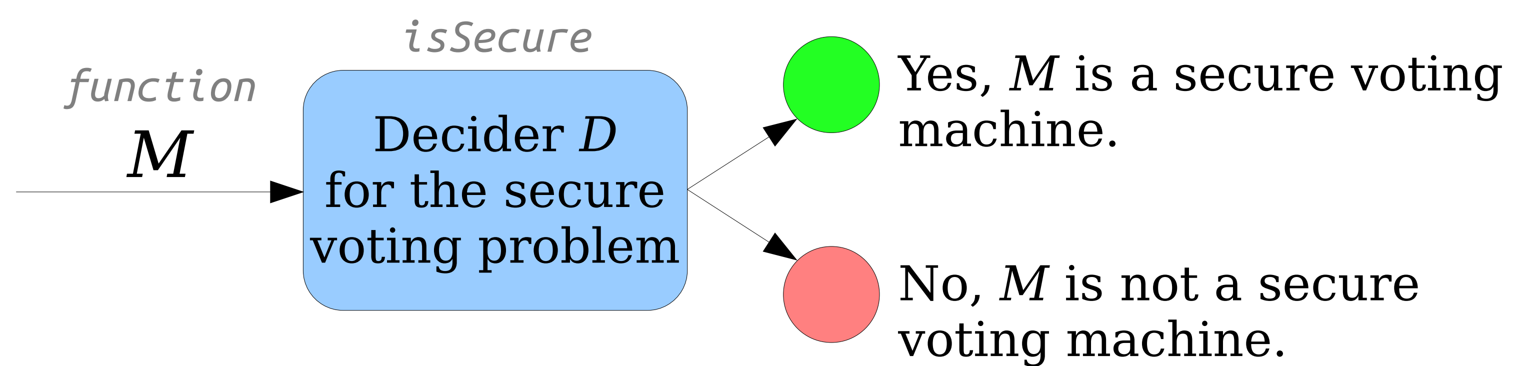
The secure voting problem is decidable.

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!




```
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.

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!

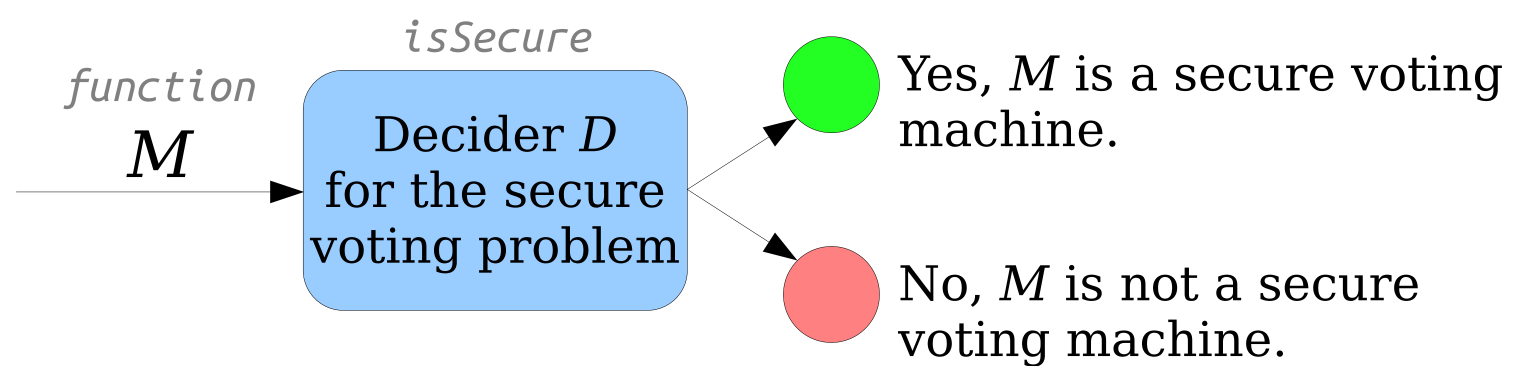
The secure voting problem is decidable.

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!




```
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.

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.

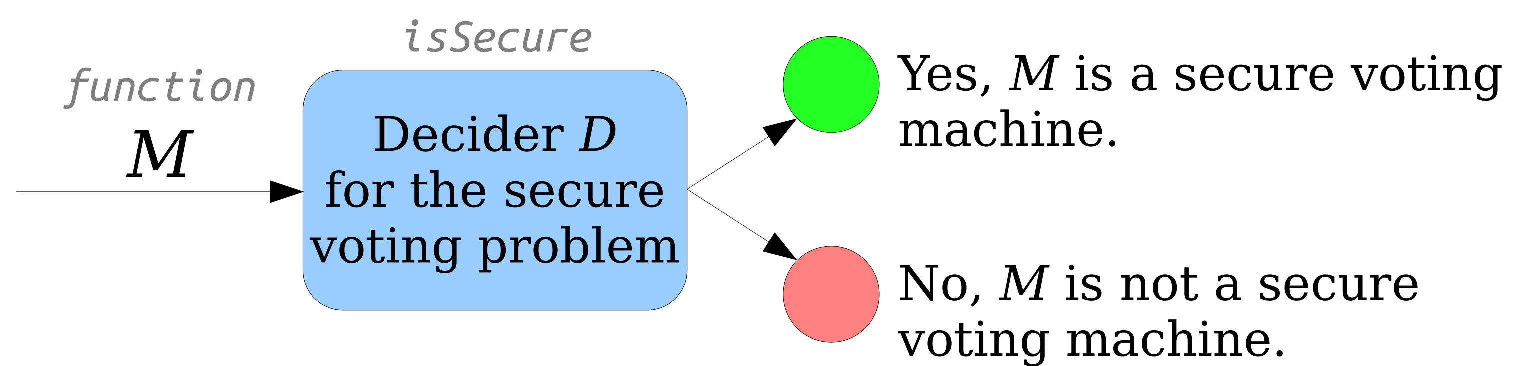
The secure voting problem is decidable.

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

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

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

Contradiction!




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

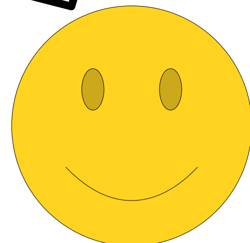
`trickster` design specification:

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) {

}
```

Our `trickster` starts off as a regular boolean function.



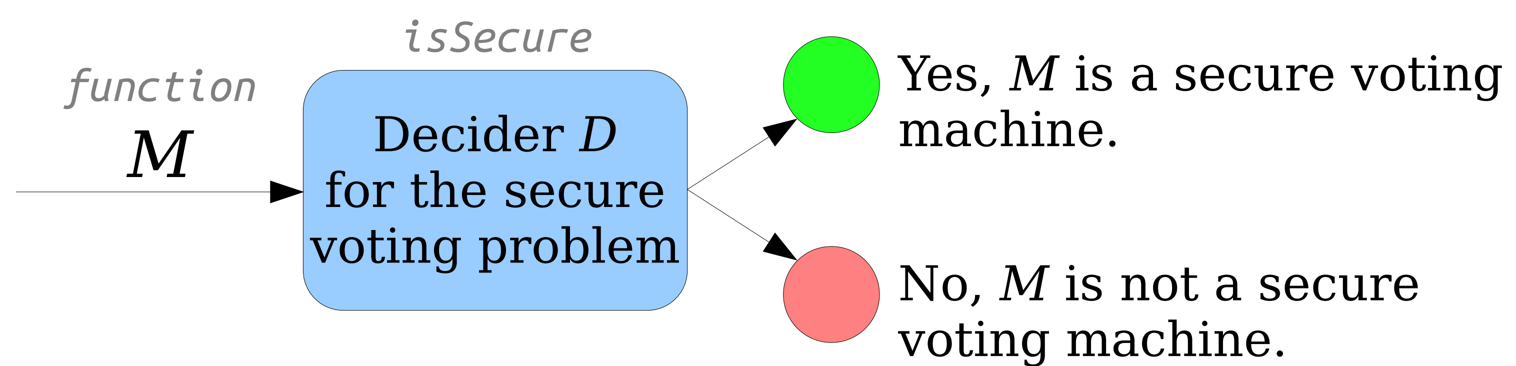
The secure voting problem is decidable.

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

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

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

Contradiction!




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

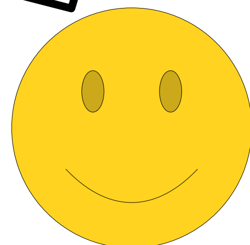
`trickster` design specification:

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) {

}
```

Ultimately, we need to figure out if we're a secure voting machine or not.



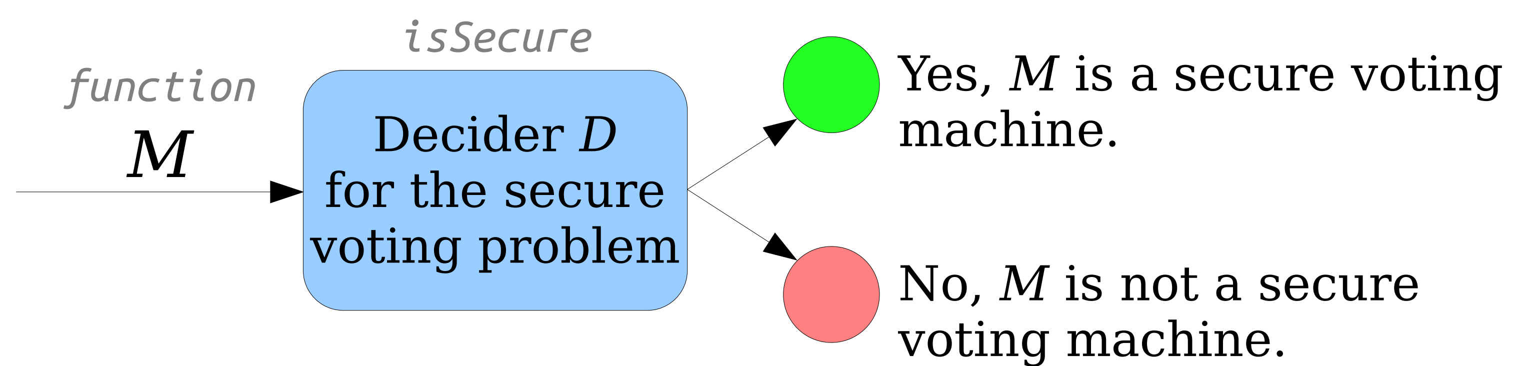
The secure voting problem is decidable.

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!



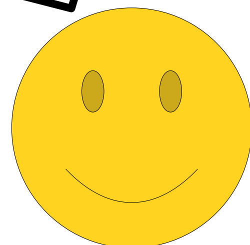

```
bool isSecure(string function)
```

`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
               */;
}
```

The best tool we have for that is some kind of self-reference trick.



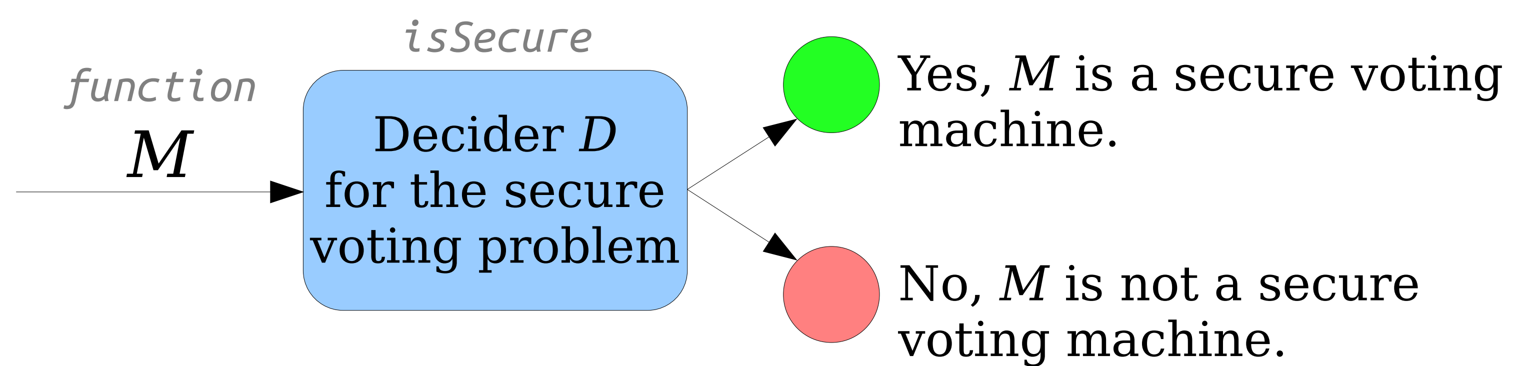
The secure voting problem is decidable.

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!



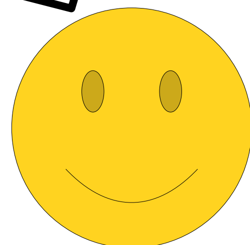

```
bool isSecure(string function)
```

`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
               */;
}
```

As before, we'll use the fact that we have this decider lying around to make `trickster` figure out what exactly it does.



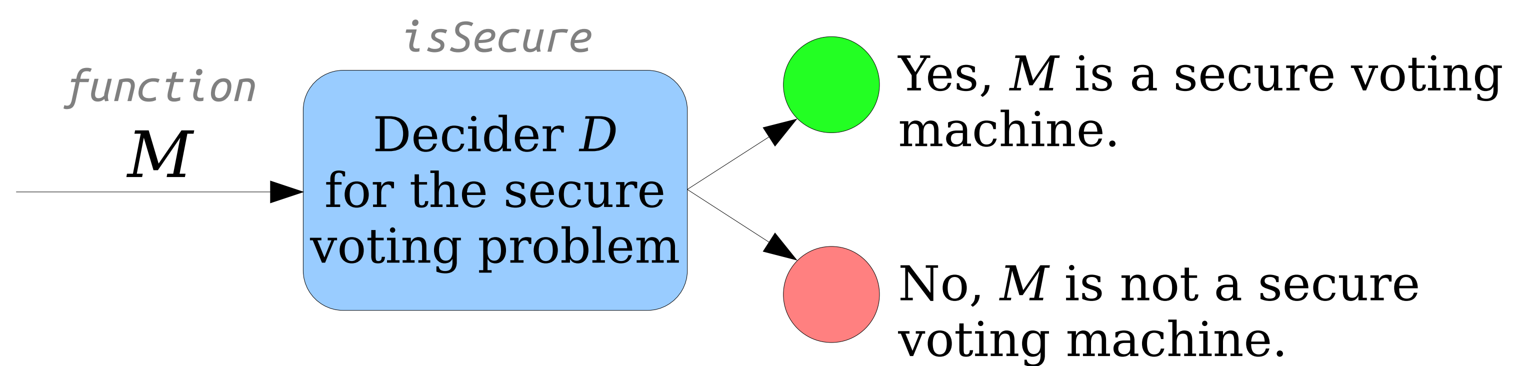
The secure voting problem is decidable.

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

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

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

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.

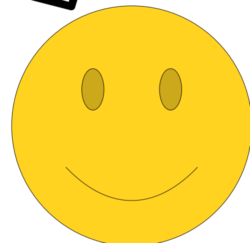
```
bool trickster(string input) {
  string me = /* source code
               * of trickster
               */;

  if (isSecure(me)) {

  } else {

  }
}
```

Specifically, let's have `trickster` ask what it's going to do.



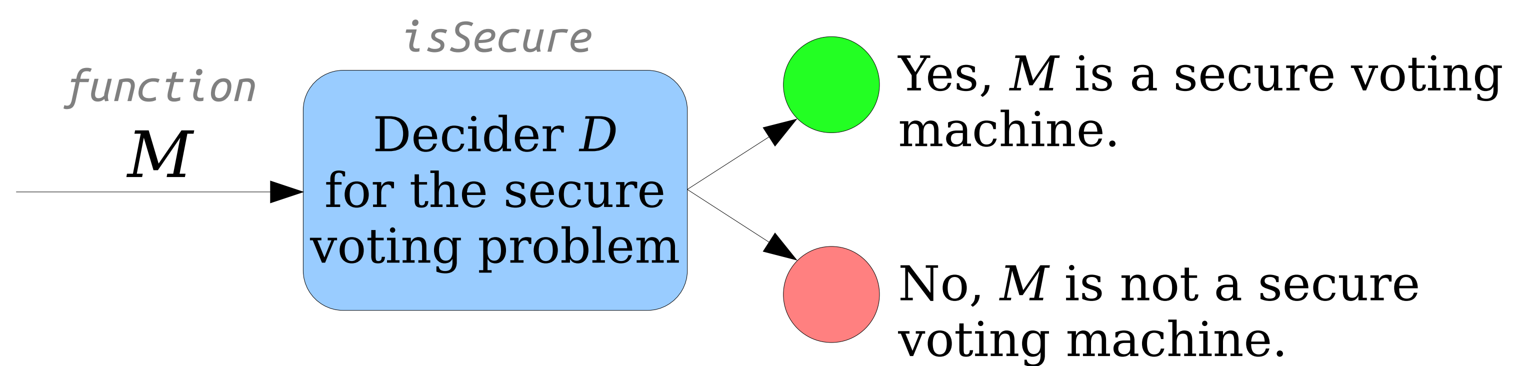
The secure voting problem is decidable.

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

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

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

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.

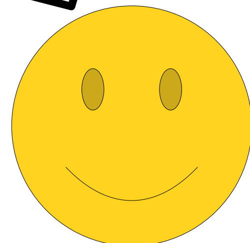
```
bool trickster(string input) {
  string me = /* source code
               * of trickster
               */;

  if (isSecure(me)) {

  } else {

  }
}
```

Let's take it one step at a time.



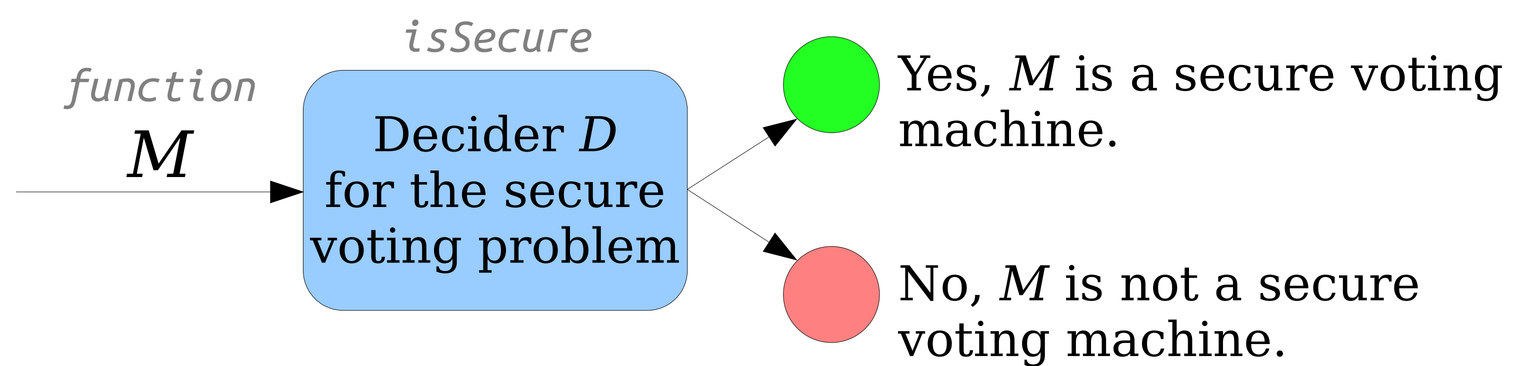
The secure voting problem is decidable.

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

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

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

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.

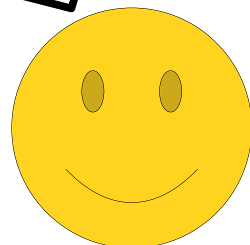
```
bool trickster(string input) {
  string me = /* source code
               * of trickster
               */;

  if (isSecure(me)) {

  } else {

  }
}
```

Oddly enough, let's look at the second requirement first. Why? I ask: why not?



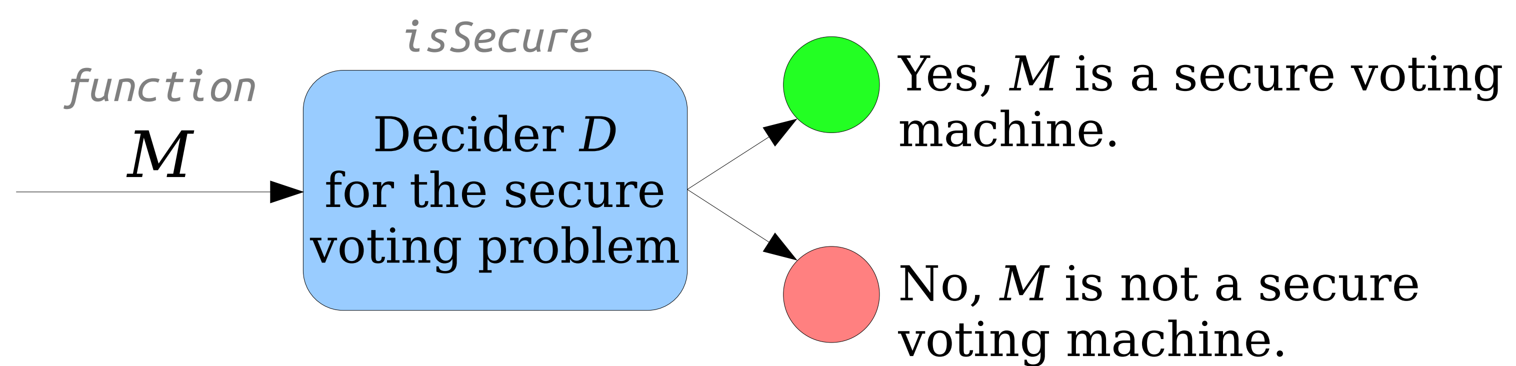
The secure voting problem is decidable.

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!




```
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.

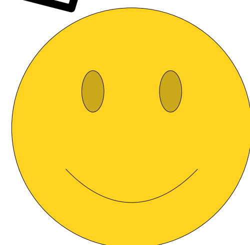
```
bool trickster(string input) {
  string me = /* source code
               * of trickster
               */;

  if (isSecure(me)) {

  } else {

  }
}
```

This requirement says that if trickster is supposed to not be a secure voting machine, then it needs to be a secure voting machine.



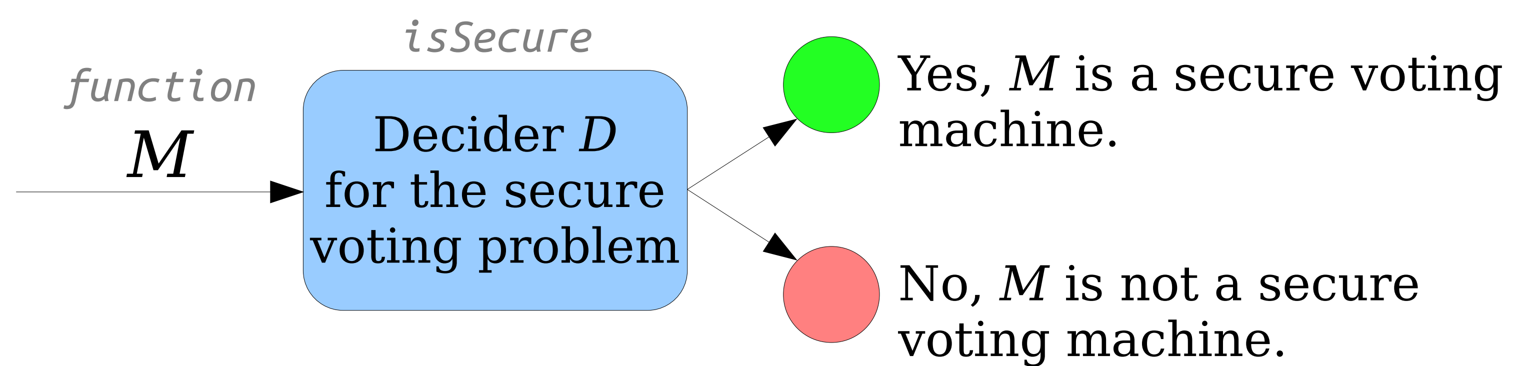
The secure voting problem is decidable.

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

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

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

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.

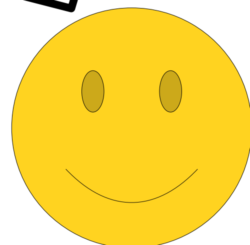
```
bool trickster(string input) {
  string me = /* source code
               * of trickster
               */;

  if (isSecure(me)) {

  } else {

  }
}
```

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.



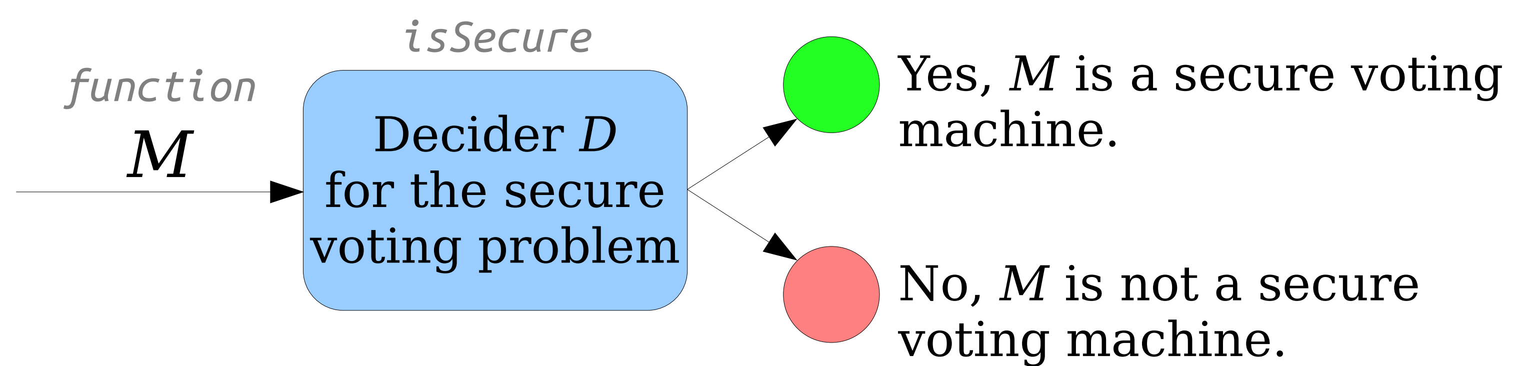
The secure voting problem is decidable.

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

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

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

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.

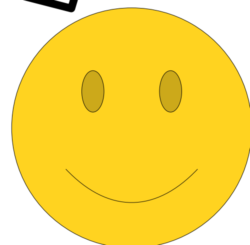
```
bool trickster(string input) {
  string me = /* source code
              * of trickster
              */;

  if (isSecure(me)) {

  } else {

  }
}
```

In this specific case, we're suppose to make **trickster** be a secure voting machine.



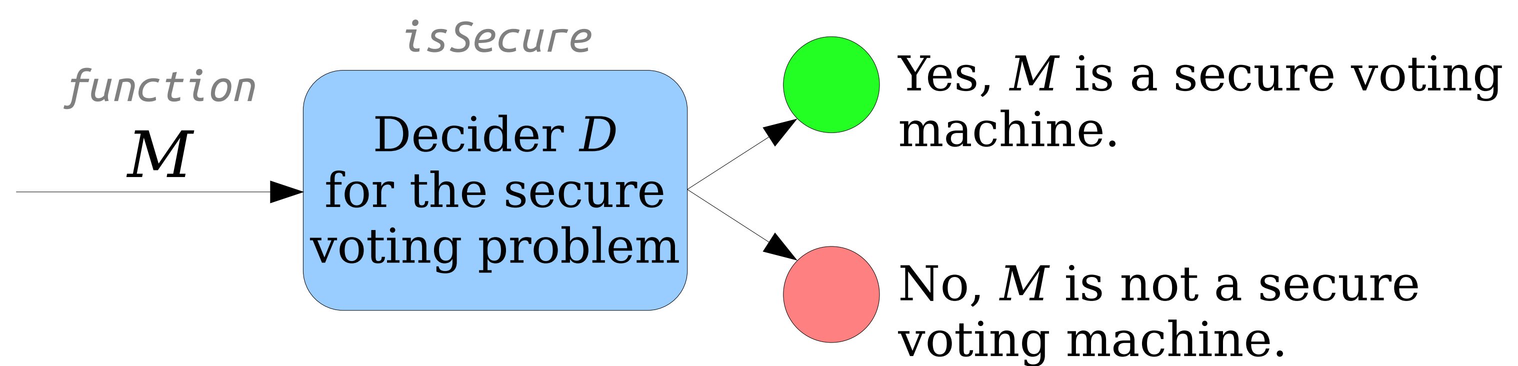
The secure voting problem is decidable.

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

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

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

Contradiction!




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

trickster design specification:

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

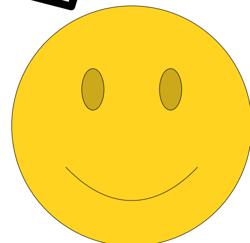
If trickster is not a secure voting machine, then trickster is a secure voting machine.

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

  if (isSecure(me)) {

  } else {

  }
}
```



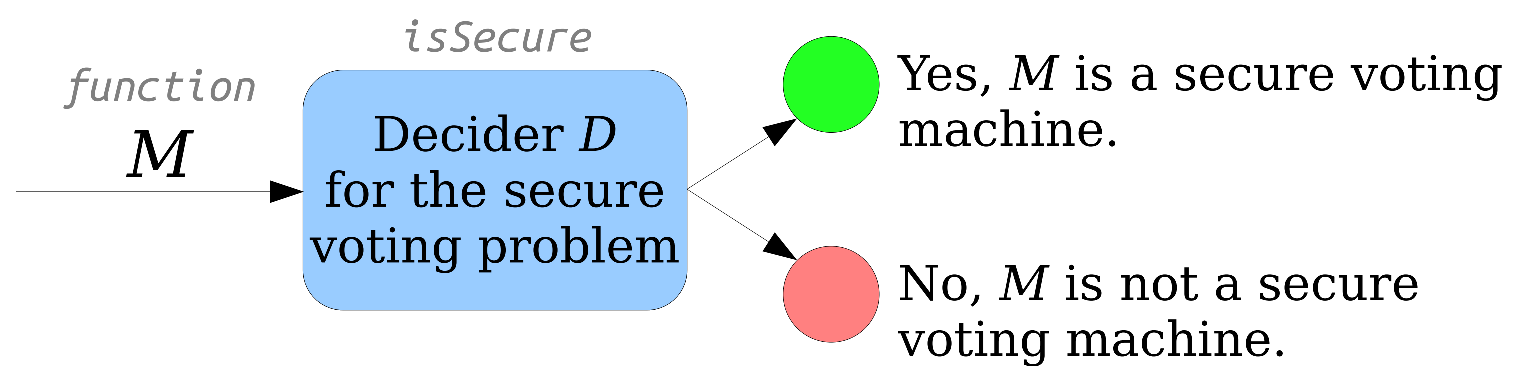
The secure voting problem is decidable.

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!




```
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.

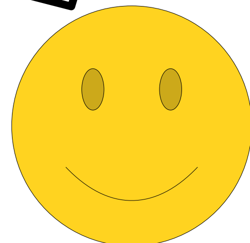
```
bool trickster(string input) {
  string me = /* source code
               * of trickster
               */;

  if (isSecure(me)) {

  } else {

  }
}
```

The good news is that, a while back, we already saw how to do that!



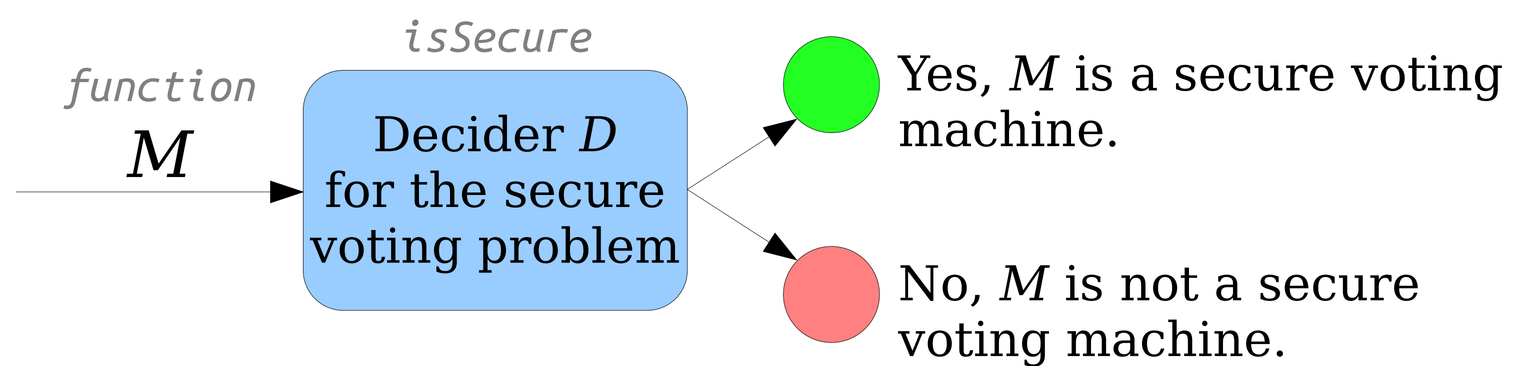
The secure voting problem is decidable.

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!




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

`trickster` design specification:

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

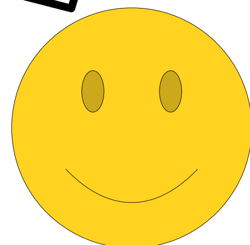
If `trickster` is not a secure voting machine, then `trickster` is a secure voting machine.

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

  if (isSecure(me)) {

  } else {
    return countRsIn(input) >
           countDsIn(input);
  }
}
```

The code looks something like this.



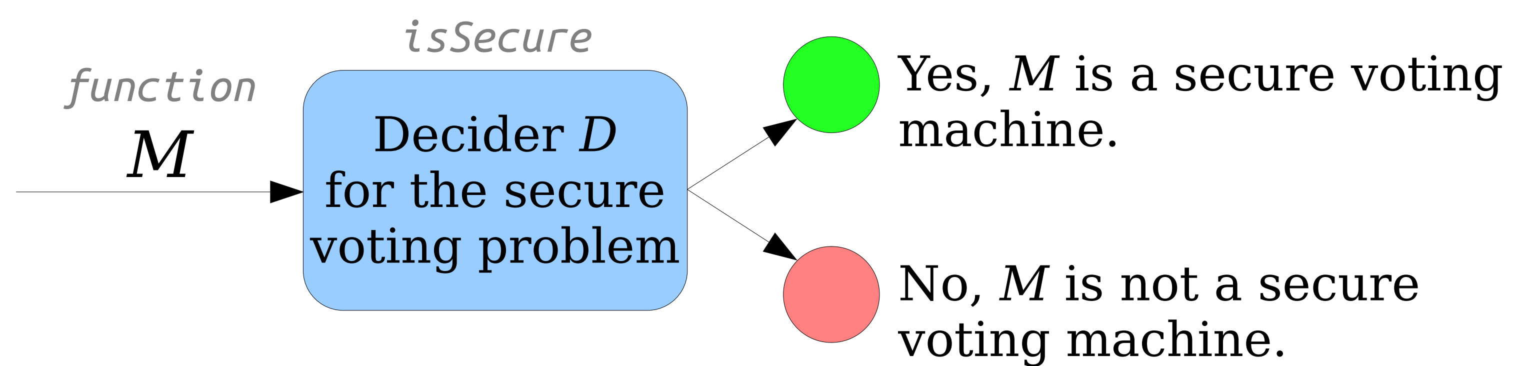
The secure voting problem is decidable.

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!




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

`trickster` design specification:

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

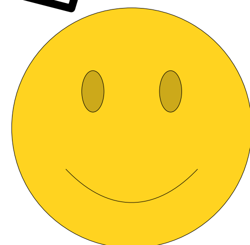
If `trickster` is not a secure voting machine, then `trickster` is a secure voting machine.

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

  if (isSecure(me)) {

  } else {
    return countRsIn(input) >
           countDsIn(input);
  }
}
```

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.



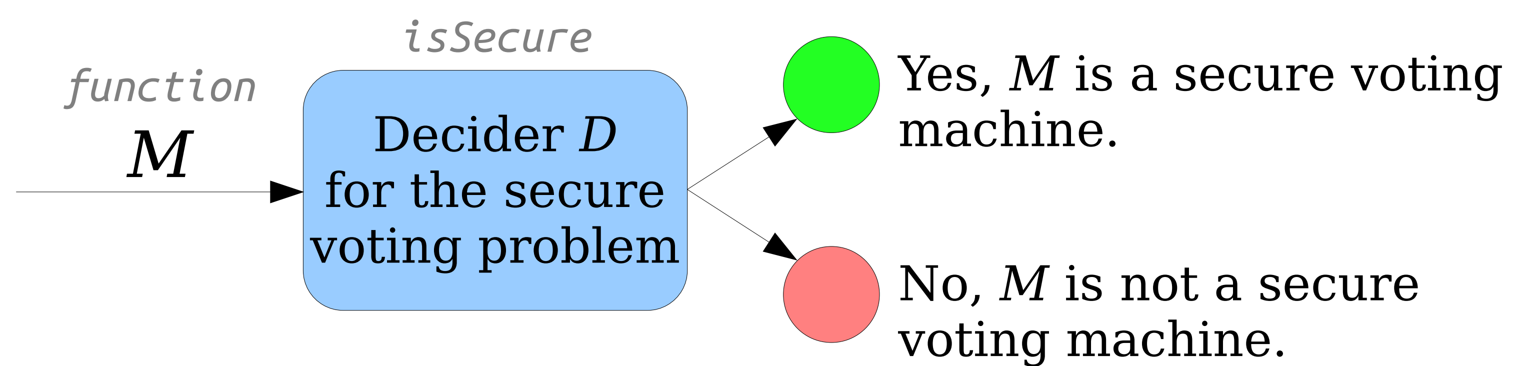
The secure voting problem is decidable.

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

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

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

Contradiction!




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

trickster design specification:

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

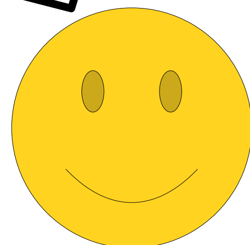
✓ If trickster is not a secure voting machine, then trickster is a secure voting machine.

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

  if (isSecure(me)) {

  } else {
    return countRsIn(input) >
           countDsIn(input);
  }
}
```

Okay! So that's one of two requirements down.



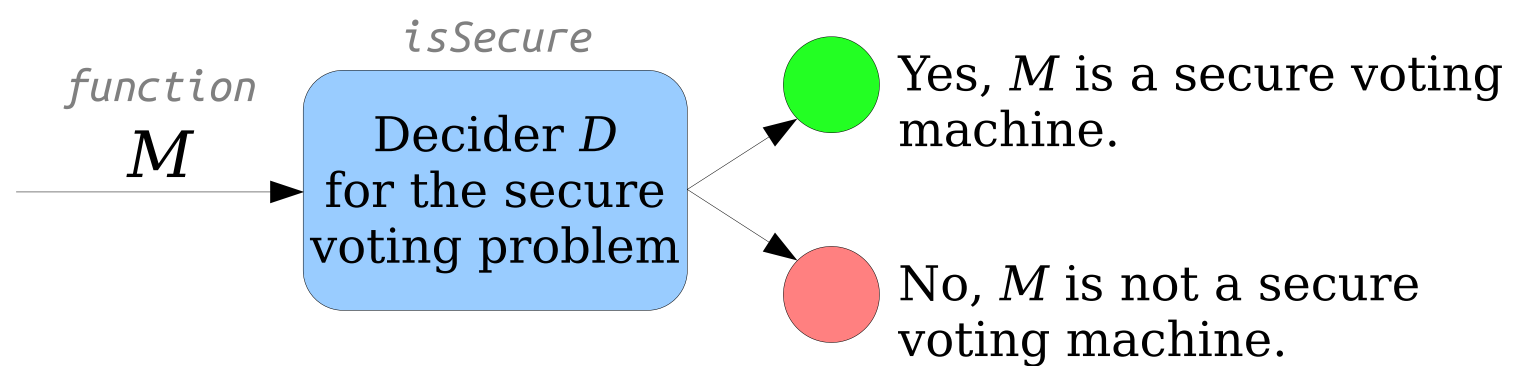
The secure voting problem is decidable.

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!




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

`trickster` design specification:

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

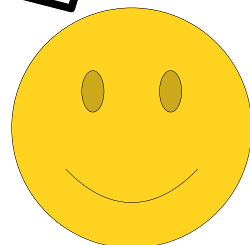
✓ If `trickster` is not a secure voting machine, then `trickster` is a secure voting machine.

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

  if (isSecure(me)) {

  } else {
     return countRsIn(input) >
            countDsIn(input);
  }
}
```

Let's move on to the other one.



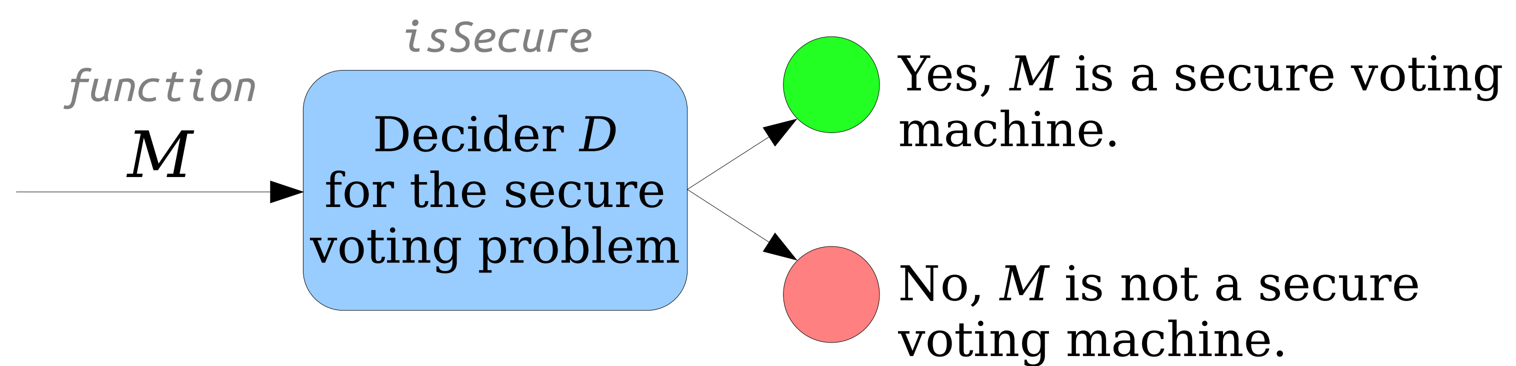
The secure voting problem is decidable.

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!




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

trickster design specification:

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

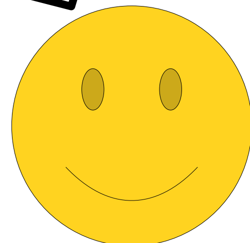
✓ If trickster is not a secure voting machine, then
trickster is a secure voting machine.

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

  if (isSecure(me)) {

  } else {
     return countRsIn(input) >
            countDsIn(input);
  }
}
```

This says that if trickster is supposed to be a secure voting machine, it needs to not be a secure voting machine.



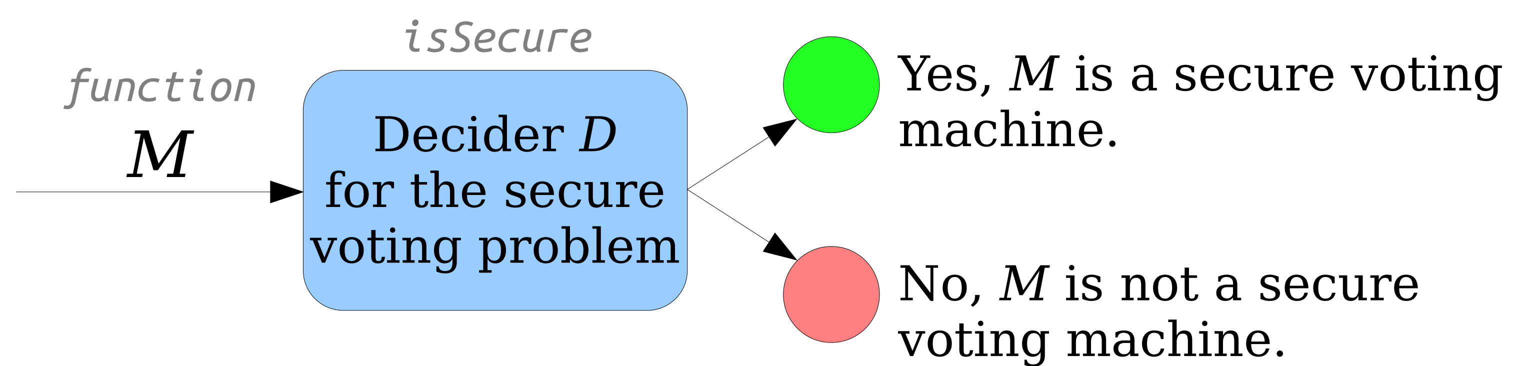
The secure voting problem is decidable.

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

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

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

Contradiction!




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

`trickster` design specification:

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

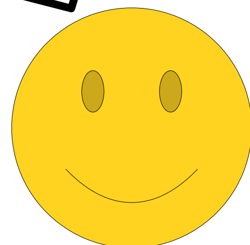
✓ If `trickster` is not a secure voting machine, then `trickster` is a secure voting machine.

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

  if (isSecure(me)) {

  } else {
     return countRsIn(input) >
            countDsIn(input);
  }
}
```

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



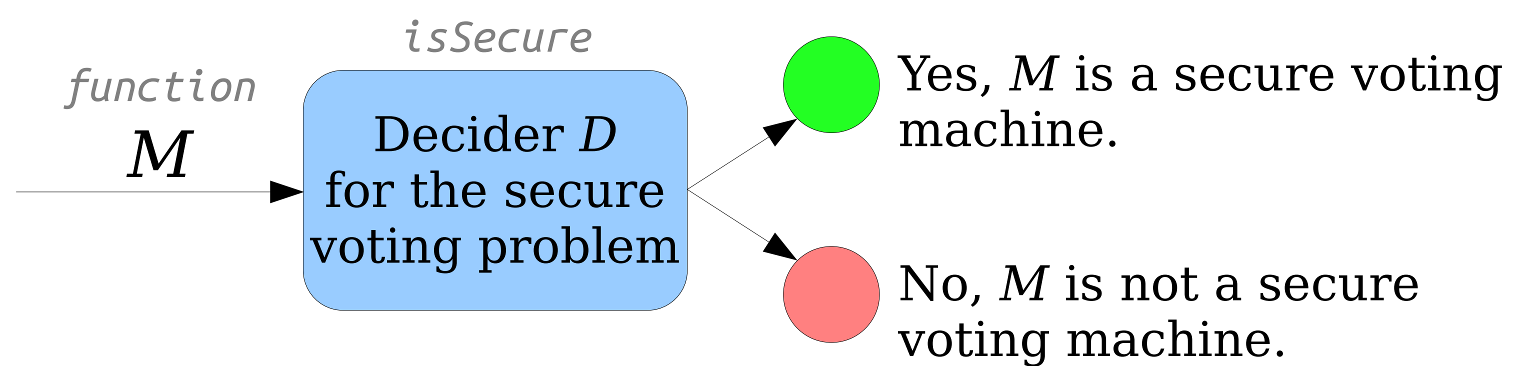
The secure voting problem is decidable.

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

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

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

Contradiction!




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

`trickster` design specification:

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

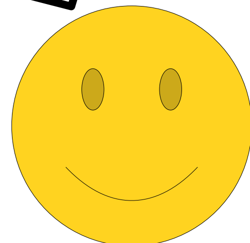
✓ If `trickster` is not a secure voting machine, then `trickster` is a secure voting machine.

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

  if (isSecure(me)) {

  } else {
     return countRsIn(input) >
            countDsIn(input);
  }
}
```

We can literally do anything we want *except* accepting all strings with more r's than d's and not accepting anything else.



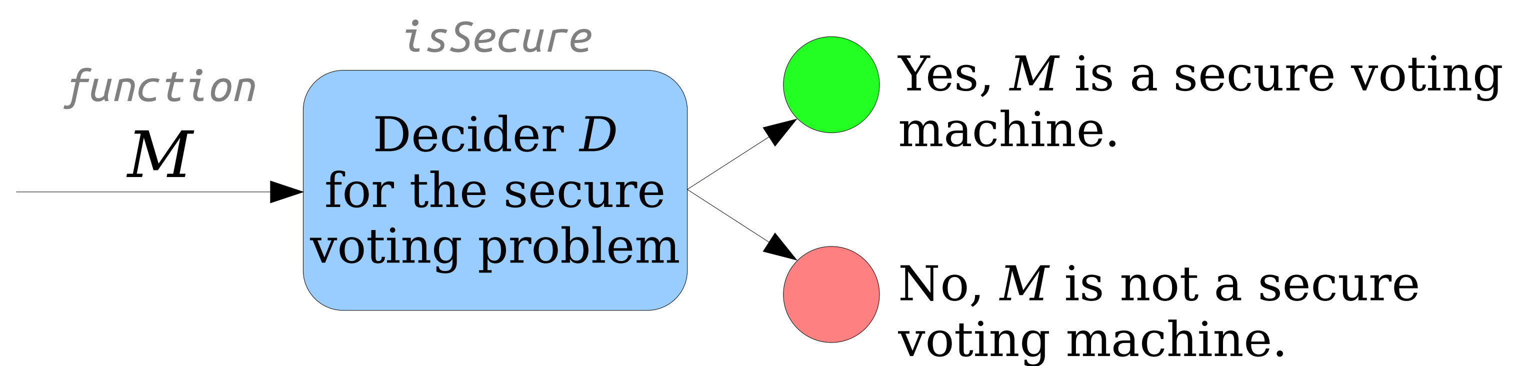
The secure voting problem is decidable.

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!




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

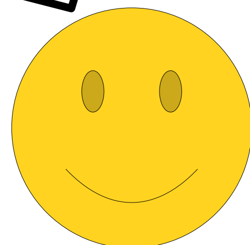
trickster design specification:

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)) {
     return true;
  } else {
     return countRsIn(input) >
            countDsIn(input);
  }
}
```



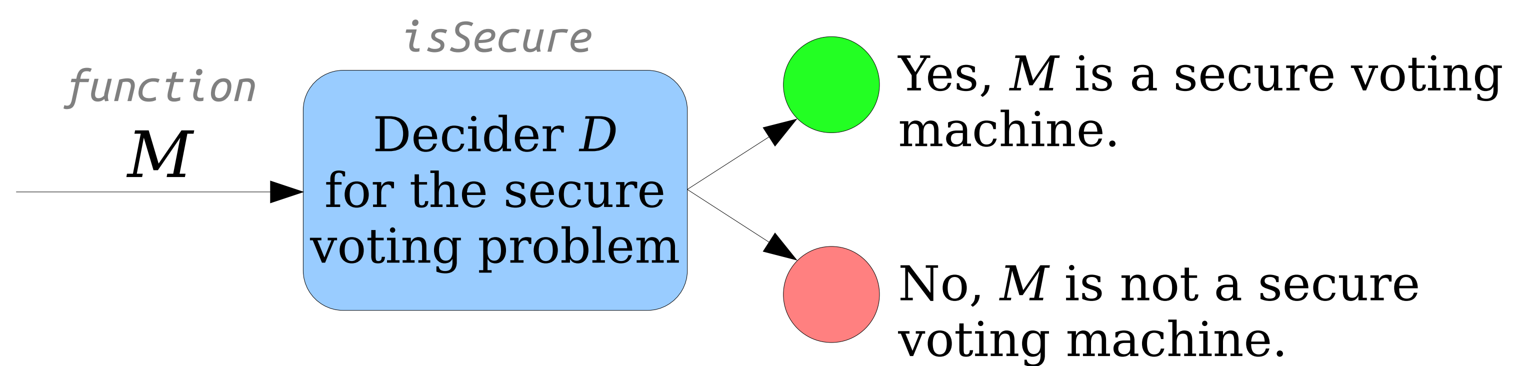
The secure voting problem is decidable.

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

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

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

Contradiction!




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

`trickster` design specification:

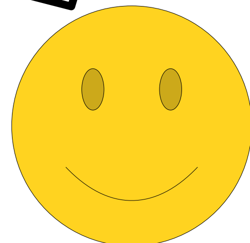
If `trickster` is a secure voting machine, then `trickster` is not a secure voting machine.

√ 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)) {
    return true;
  } else {
    return countRsIn(input) >
           countDsIn(input);
  }
}
```

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.



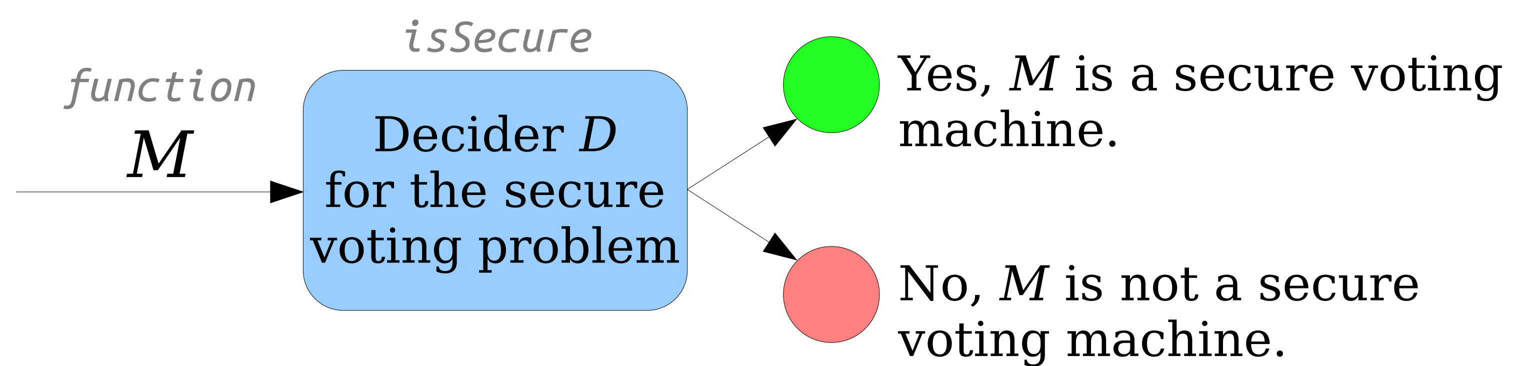
The secure voting problem is decidable.

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

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

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

Contradiction!




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

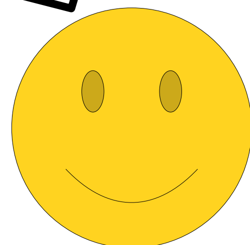
`trickster` design specification:

- ✓ 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)) {
     return true;
  } else {
     return countRsIn(input) >
            countDsIn(input);
  }
}
```

So we're done with this part of the design!



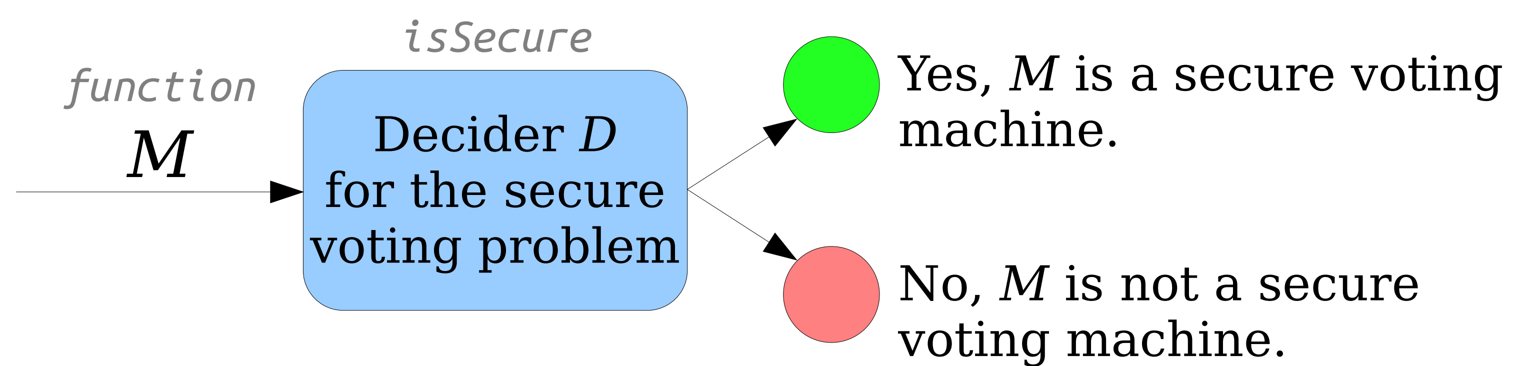
The secure voting problem is decidable.

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

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

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

Contradiction!




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

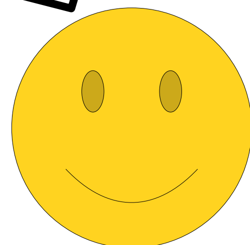
`trickster` design specification:

- ✓ 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)) {
     return true;
  } else {
     return countRsIn(input) >
            countDsIn(input);
  }
}
```

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!



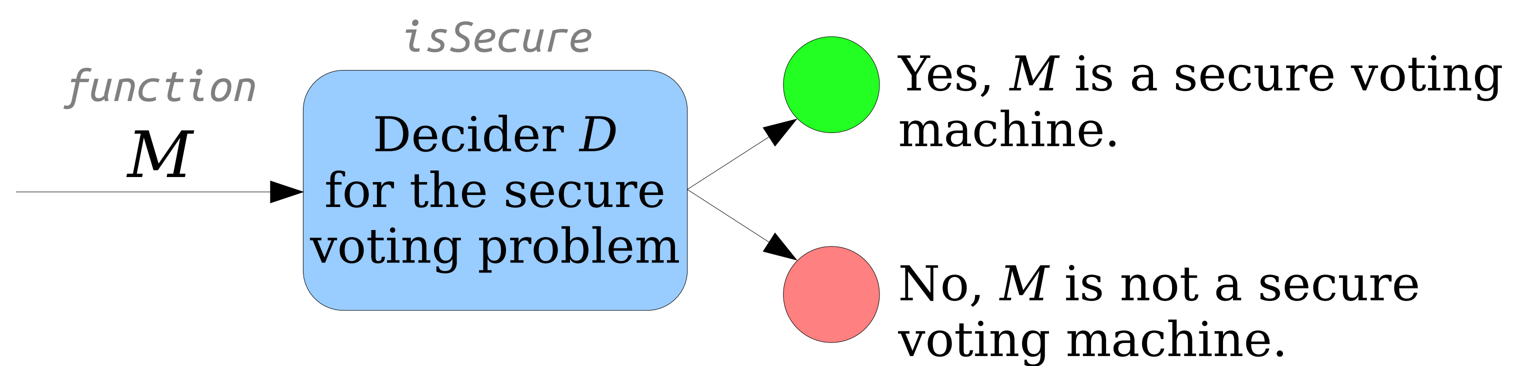
The secure voting problem is decidable.

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

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

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

Contradiction!




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

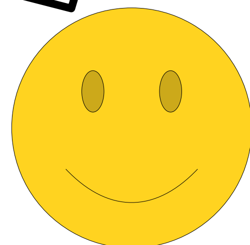
trickster design specification:

- ✓ 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)) {
     return true;
  } else {
     return countRsIn(input) >
            countDsIn(input);
  }
}
```

That gives us the contradiction that we needed to get.



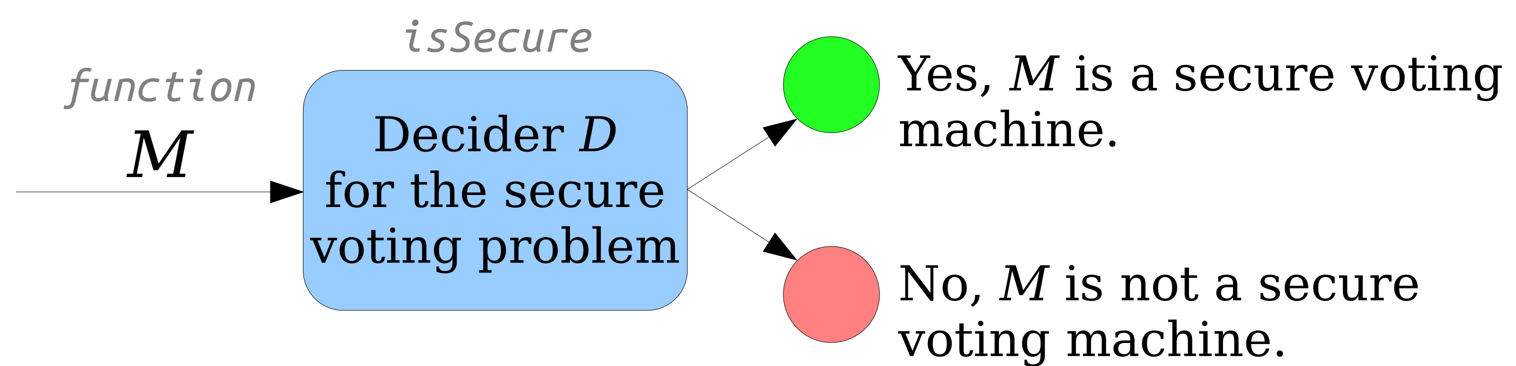
The secure voting problem is decidable.

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!




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

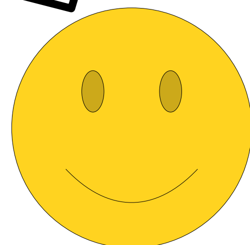
trickster design specification:

- ✓ 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)) {
     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.



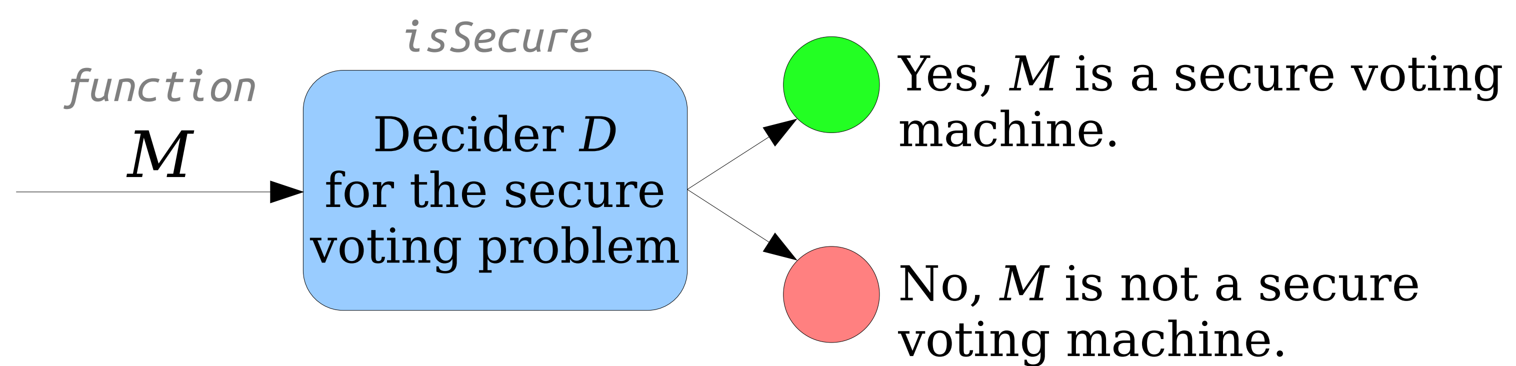
The secure voting problem is decidable.

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

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

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

Contradiction!




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

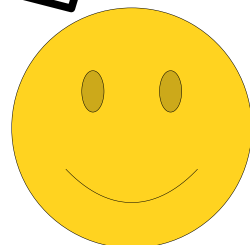
trickster design specification:

- ✓ 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)) {
    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!



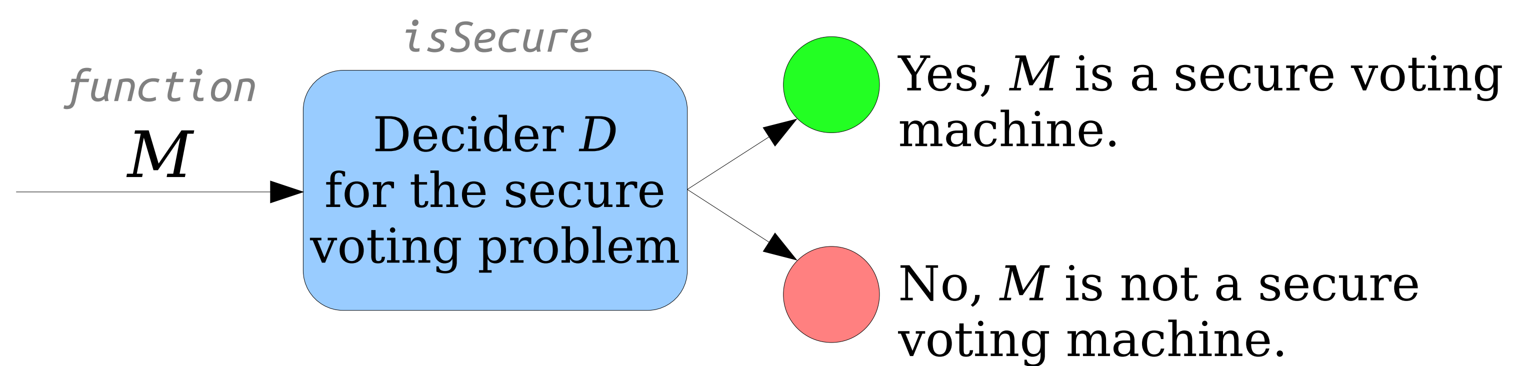
The secure voting problem is decidable.

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

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

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

Contradiction!




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

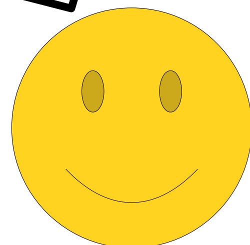
`trickster` design specification:

- √ 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)) {
     return true;
  } else {
     return countRsIn(input) >
            countDsIn(input);
  }
}
```

There can be all sorts of programs that meet the design specification we set out above.



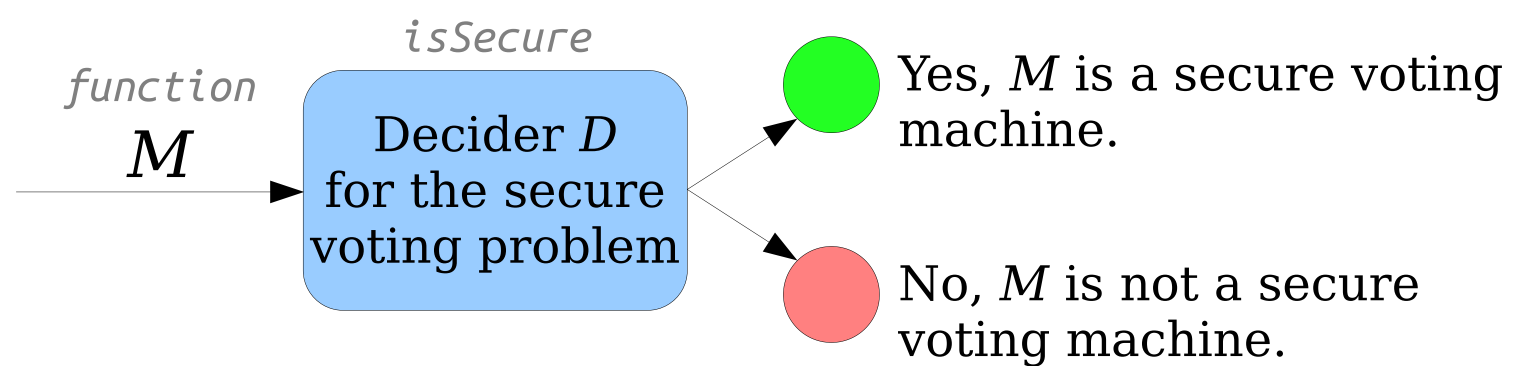
The secure voting problem is decidable.

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

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

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

Contradiction!




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

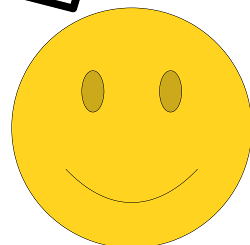
`trickster` design specification:

- √ 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)) {
    return true;
  } else {
    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.



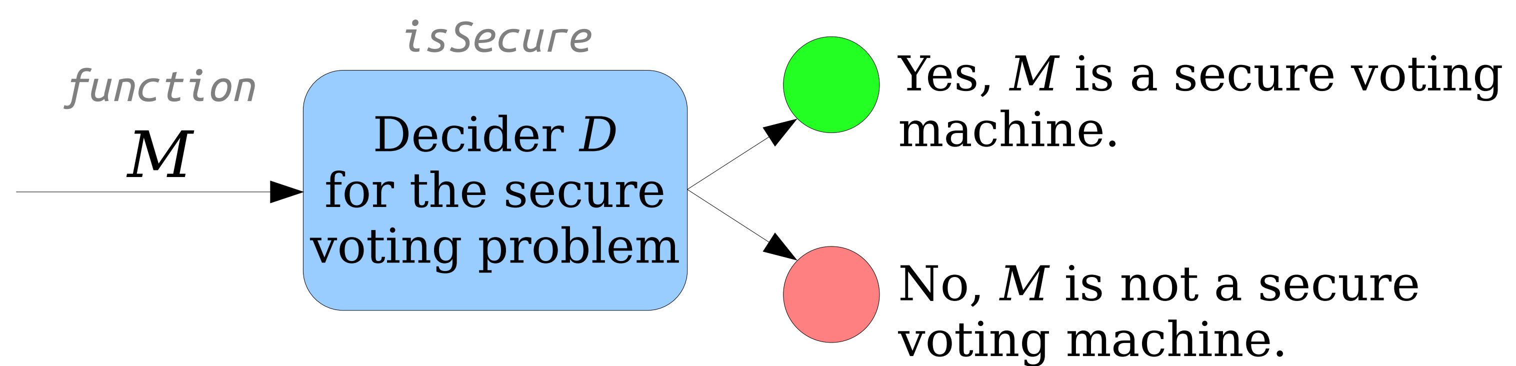
The secure voting problem is decidable.

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

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

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

Contradiction!




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

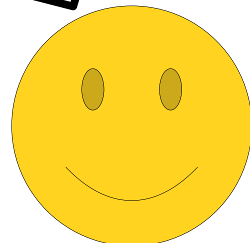
trickster design specification:

- ✓ 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)) {
     return true;
  } else {
     return countRsIn(input) >
            countDsIn(input);
  }
}
```

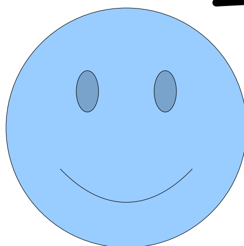
As long as you meet the design criteria, you should be good to go!



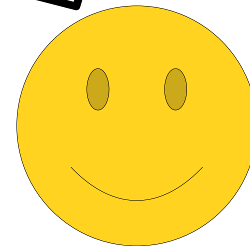
Let's take a minute to review the general process that we followed to get these results to work.

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.

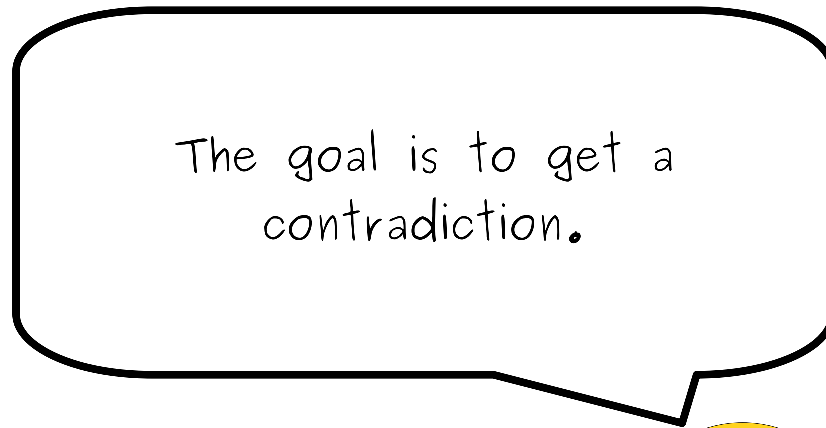


The problem in question is decidable

Start off by assuming it's decidable.

The problem in question is decidable

Contradiction!



The problem in question is decidable

Contradiction!

The problem in question is decidable

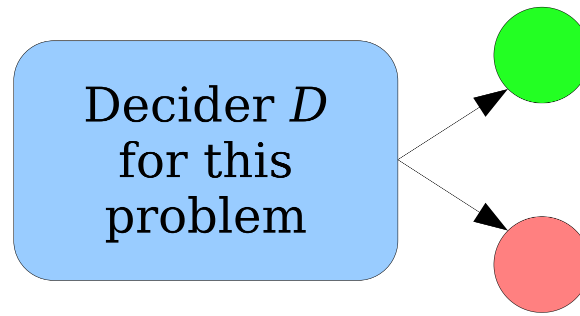
There is a decider *D* for that problem.

Contradiction!

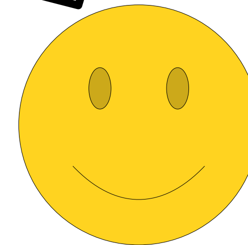
...the first step is to suppose that you have a decider for the language in question.

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.

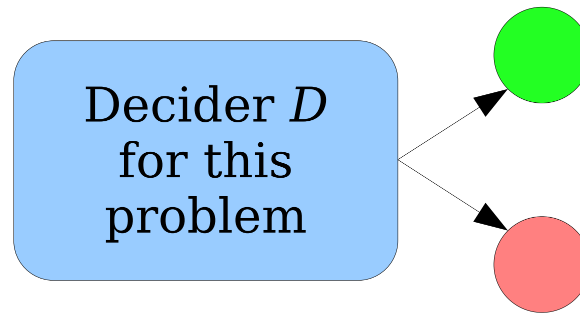


Contradiction!

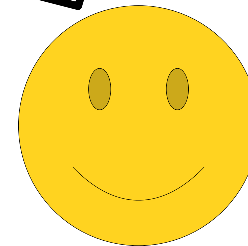
The problem in question is decidable

There is a decider $D$ for that problem.

Contradiction!

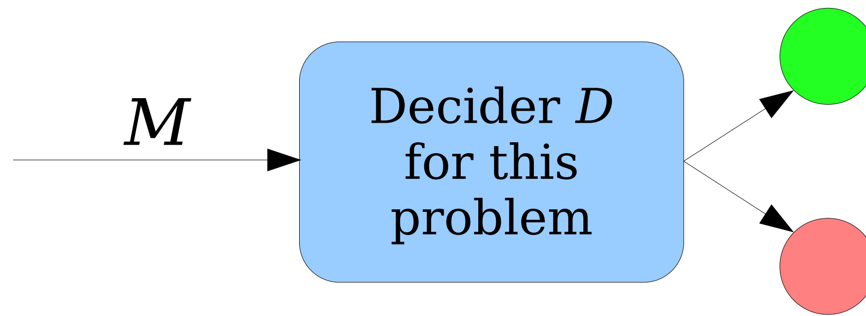



Think about what the inputs to the decider are going to look like. That depends on the language.

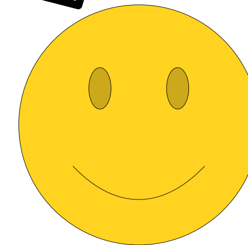


The problem in question is decidable

There is a decider *D* for that problem.



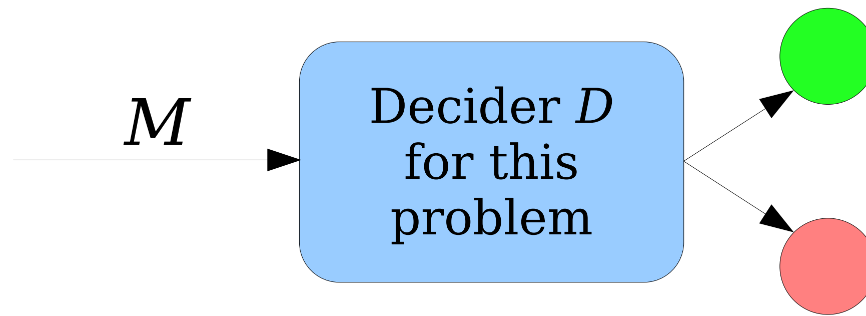

In the cases we're exploring in this class, there will always be at least one input that's a TM of some sort.



Contradiction!

The problem in question is decidable

There is a decider $D$ for that problem.

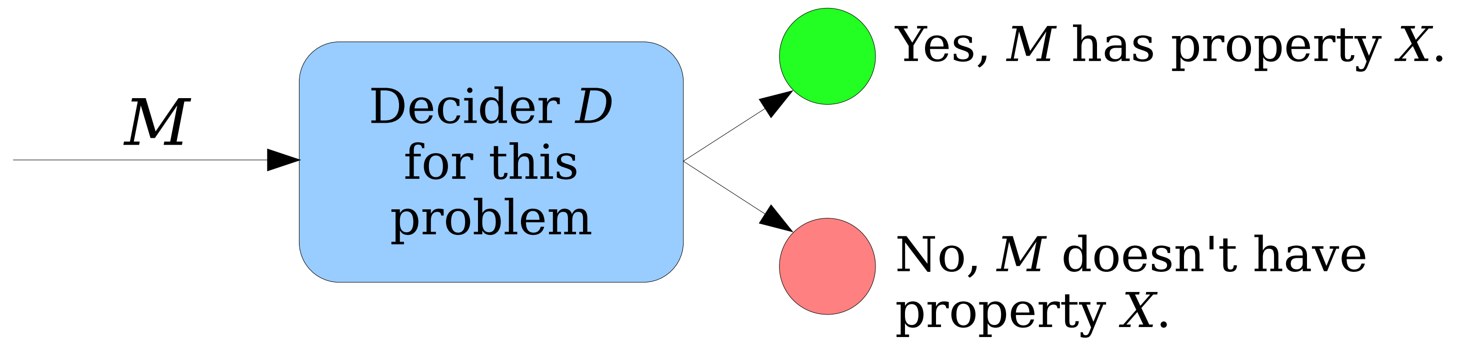



Next, think about what the decider is going to tell you about those inputs. That depends on the problem at hand.

Contradiction!

The problem in question is decidable

There is a decider $D$ for that problem.




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.

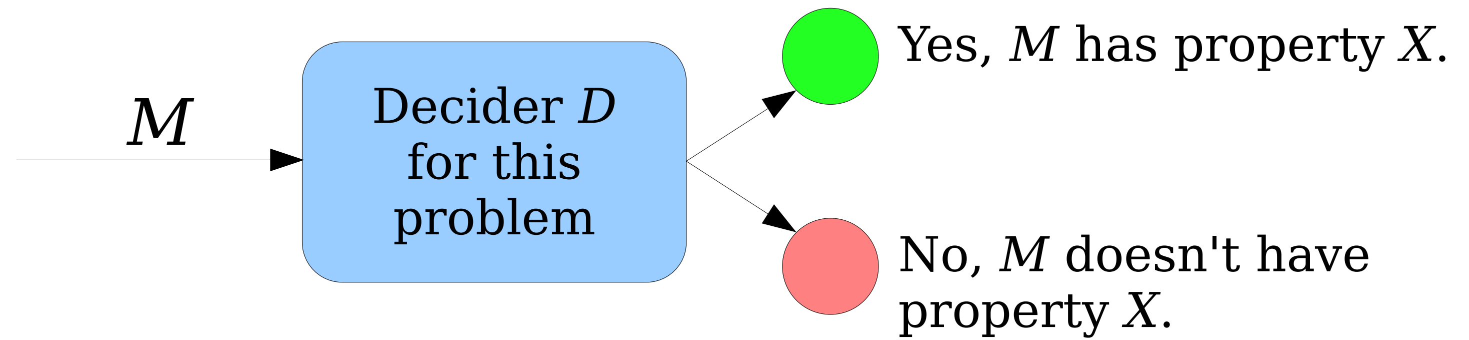
Contradiction!

The problem in question is decidable

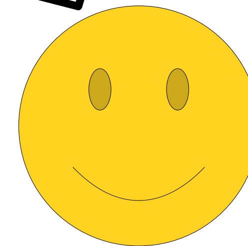
There is a decider $D$ for that problem.

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

Contradiction!




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

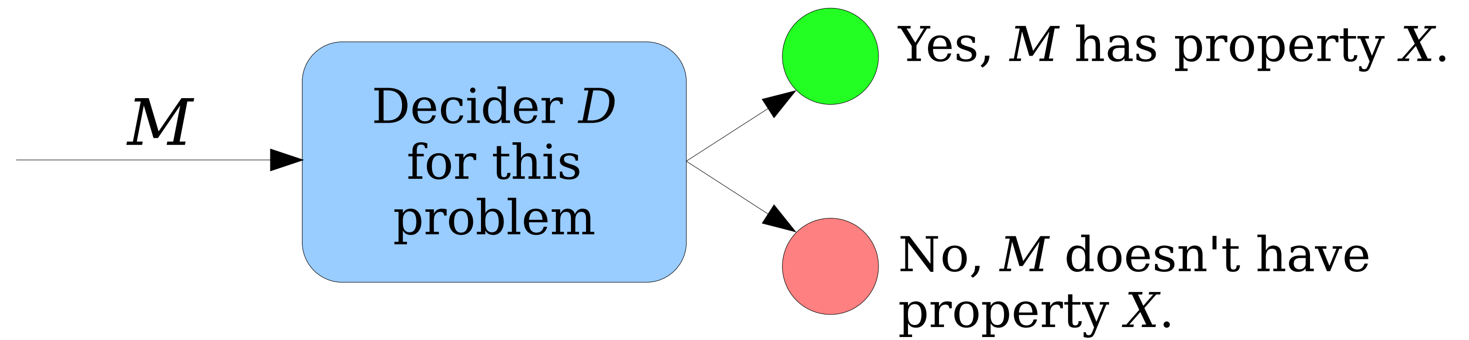


The problem in question is decidable

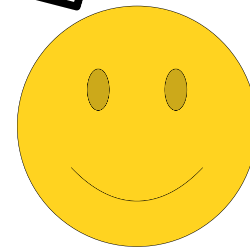
There is a decider $D$ for that problem.

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

Contradiction!




Think about what the decider would look like as a method in some high-level programming language.

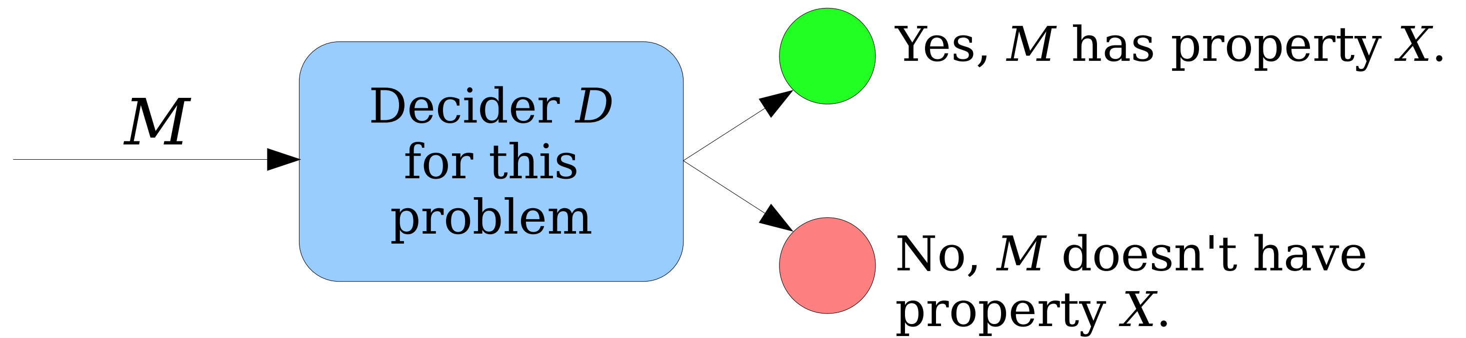


The problem in question is decidable

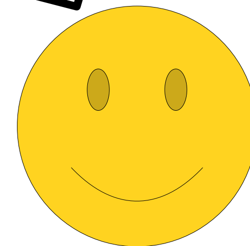
There is a decider $D$ for that problem.

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

Contradiction!




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.

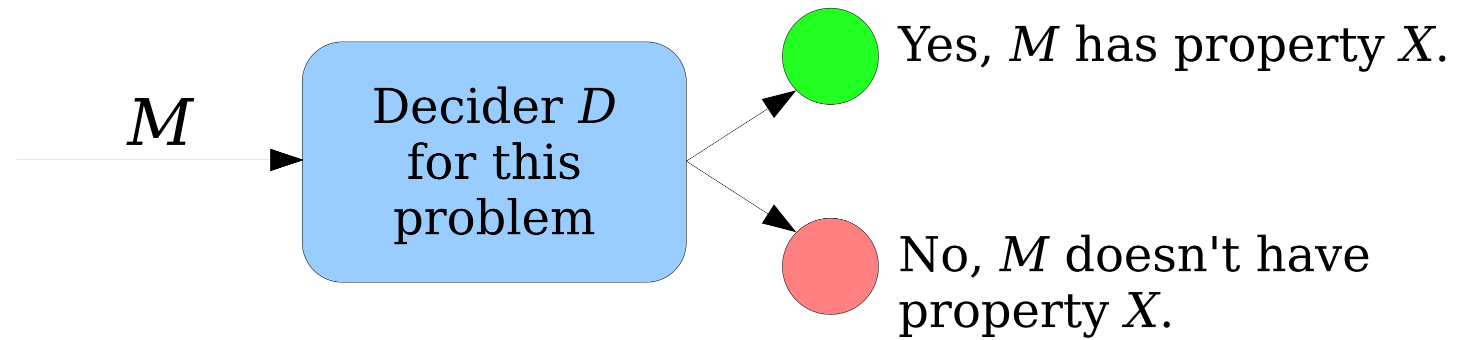


The problem in question is decidable

There is a decider $D$ for that problem.

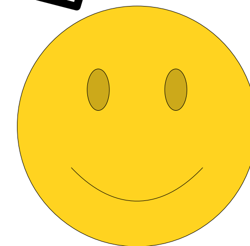
We can write programs that use $D$ as a helper function

Contradiction!




```
bool hasPropertyX(string function)
```

In this case, since our decider says whether the program has some property X, a good name would be something like hasPropertyX.

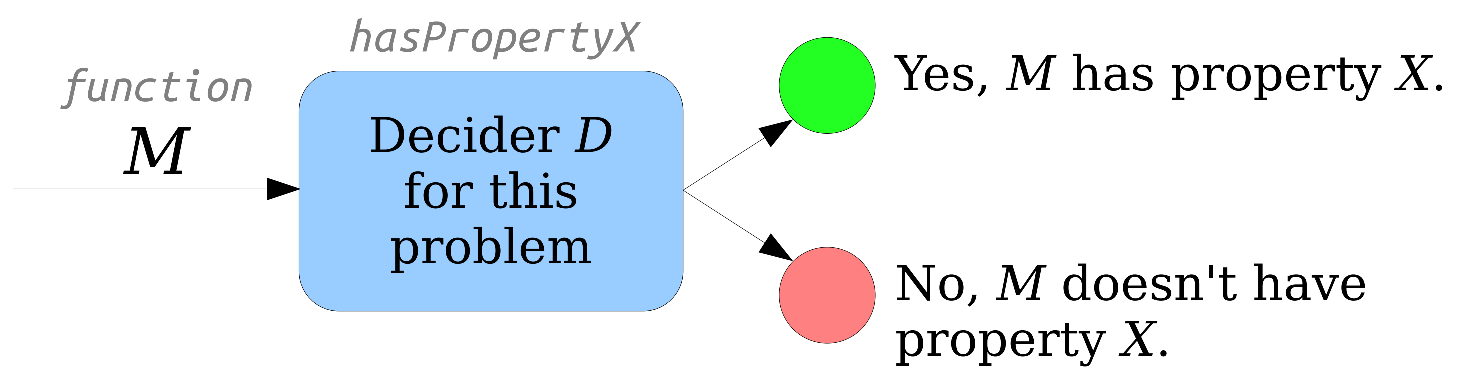


The problem in question is decidable

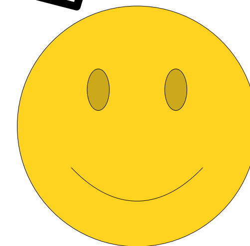
There is a decider $D$ for that problem.

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

Contradiction!




```
bool hasPropertyX(string function)
```

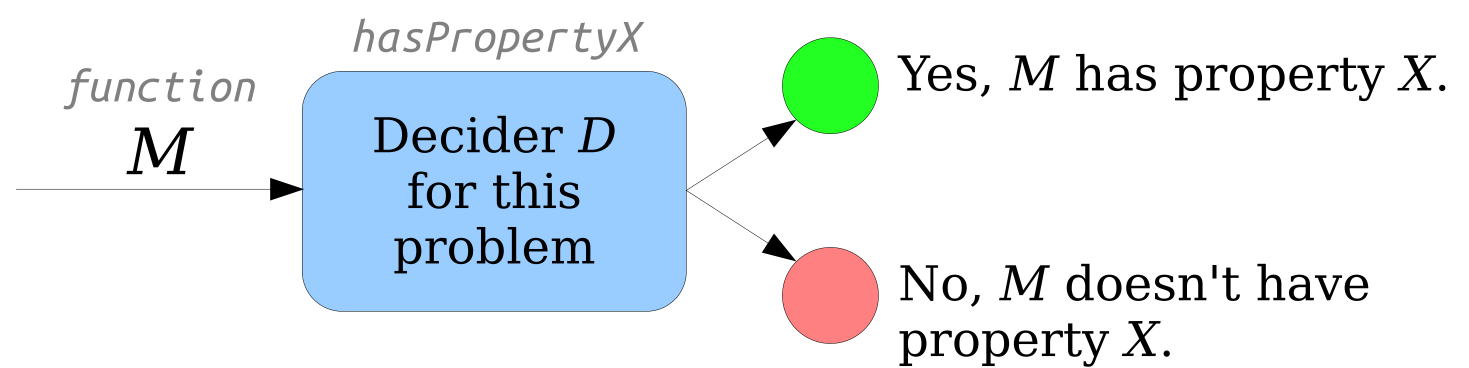


The problem in question is decidable

There is a decider $D$ for that problem.

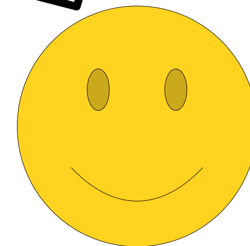
We can write programs that use $D$ as a helper function

Contradiction!




```
bool hasPropertyX(string function)
```

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



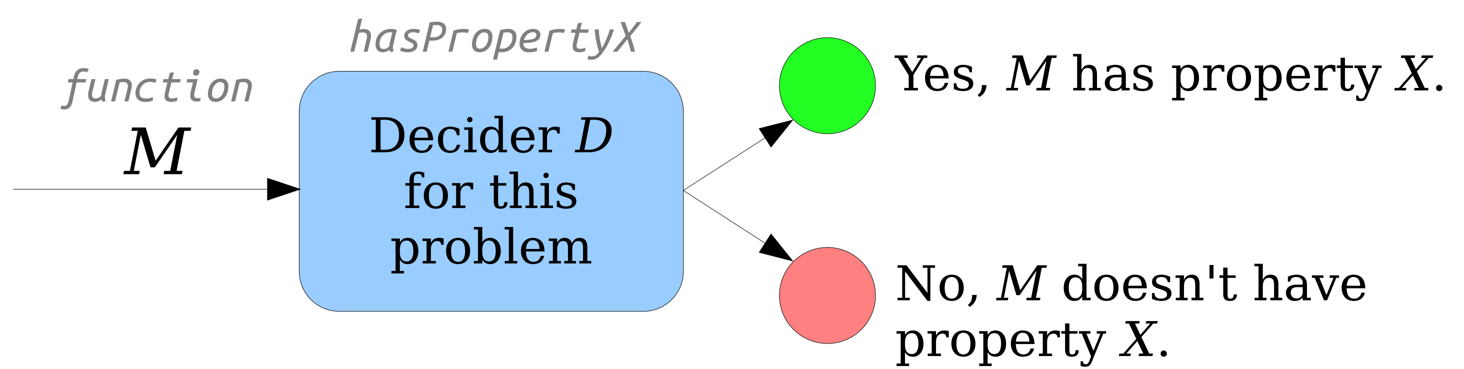
The problem in question is decidable

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!




```
bool hasPropertyX(string function)
```

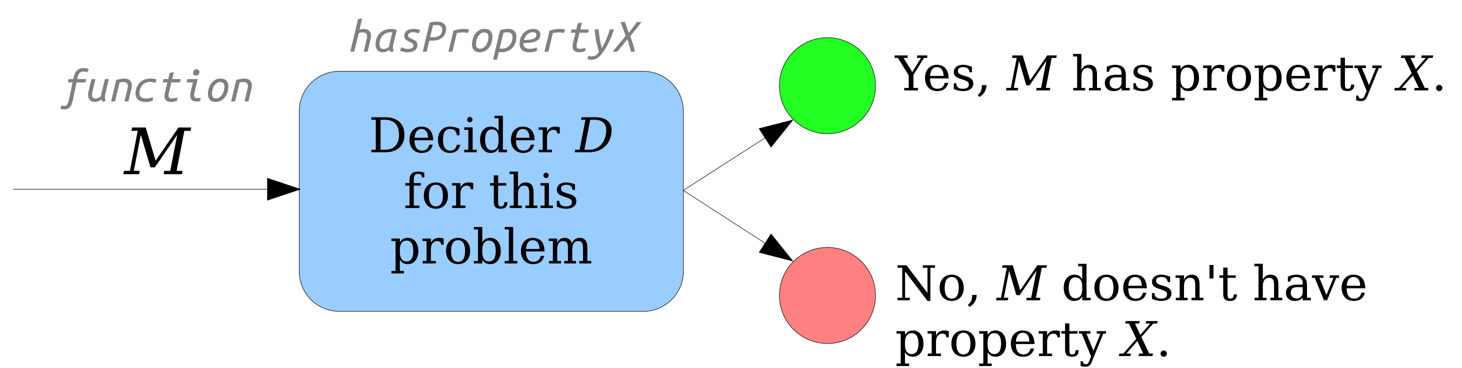
The problem in question is decidable

There is a decider $D$ for that problem.

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

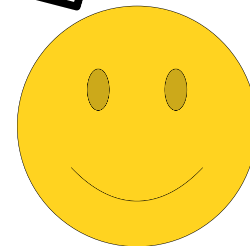
`trickster` has property $X$ if and only if it doesn't have property $X$

Contradiction!




```
bool hasPropertyX(string function)
```

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



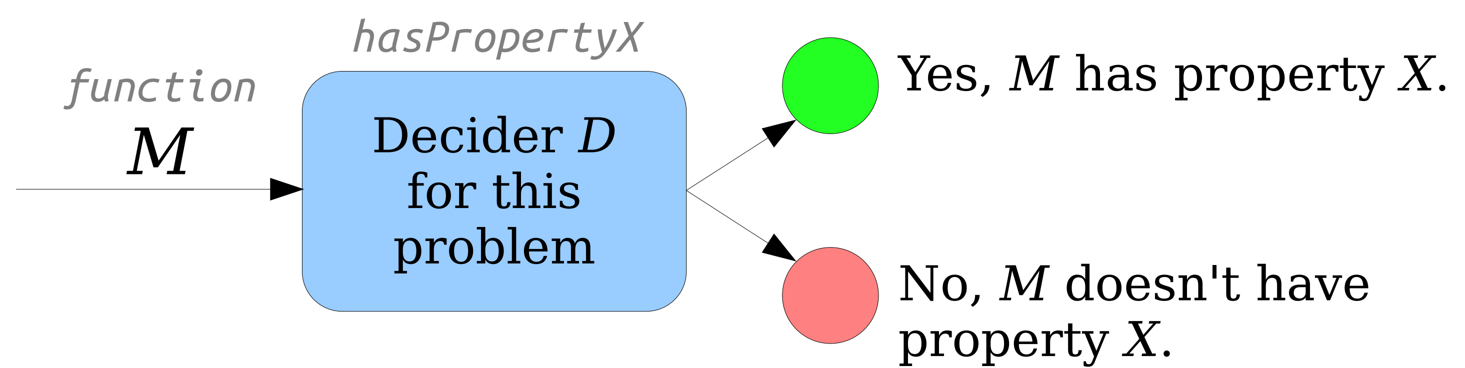
The problem in question is decidable

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!




```
bool hasPropertyX(string function)
```

`trickster` design specification:

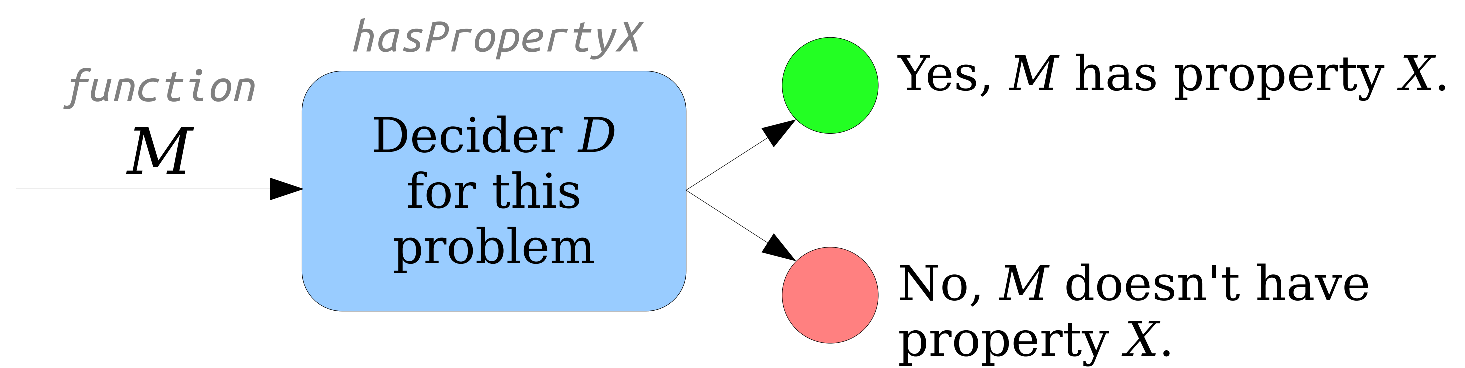
The problem in question is decidable

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!




```
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 $X$.

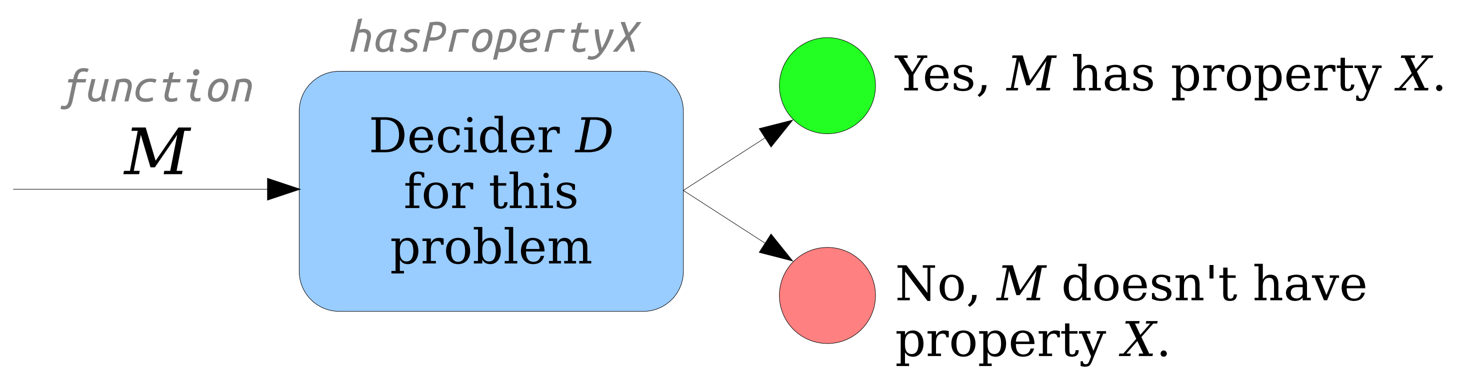
The problem in question is decidable

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!




```
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.

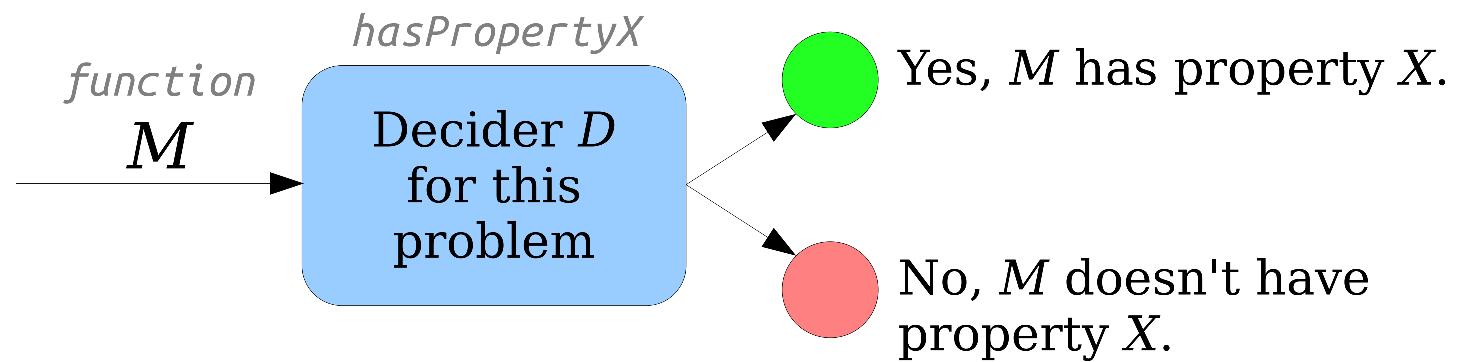
The problem in question is decidable

There is a decider *D* for that problem.

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

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

Contradiction!




```
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*.

```
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.

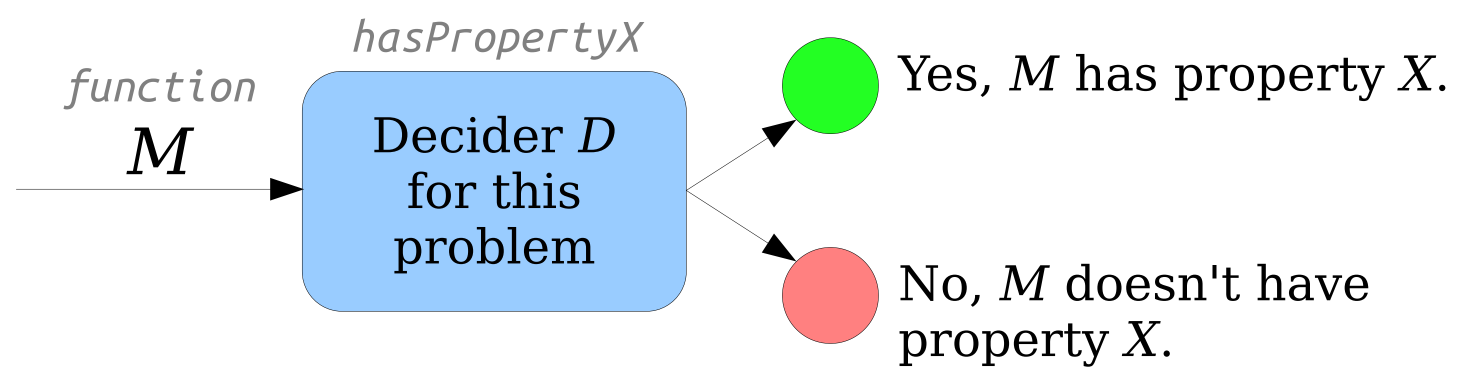
The problem in question is decidable

There is a decider *D* for that problem.

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

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

Contradiction!




```
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*.

```
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.
  }
}
```

And now you have a contradiction!

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.