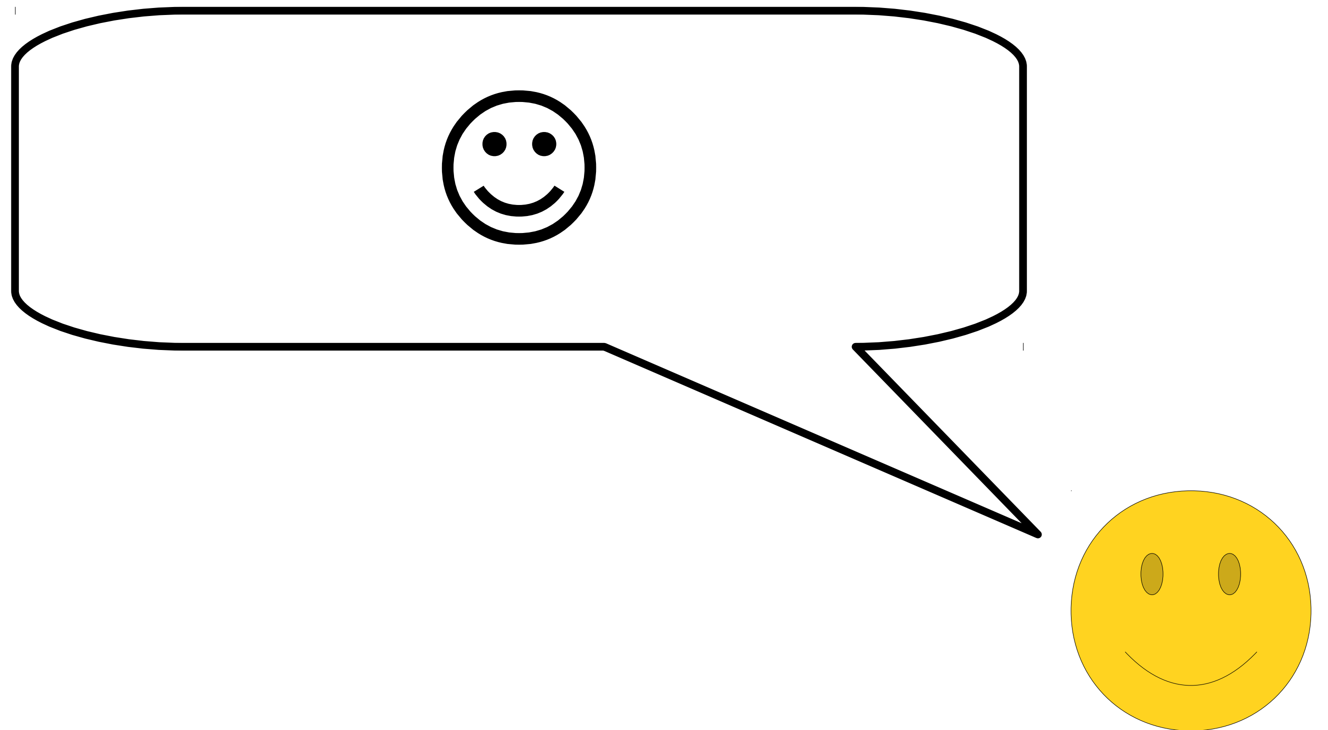
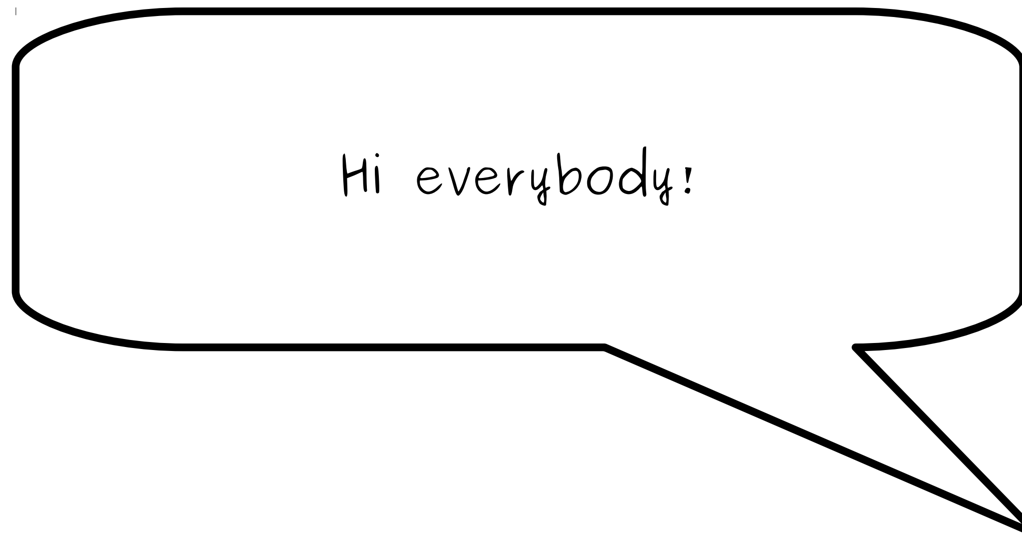
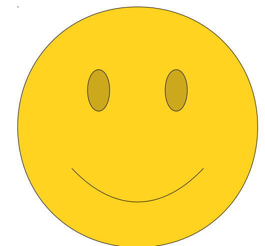
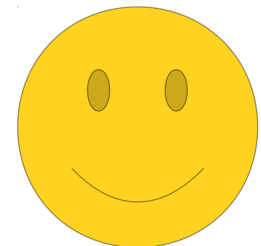
The Guide to Self-Reference



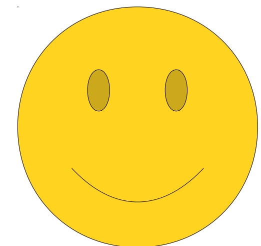

Hi everybody!


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.



What does this program do?

```
bool willAccept(st
    /* … some im
}

int main() {
    string me =
    string input

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

This lecture slide was the first time that we really saw self-reference, and a lot of you got pretty tripped up by what was going on.

Try running this program on any input.
What happens if

… this program accepts its input?
It rejects the input!

… this program doesn't accept its input?
It accepts the input!

What does this program do?

```
bool willAcc
    /* … some
}

int main() {
    string me
    string inpu

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

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

Try running this program on any input.
What happens if

… this program accepts its input?
It rejects the input!

… this program doesn't accept its input?
It accepts the input!

What does this program do?

```
bool willAccept(string program, string input) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (willAccept(me, input)) {
      reject();
   } else {
      accept();
   }
}
```

Let's see where it comes from!

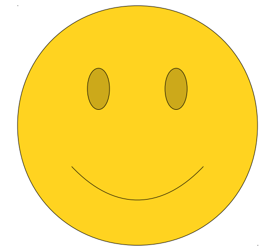
We'll take it from the top.

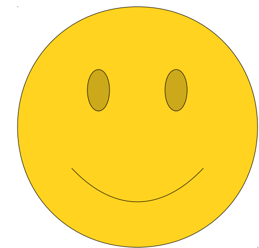
Try running this program on any input. What happens if

… this program accepts its input?
It rejects the input!

… this program doesn't accept its input?
It accepts the input!

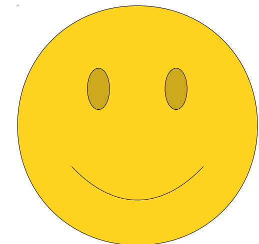
Let's try to use self-reference to prove that A_{TM} is undecidable.




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

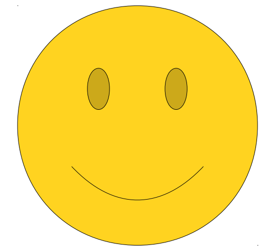
$A_{TM} \in \mathbf{R}$

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



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

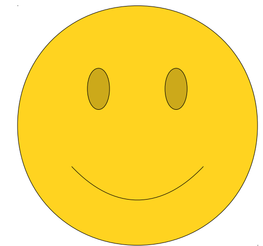
Contradiction!



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

Contradiction!

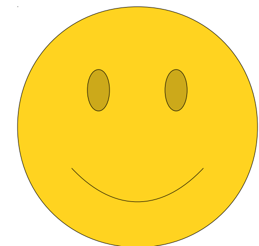
If we can get a contradiction – any contradiction – we'll see that our assumption was wrong.



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

Contradiction!

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



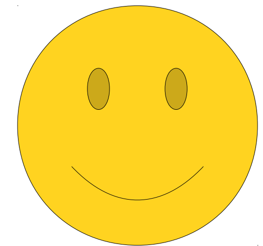
$A_{TM} \in \mathbf{R}$

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?

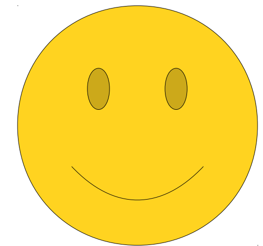


Contradiction!

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

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*.

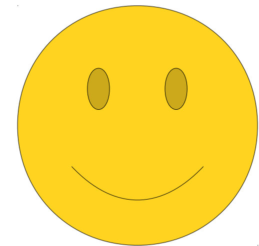


Contradiction!

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

There is a decider D for A_{TM}

What might this decider look like?



Contradiction!

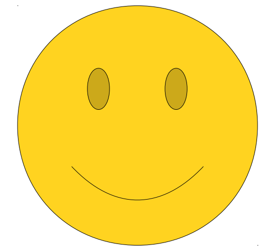
$A_{TM} \in \mathbf{R}$

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.



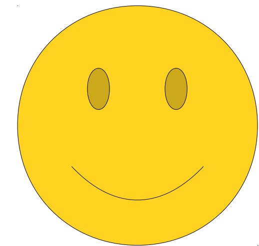
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

Contradiction!

Decider D for A_{TM}

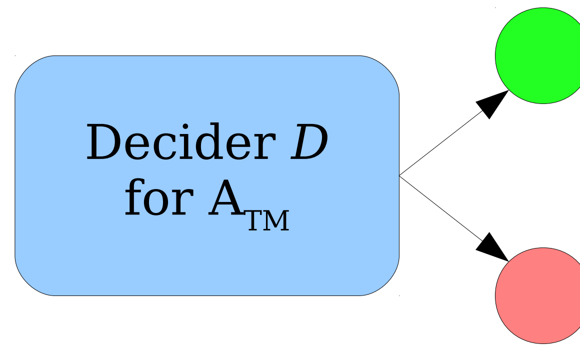
First, it has to always halt.



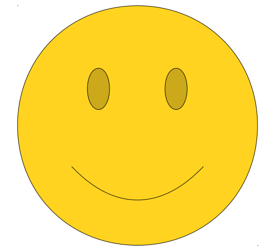
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

Contradiction!



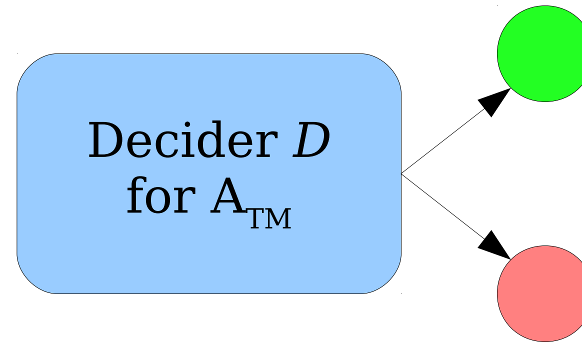
That means that if you give it any input, it has to either accept or reject it. We'll visualize this with these two possible outputs.



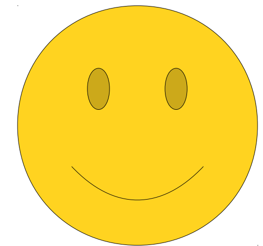
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

Contradiction!

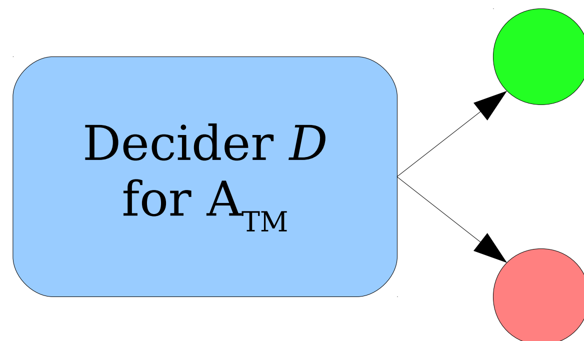


Next, the decider has to tell us something about A_{TM}.

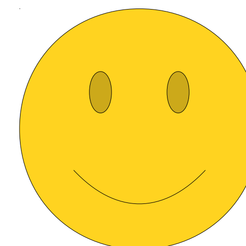


$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

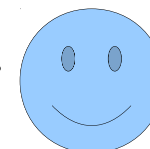


Next, the decider has to tell us something about A_{TM}.



As a reminder, A_{TM} is the language

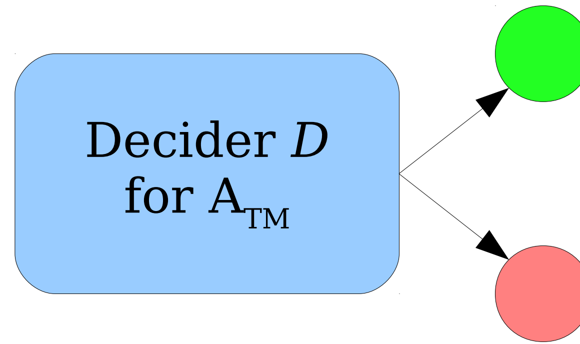
$\{ \langle M, w \rangle \mid M$ is a TM and M accepts $w \}$



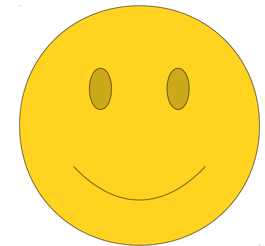
Contradiction!

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

There is a decider D for A_{TM}

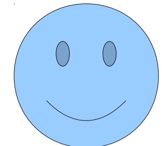


Specifically, the decider D needs to take in an input and tell us whether that input is in A_{TM}



As a reminder, A_{TM} is the language

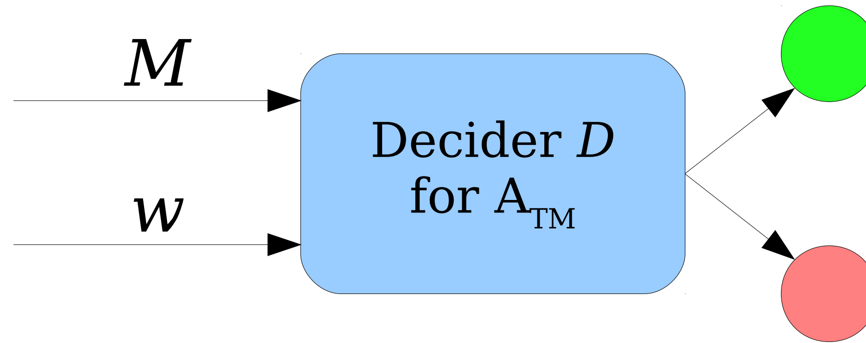
$\{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$



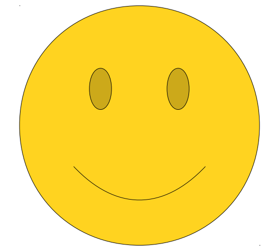
Contradiction!

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

There is a decider D for A_{TM}

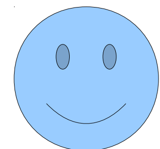


A_{TM} is a language of pairs of TMs and strings, so D will take in two inputs, a machine M and a string w.



As a reminder, A_{TM} is the language

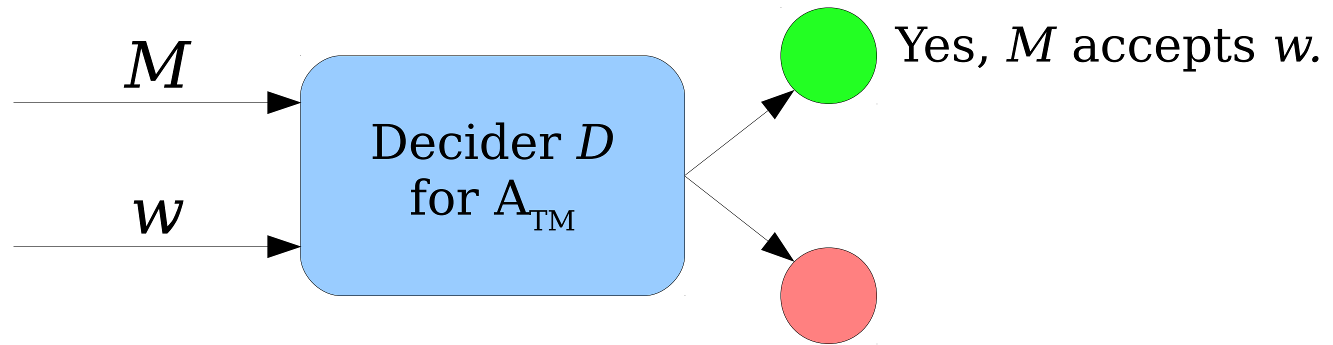
$\{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$



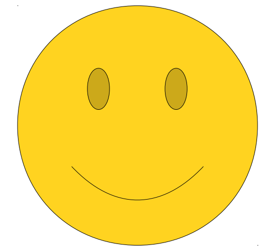
Contradiction!

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

There is a decider D for A_{TM}

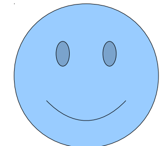


If D accepts its input, it means that $\langle M, w \rangle$ is in A_{TM}.



As a reminder, A_{TM} is the language

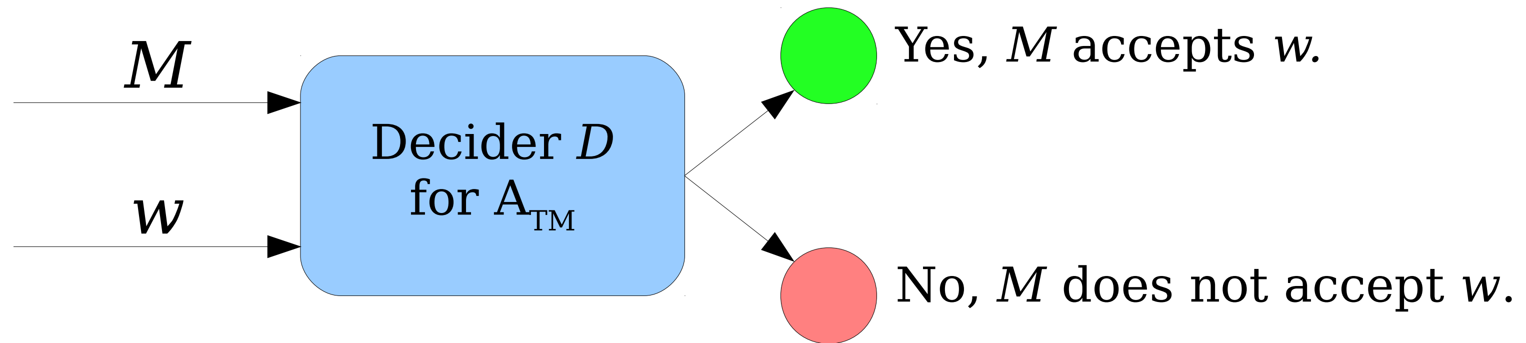
$\{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$



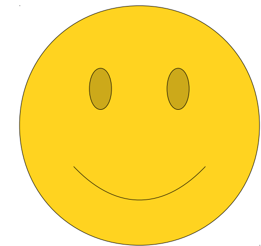
Contradiction!

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

There is a decider D for A_{TM}

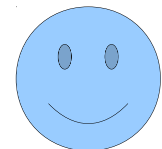


Otherwise, if D rejects its input, it means that M doesn't accept w.



As a reminder, A_{TM} is the language

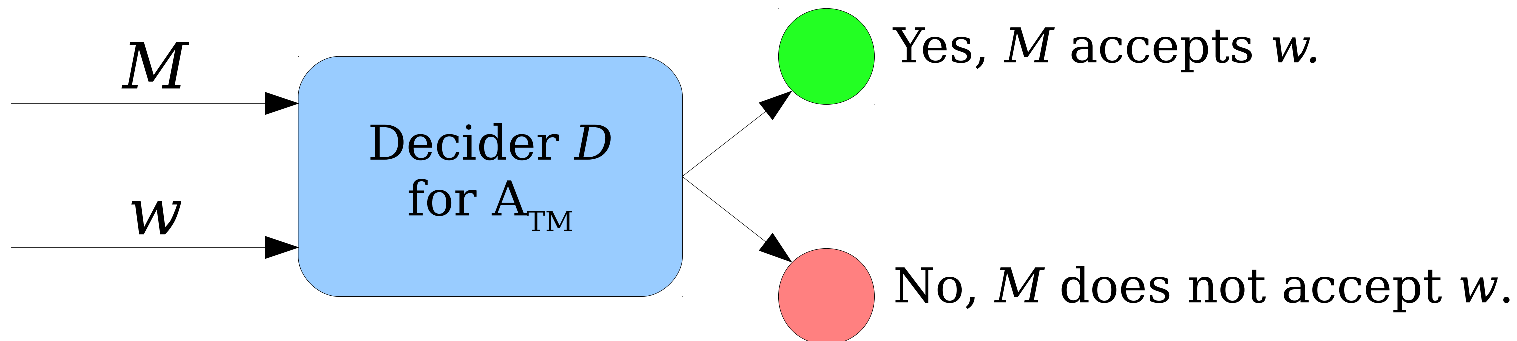
{ ⟨M, w⟩ | M is a TM and M accepts w }



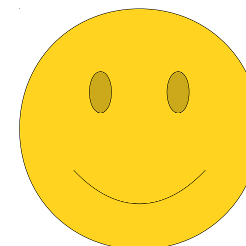
Contradiction!

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

There is a decider D for A_{TM}



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



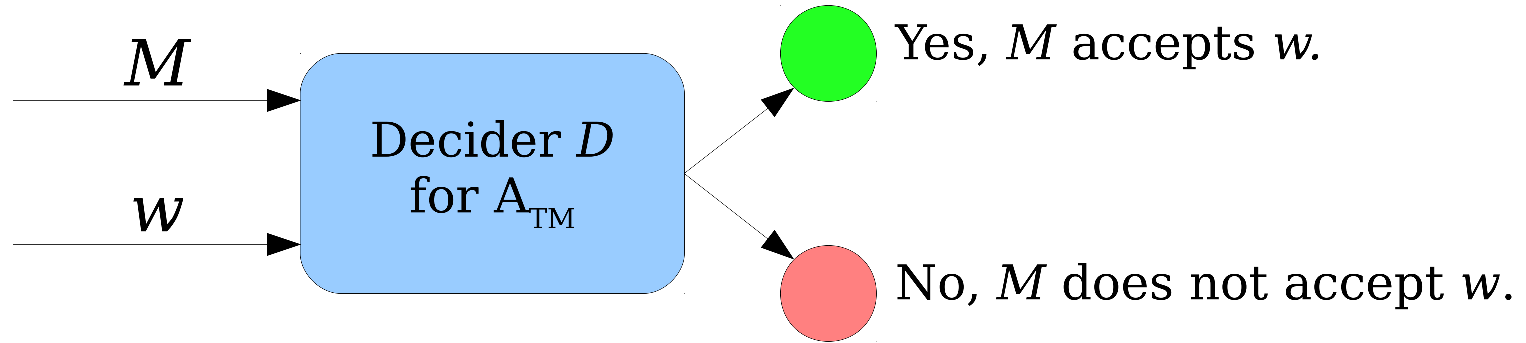
Contradiction!

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

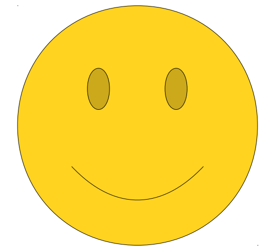
There is a decider D for A_{TM}

We can write programs that use D as a helper method

Contradiction!



We've seen the idea that TMs can run other TMs as subroutines. This means we can write programs that use D as a subroutine.

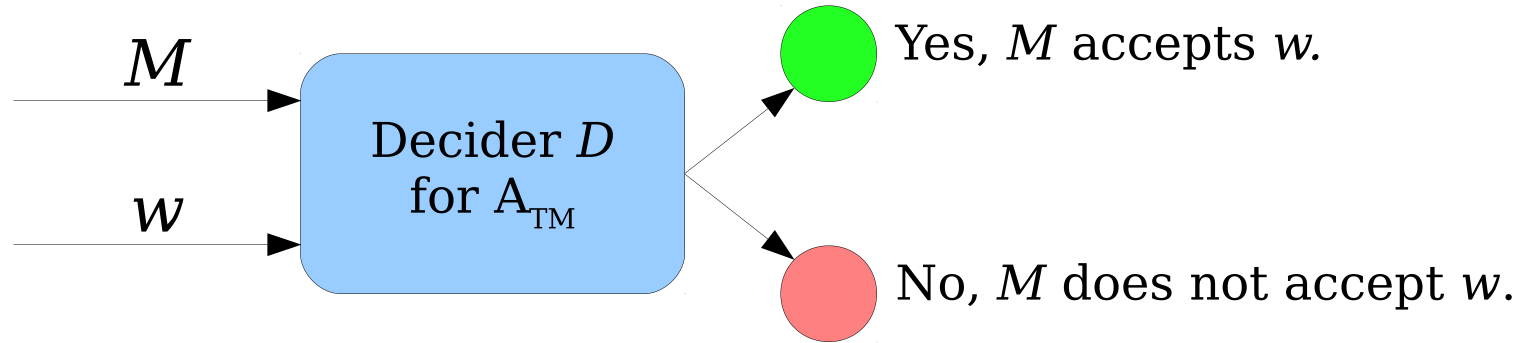


$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

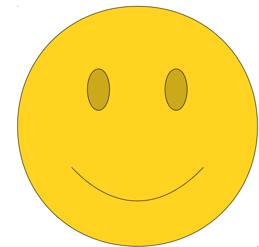
We can write programs that use D as a helper method

Contradiction!




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

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

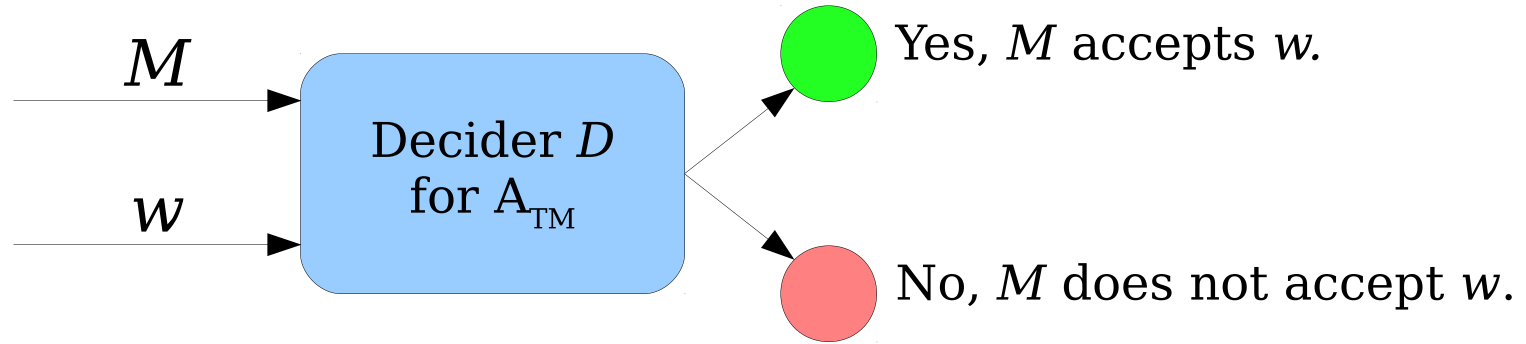


$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

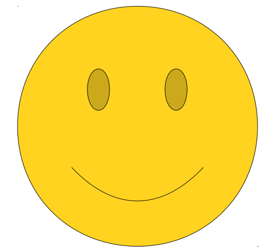
We can write programs that use D as a helper method

Contradiction!




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

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

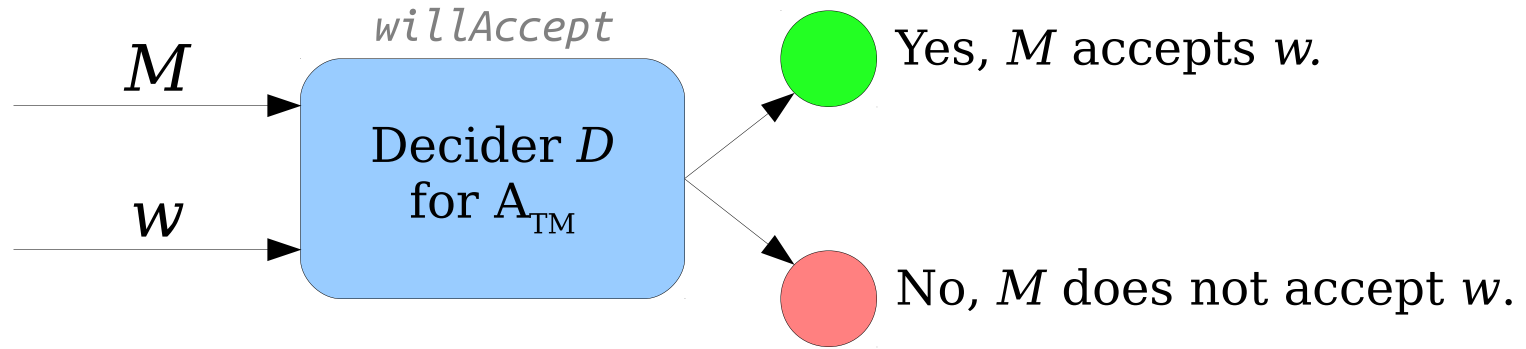


$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

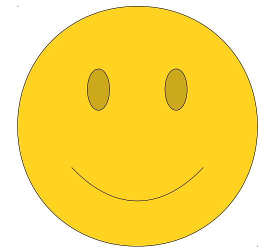
We can write programs that use D as a helper method

Contradiction!




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

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

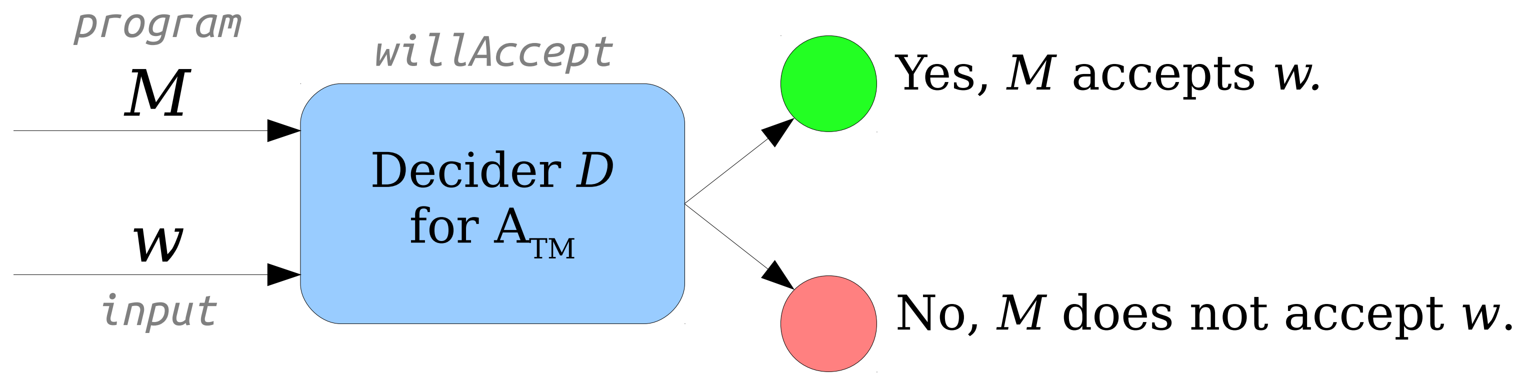


$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

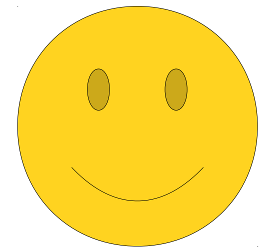
We can write programs that use D as a helper method

Contradiction!




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

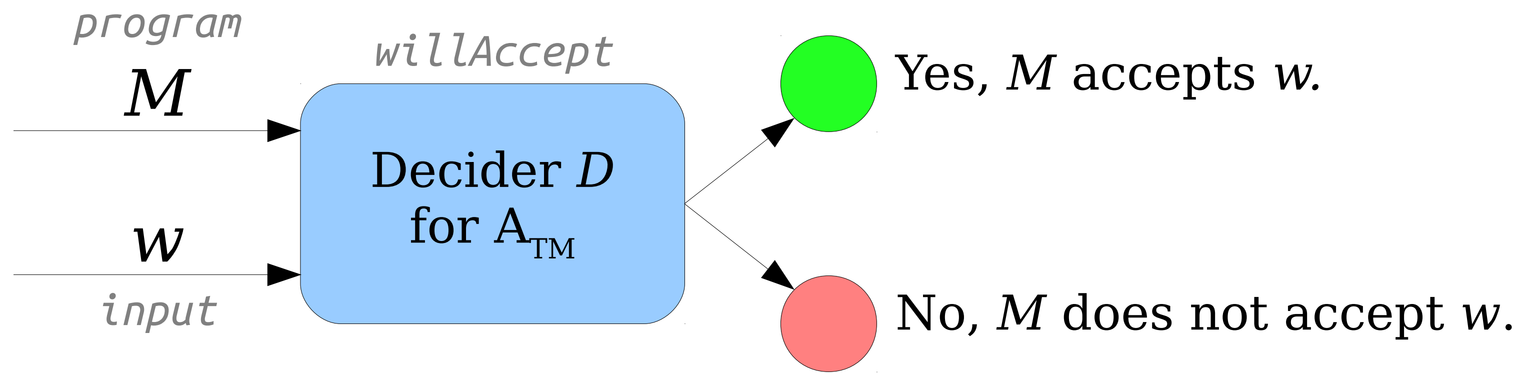
The two arguments to willAccept then correspond to the inputs to the decider D.



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

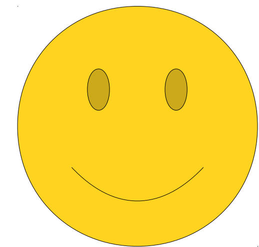
There is a decider D for A_{TM}

We can write programs that use D as a helper method




```
bool willAccept(string program, 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.



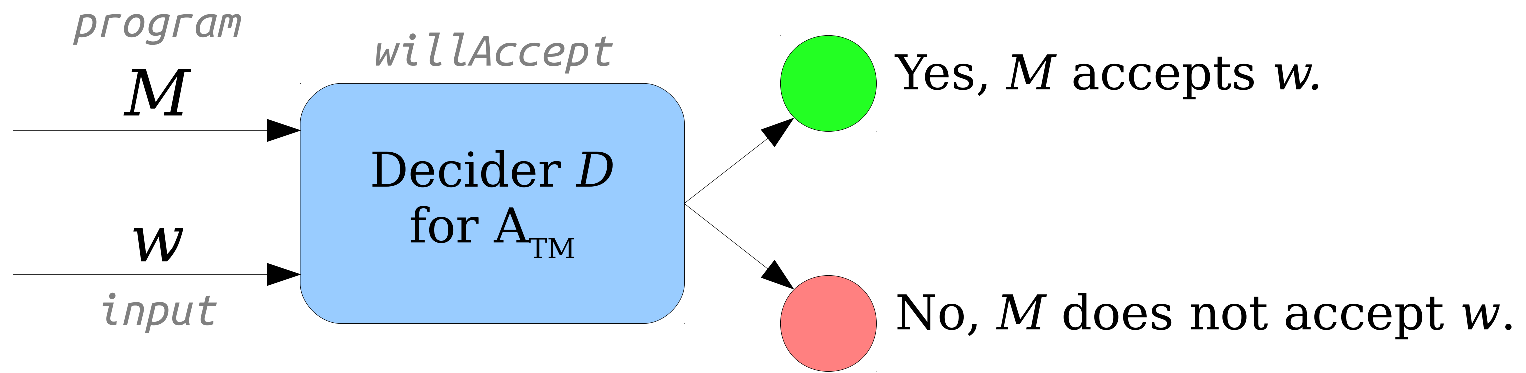
Contradiction!

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

There is a decider D for A_{TM}

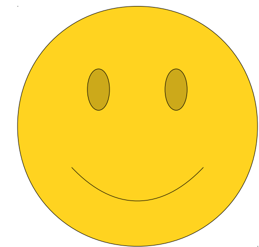
We can write programs that use D as a helper method

Contradiction!




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

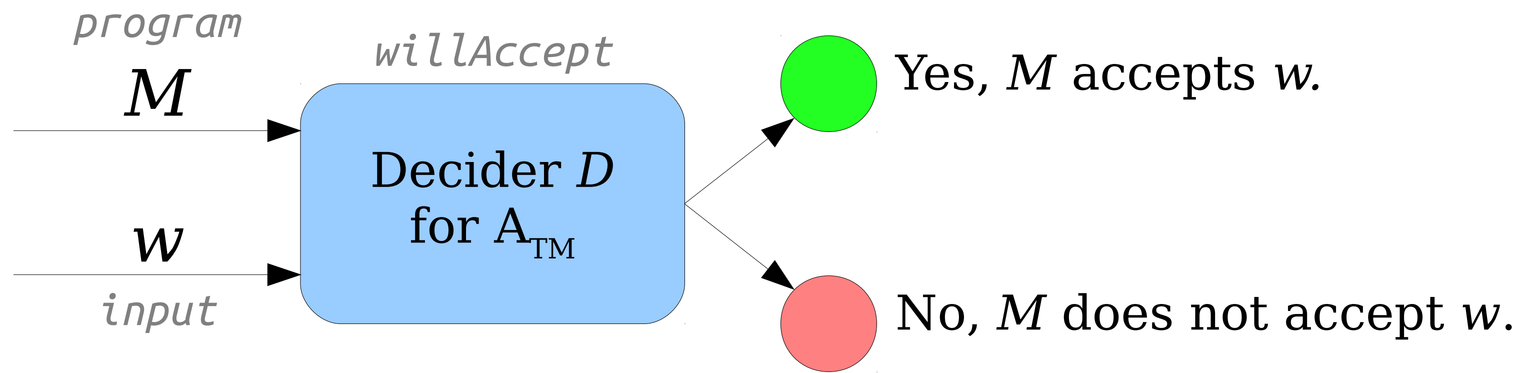
So at this point we've just set up the fact that this subroutine exists. What exactly are we going to do with it?



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

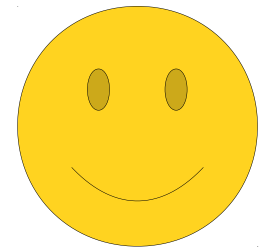
There is a decider *D* for A_{TM}

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




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

Ultimately, we're trying to get a contradiction.



Contradiction!

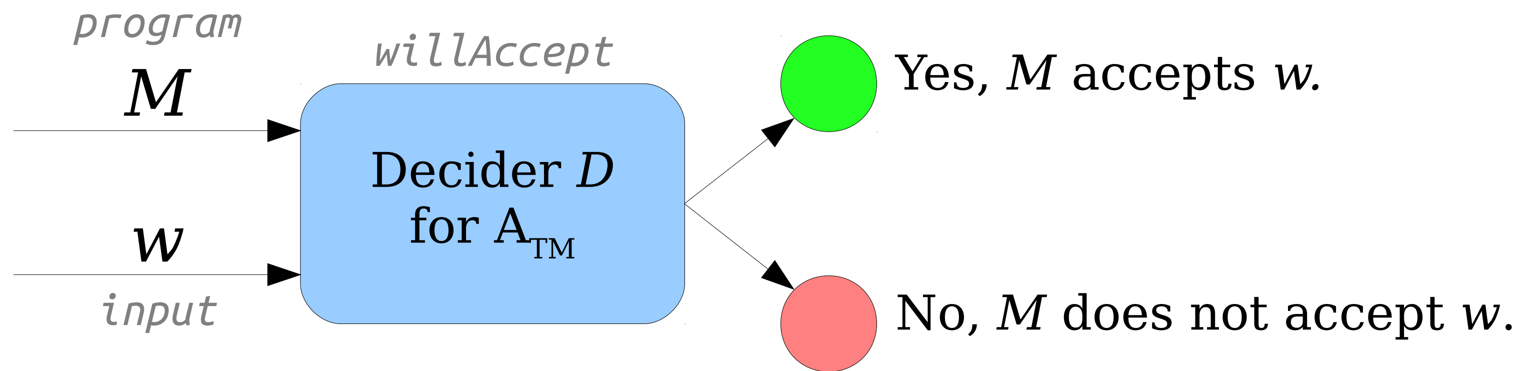
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

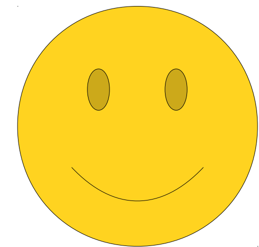
Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

Specifically, we're going to build a program – which we'll call P – that has some really broken behavior… it will accept its input if and only if it doesn't accept its input!



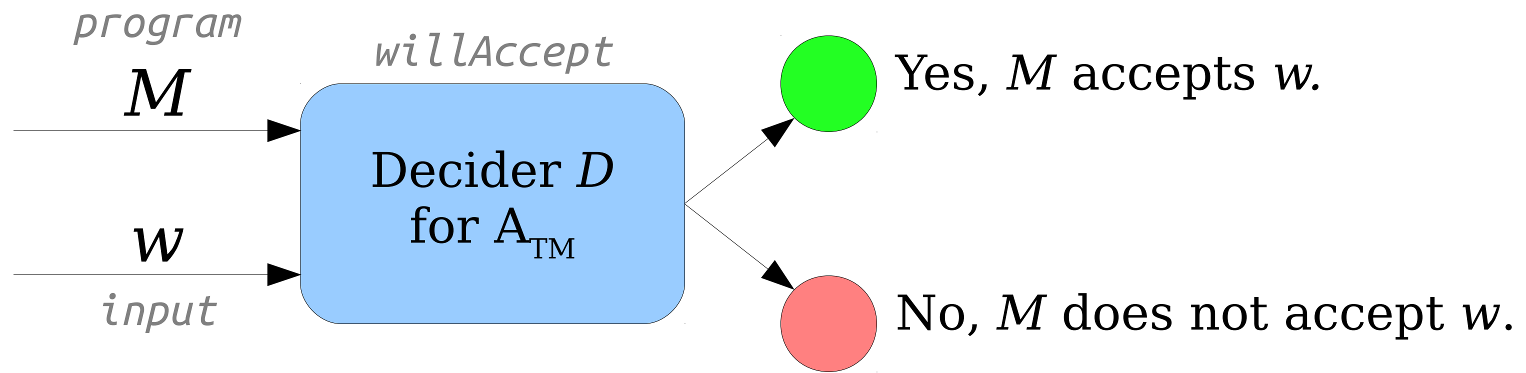
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

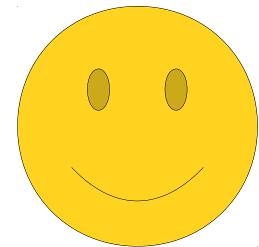
Program P accepts its input if and only if program P does not accept its input

Contradiction!




```
bool willAccept(string program, 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.



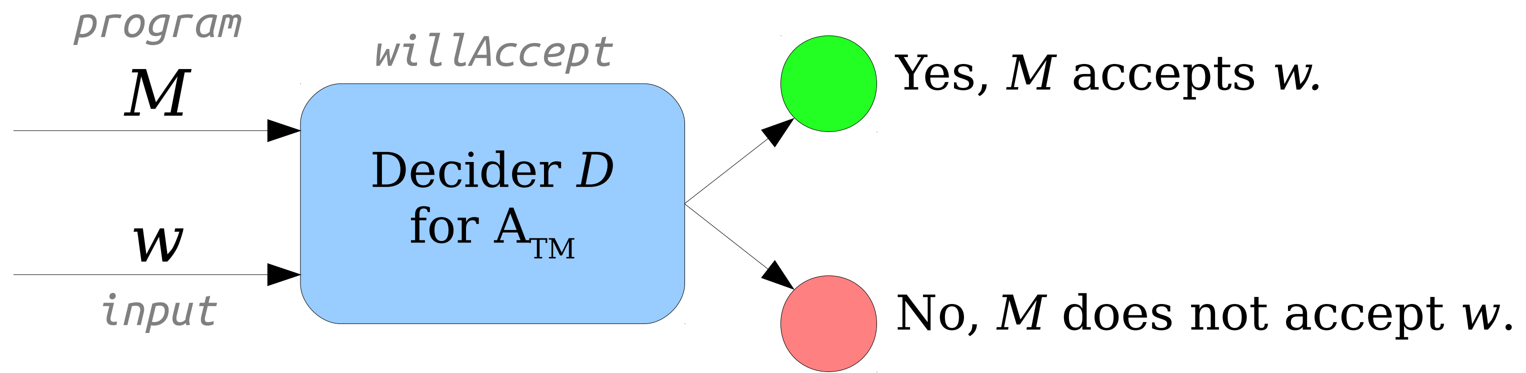
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

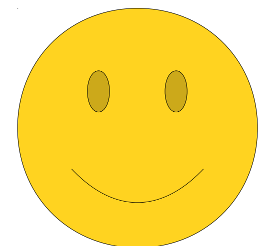
Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

Now, we haven't actually written this program P yet. That's the next step.



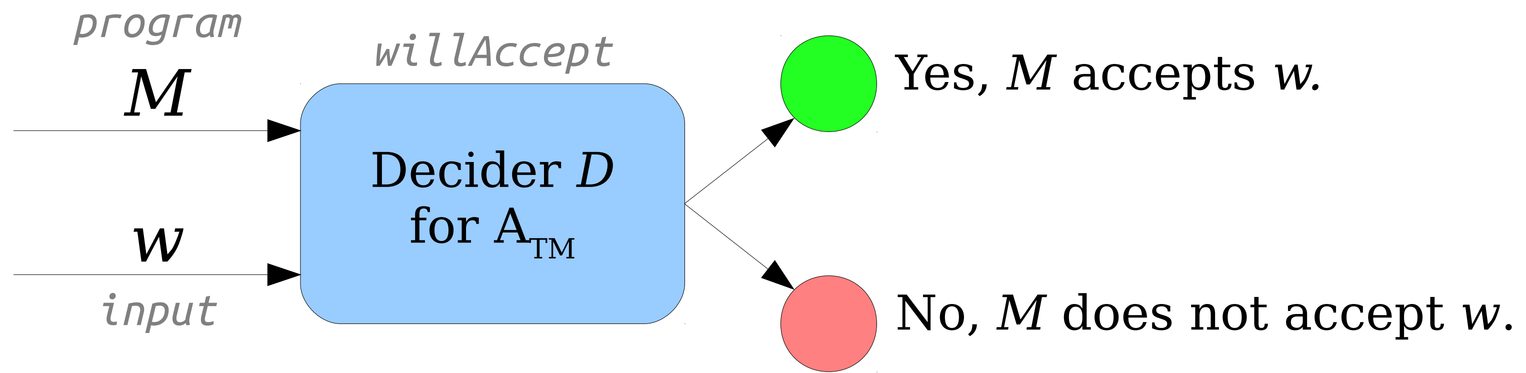
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

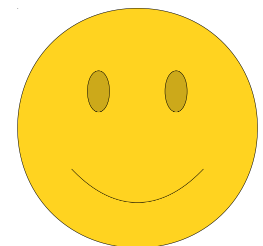
We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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



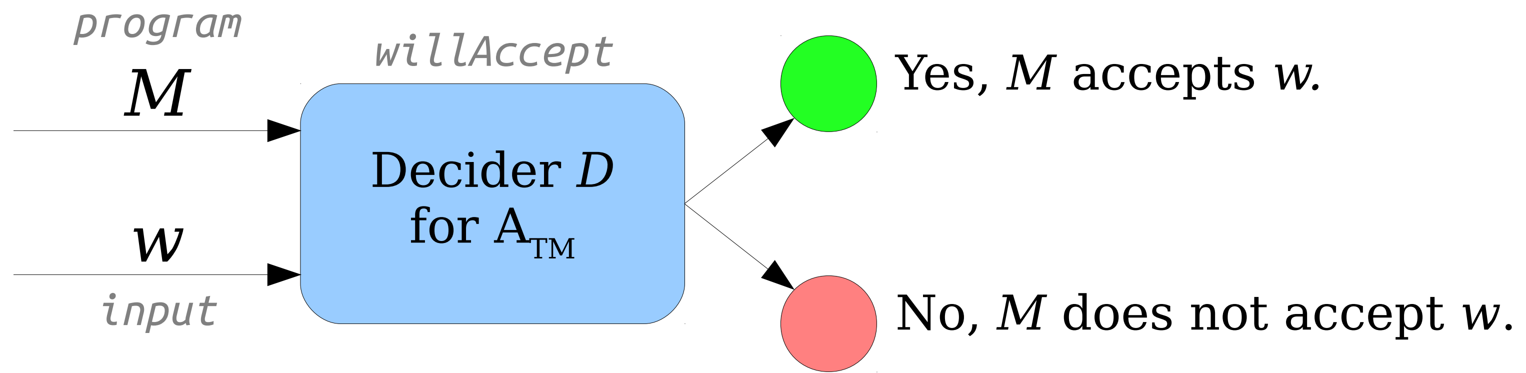
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

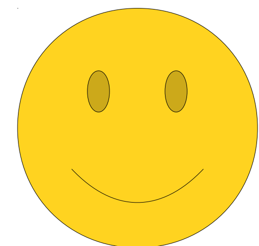
Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

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



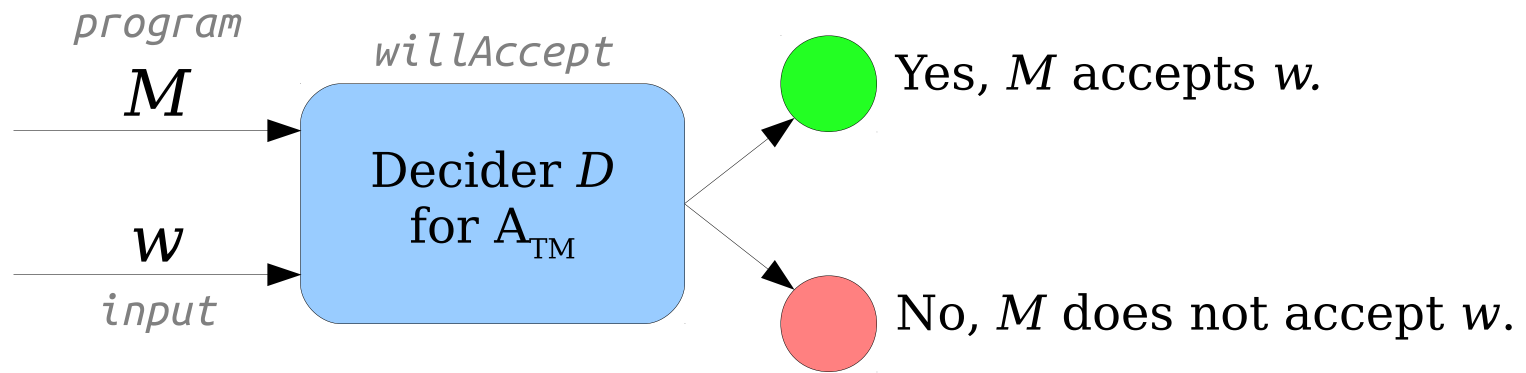
$A_{TM} \in \mathbf{R}$

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

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

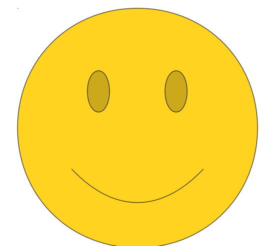
Program *P* accepts its input if and only if program *P* does not accept its input

Contradiction!




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

Program *P* design specification:



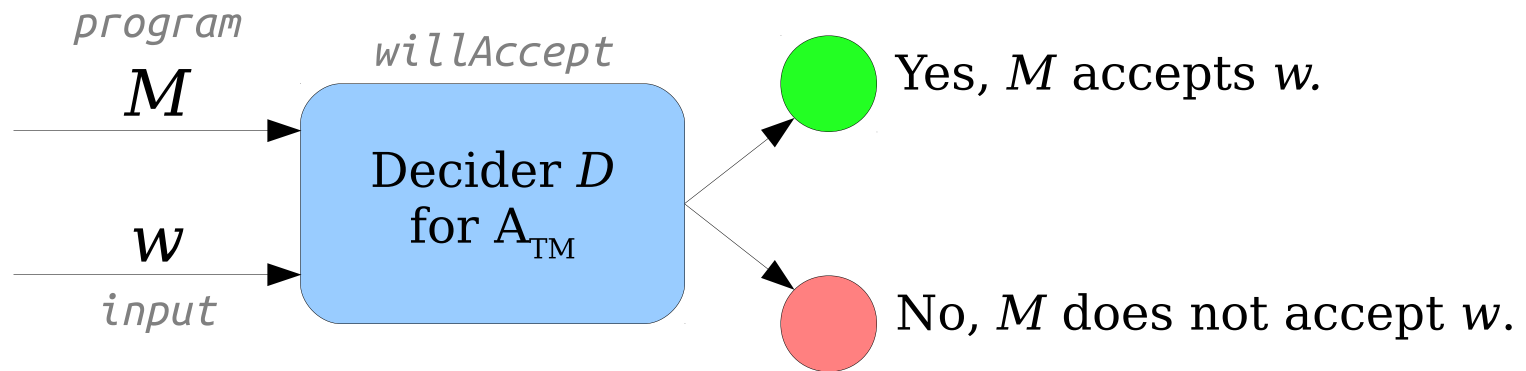
$A_{TM} \in \mathbf{R}$

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

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

Program *P* accepts its input if and only if program *P* does not accept its input

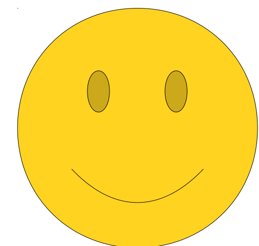
Contradiction!




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

Program *P* design specification:

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



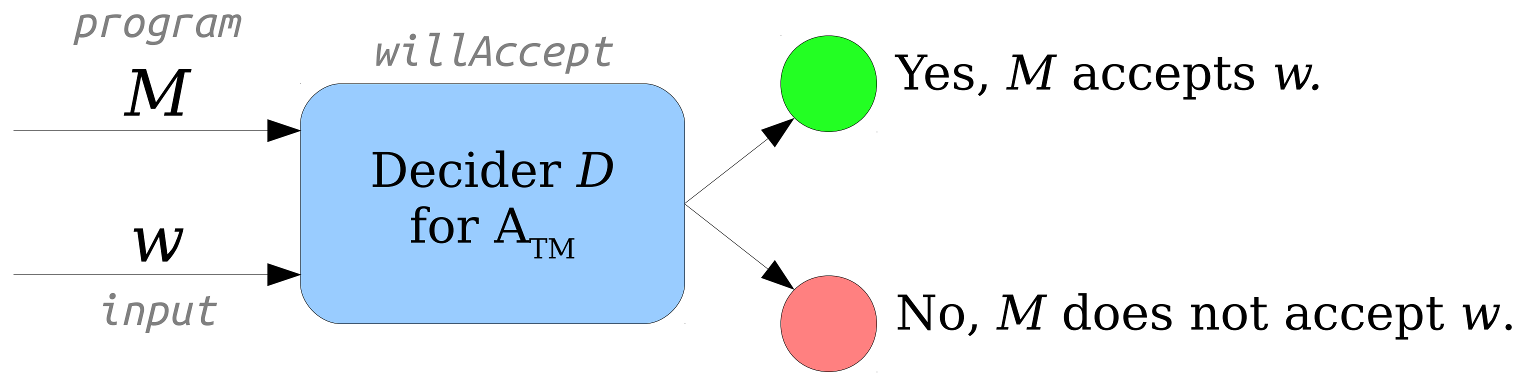
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!



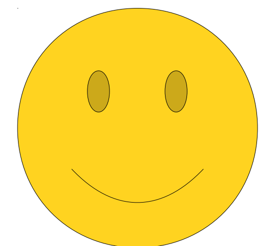

```
bool willAccept(string program, string input)
```

Program P design specification:

If P accepts its input, then

P does not accept its input.

First, if this program P is supposed to accept its input, then it needs to not accept its input.



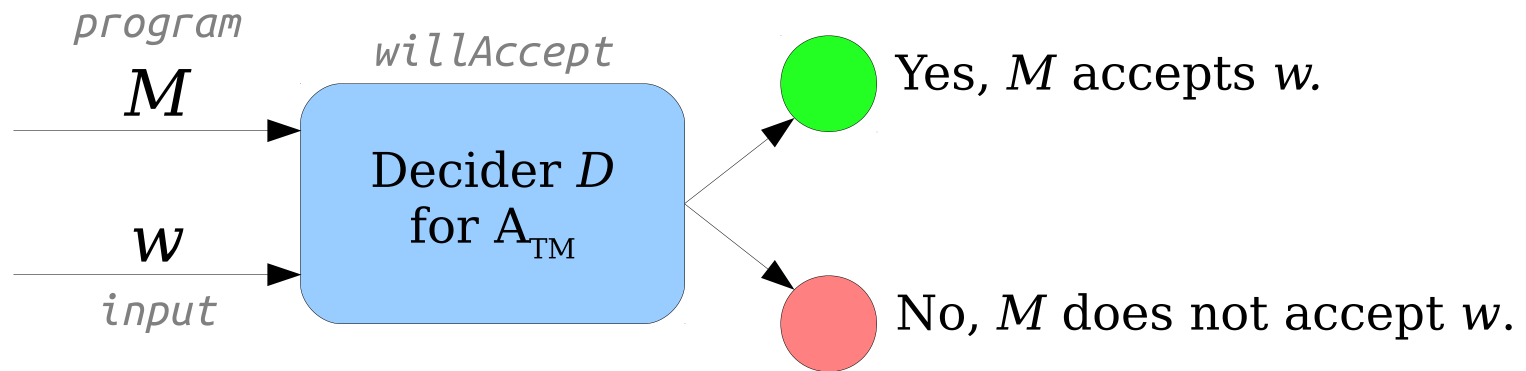
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!

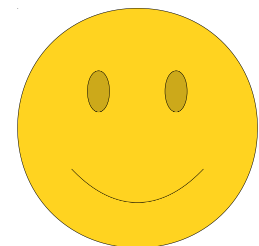



```
bool willAccept(string program, string input)
```

Program P design specification:

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

Next, if this program P is supposed to not accept its input, then it needs to accept its input.



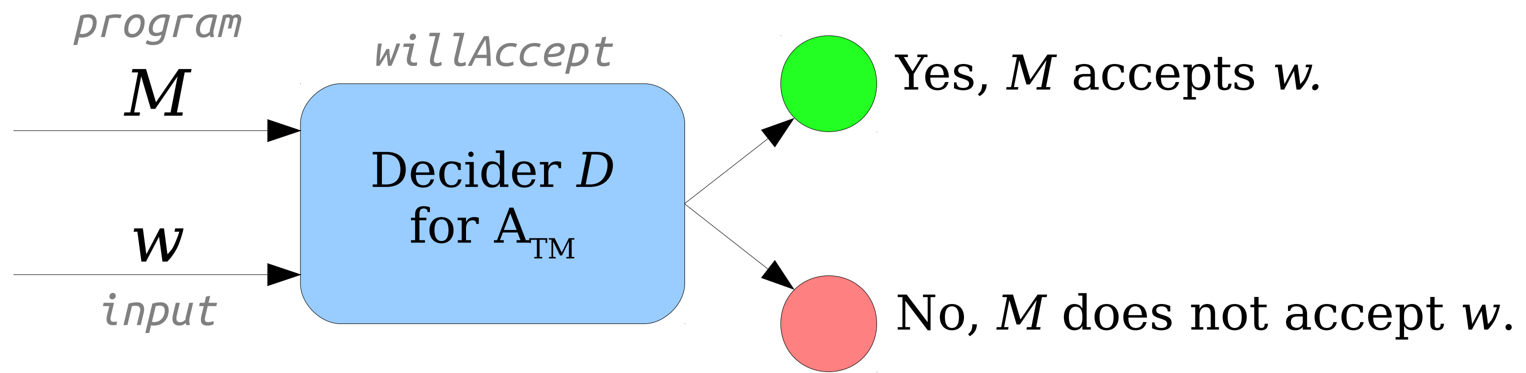
$A_{TM} \in \mathbf{R}$

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

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

Program *P* accepts its input if and only if program *P* does not accept its input

Contradiction!

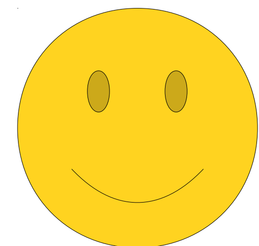



```
bool willAccept(string program, string input)
```

Program *P* design specification:

If *P* accepts its input, then
P does not accept its input.
If *P* does not accept its input, then
P accepts its input.

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



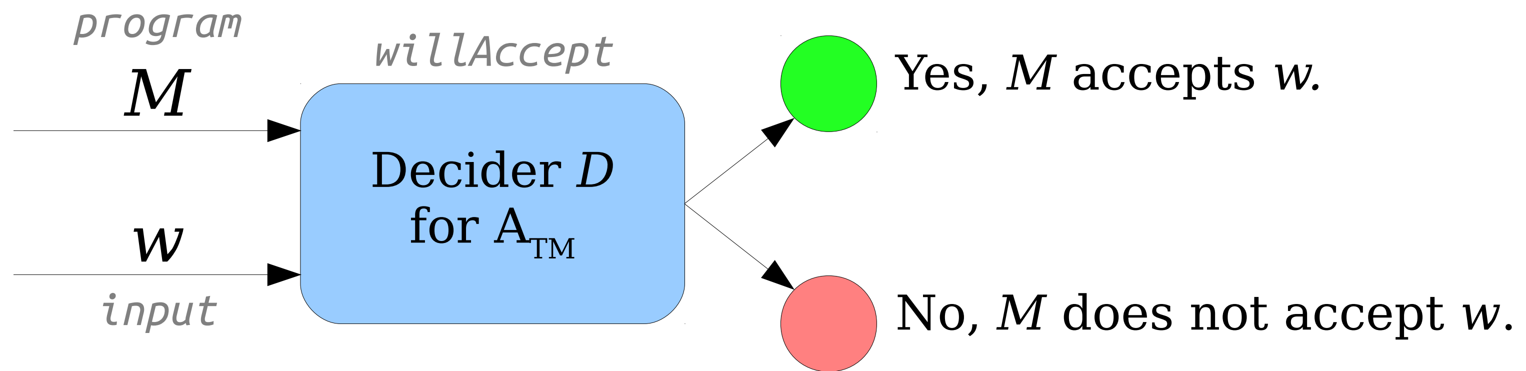
$A_{TM} \in \mathbf{R}$

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

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

Program *P* accepts its input if and only if program *P* does not accept its input

Contradiction!



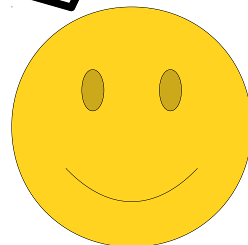

```
bool willAccept(string program, string input)
```

Program *P* design specification:

If *P* accepts its input, then
P does not accept its input.
If *P* does not accept its input, then
P accepts its input.

```
// Program P
```

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



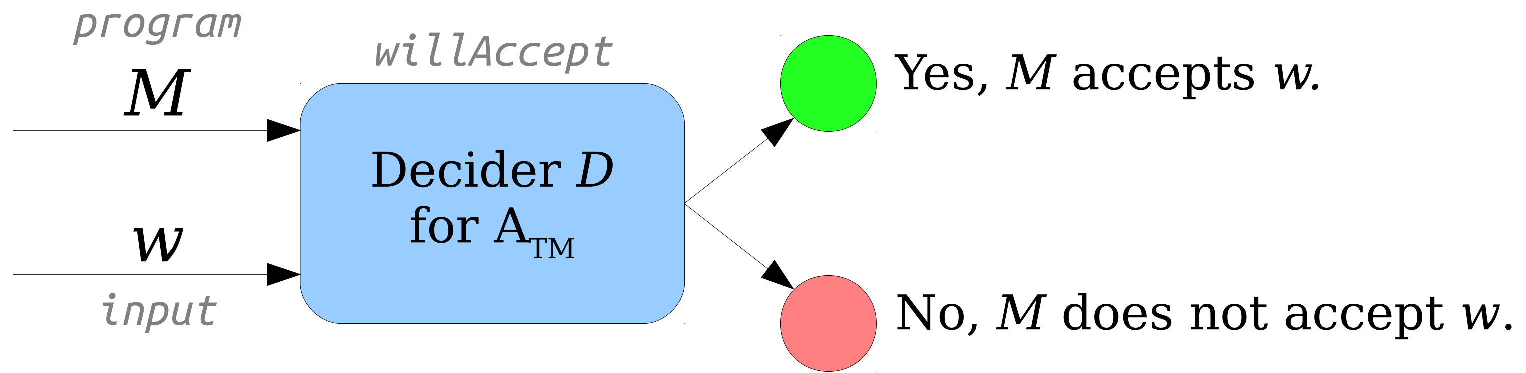
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!



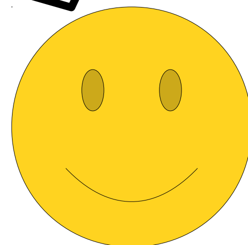

```
bool willAccept(string program, string input)
```

Program P design specification:

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

```
// Program P

int main() {



}
```

Like most programs, our program begins execution in main().



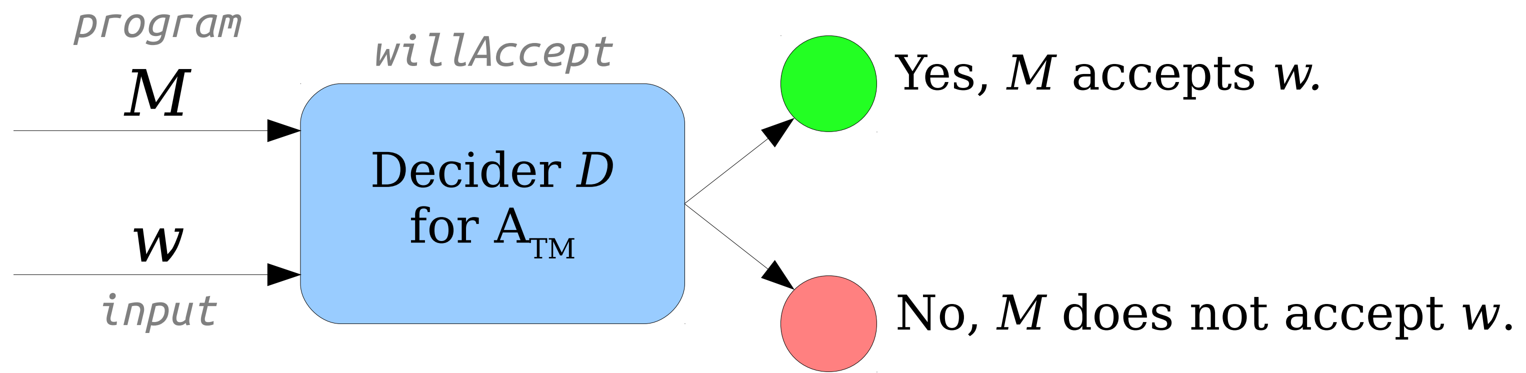
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!

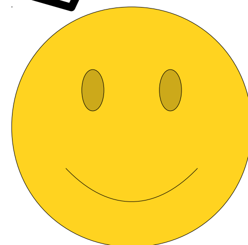



```
bool willAccept(string program, string input)
```

Program P design specification:

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

```
// Program P

int main() {
  string input = getInput();


}
```



Our program needs to get some input, so let's do that here.

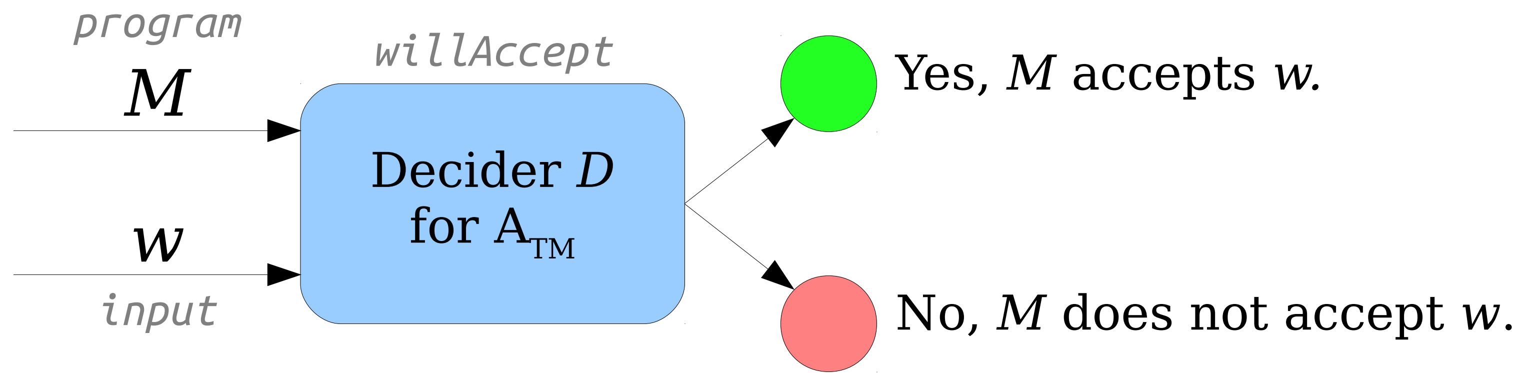
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!



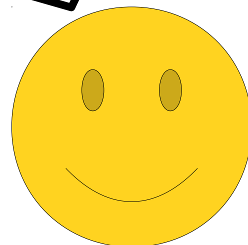

```
bool willAccept(string program, string input)
```

Program P design specification:

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

```
// Program P

int main() {
  string input = getInput();


}
```

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



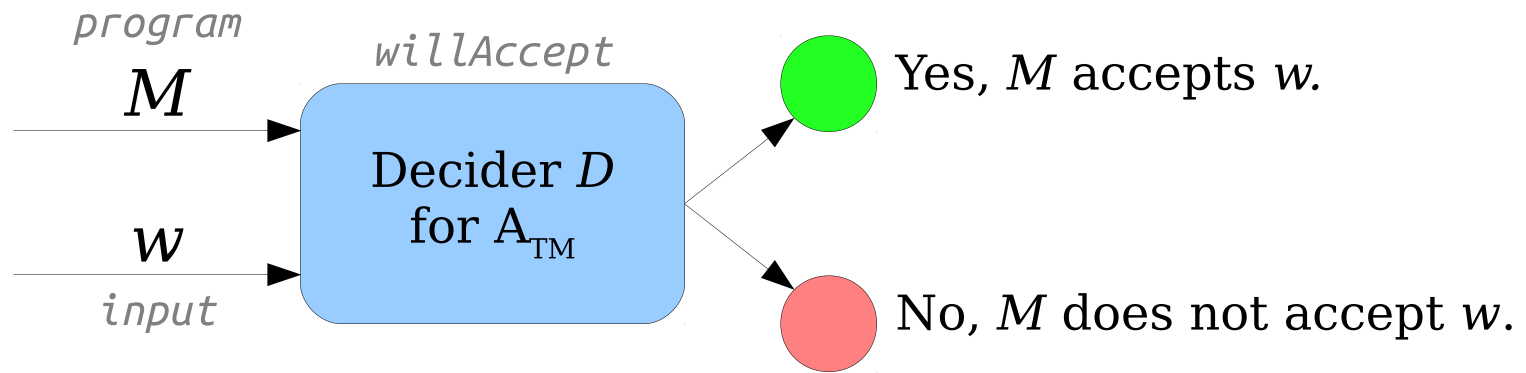
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!



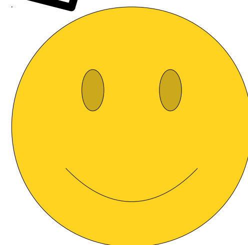

```
bool willAccept(string program, string input)
```

Program P design specification:

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

```
// Program P

int main() {
  string input = getInput();

}
```

That means we need to be able to figure out whether we're going to accept.



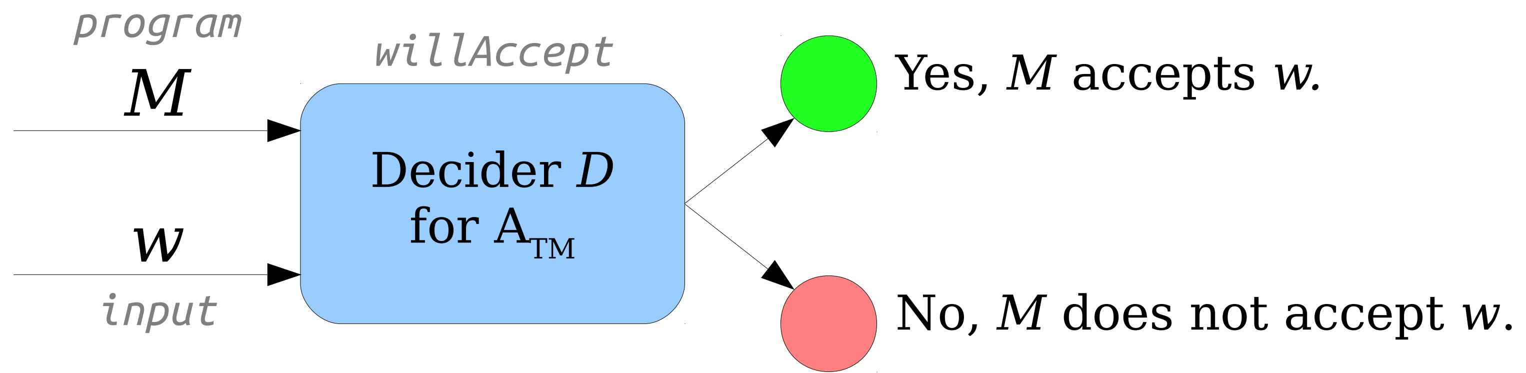
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!



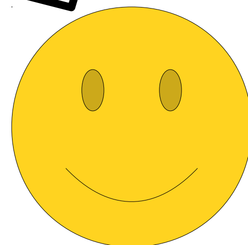

```
bool willAccept(string program, string input)
```

Program P design specification:

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

```
// Program P

int main() {
  string input = getInput();



}
```

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



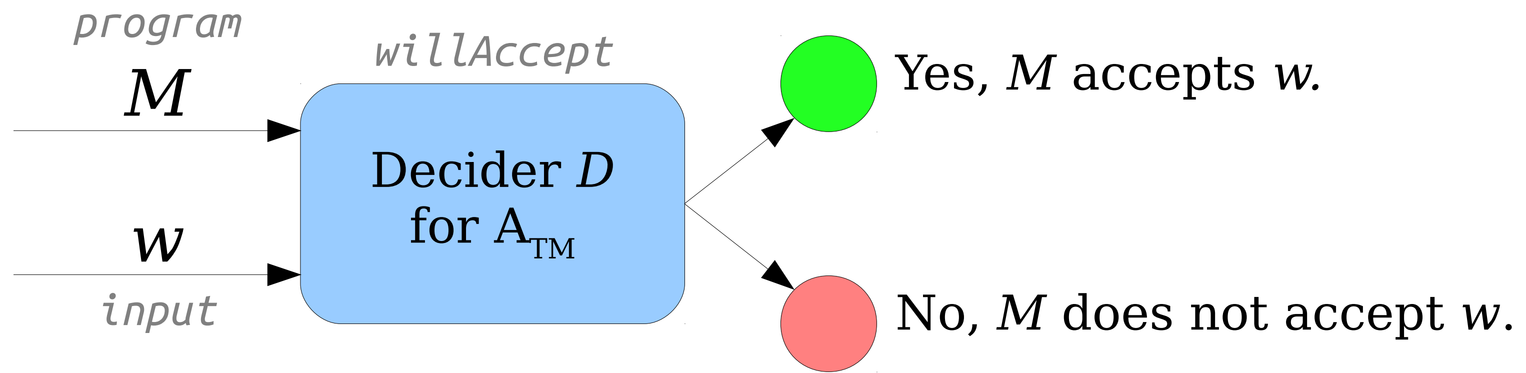
$A_{TM} \in \mathbf{R}$

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

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

Program *P* accepts its input if and only if program *P* does not accept its input

Contradiction!



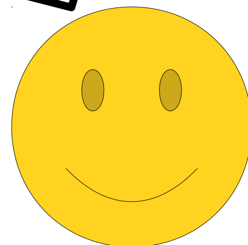

```
bool willAccept(string program, string input)
```

Program *P* design specification:

If *P* accepts its input, then
P does not accept its input.
If *P* does not accept its input, then
P accepts its input.

```
// Program P

int main() {
  string input = getInput();


}
```

What if we had program P ask whether it was going to accept something?



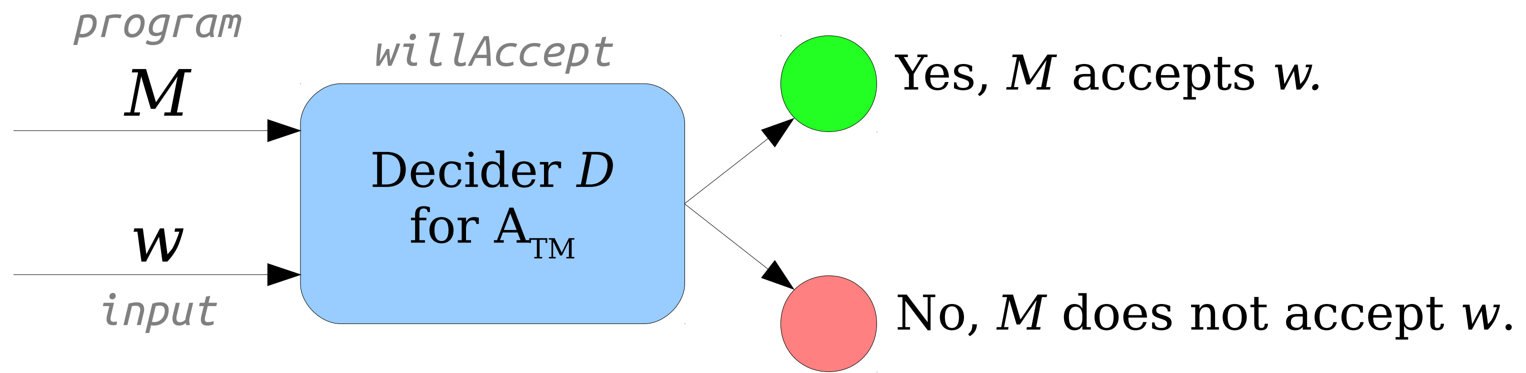
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!

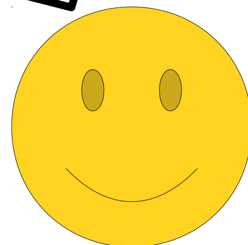



```
bool willAccept(string program, string input)
```

Program P design specification:

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

```
// Program P

int main() {
  string input = getInput();



}
```



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

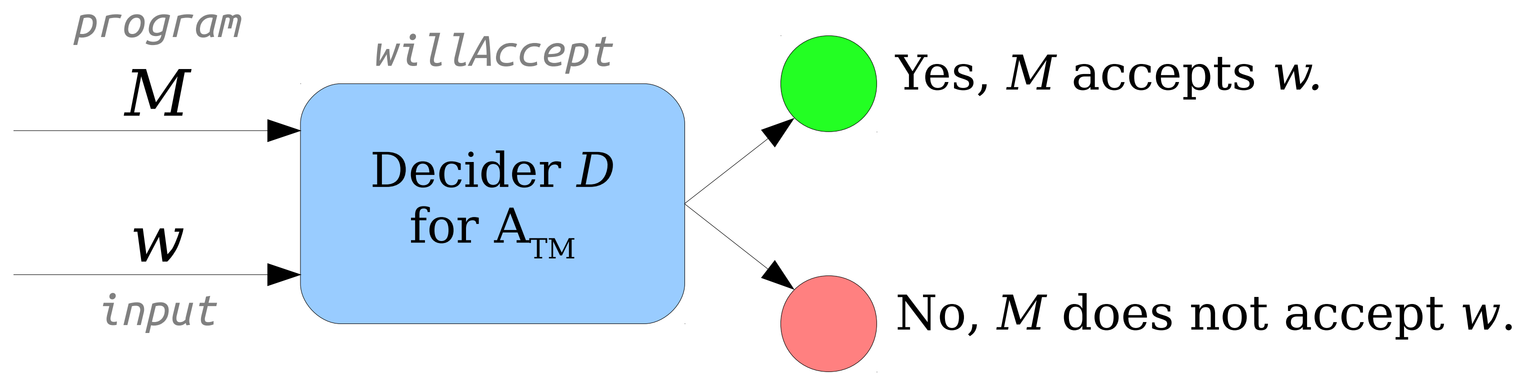
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!



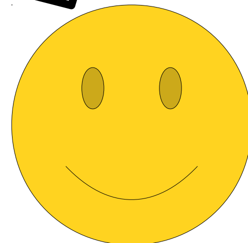

```
bool willAccept(string program, string input)
```

Program P design specification:

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();


}
```

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



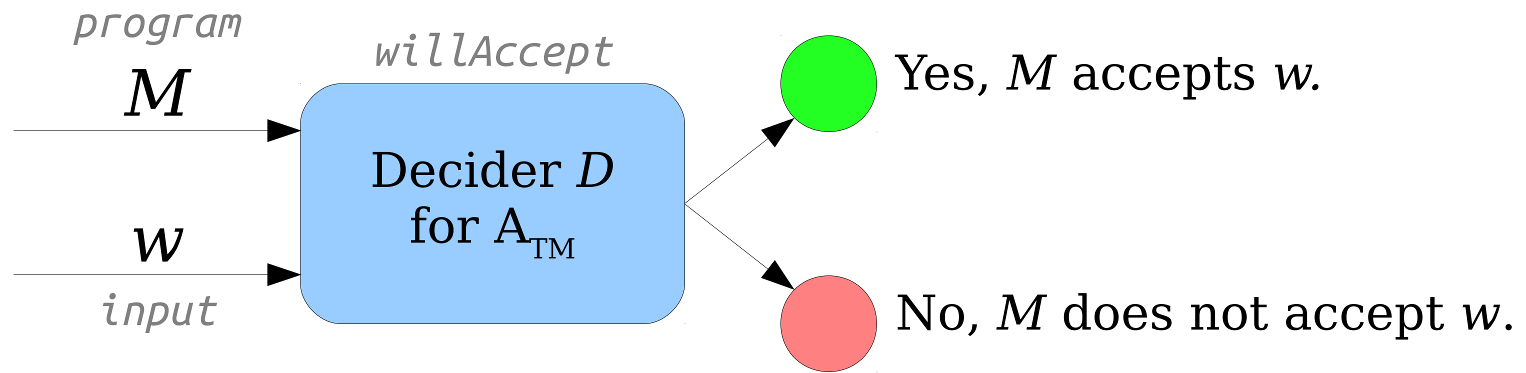
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!

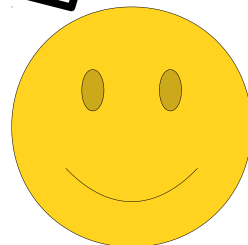



```
bool willAccept(string program, string input)
```

Program P design specification:

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {

  } else {

  }
}
```



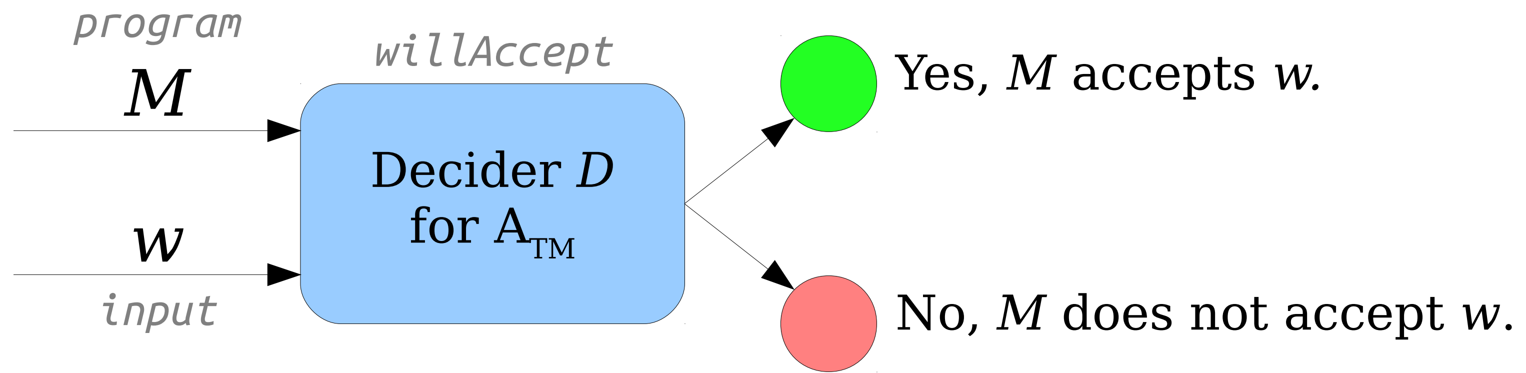
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!



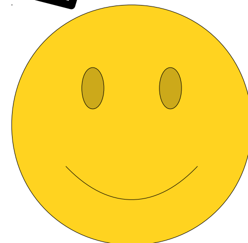

```
bool willAccept(string program, string input)
```

Program P design specification:

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {

  } else {

  }
}
```

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



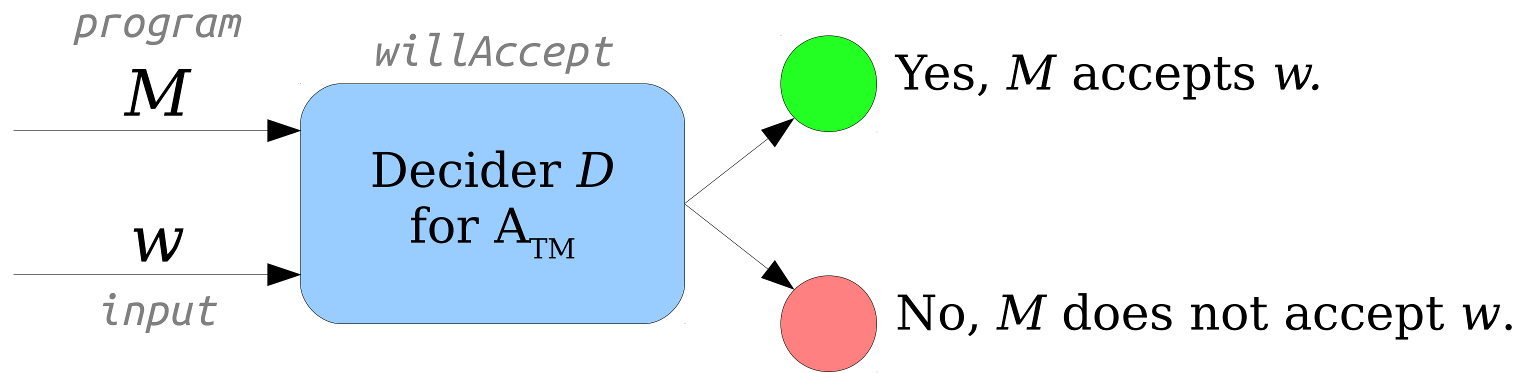
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!

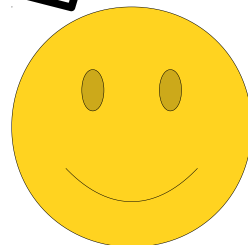



```
bool willAccept(string program, string input)
```

Program P design specification:

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {

  } else {

  }
}
```



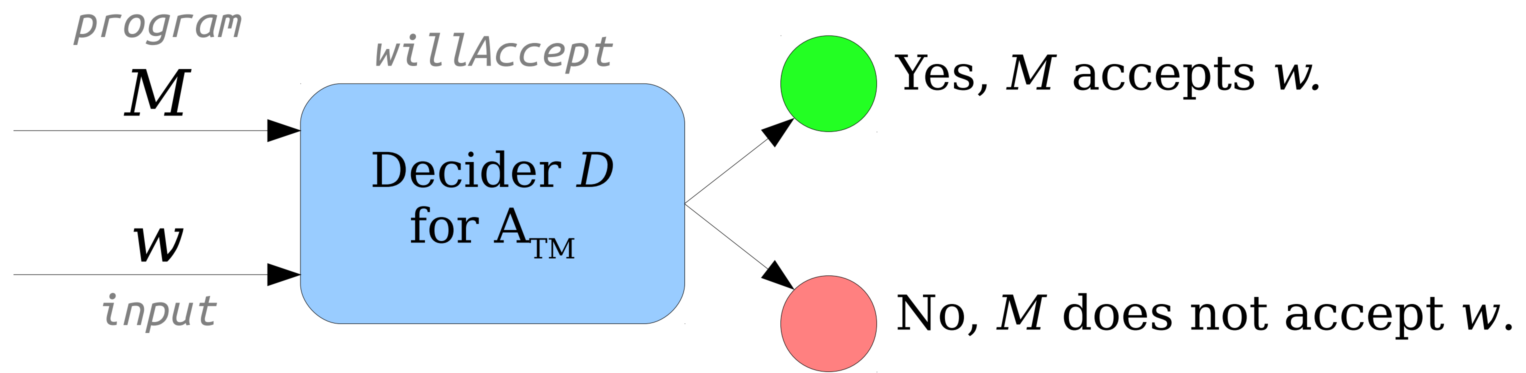
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!



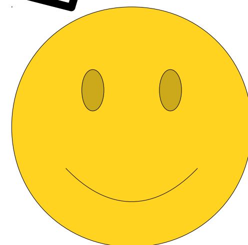

```
bool willAccept(string program, string input)
```

Program P design specification:

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {

  } else {

  }
}
```

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



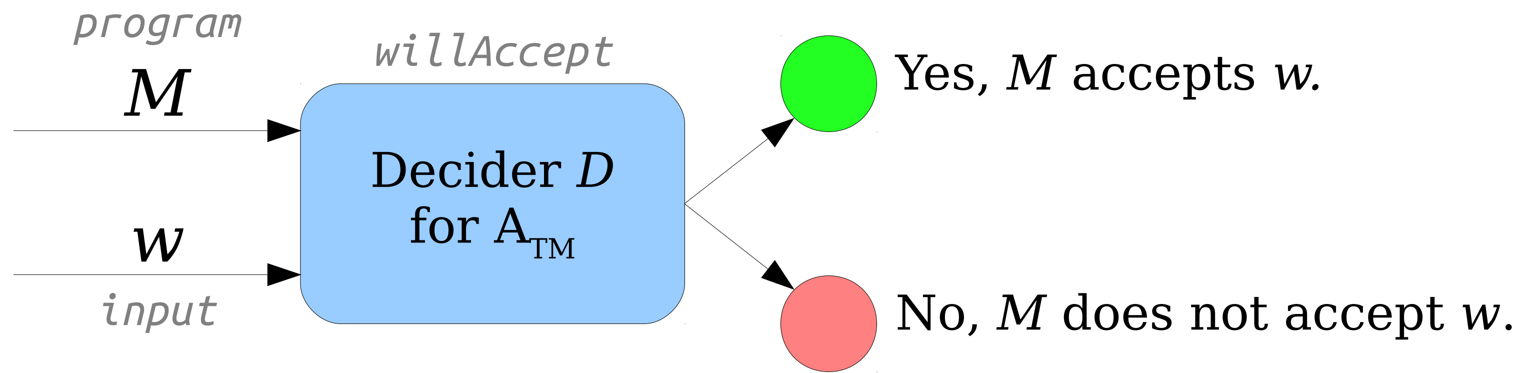
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!



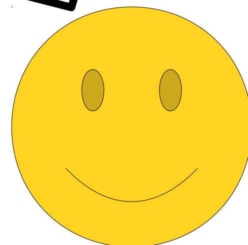

```
bool willAccept(string program, string input)
```

Program P design specification:

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {

  }
}
```

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



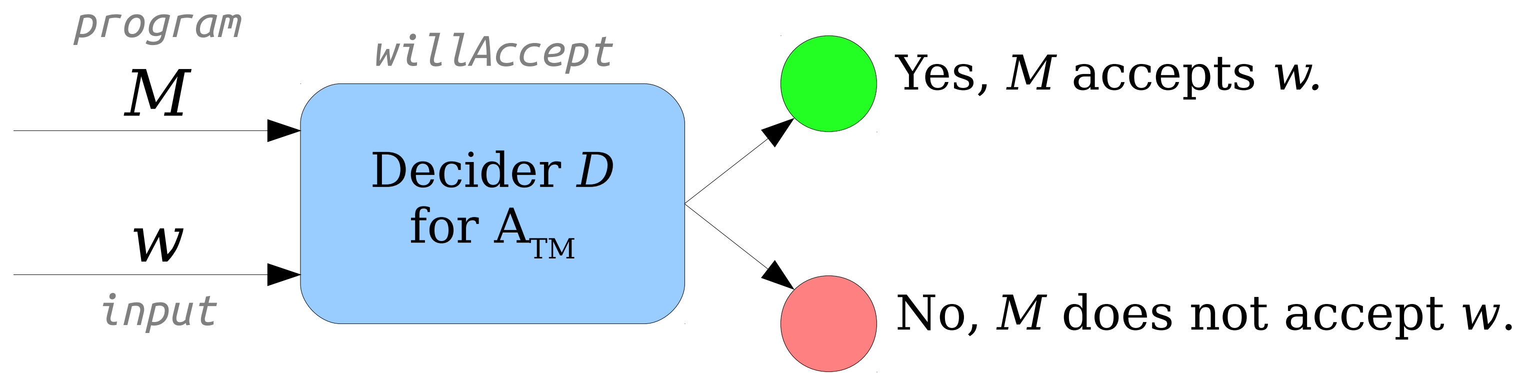
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

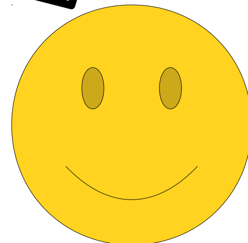
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {

  }
}
```

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



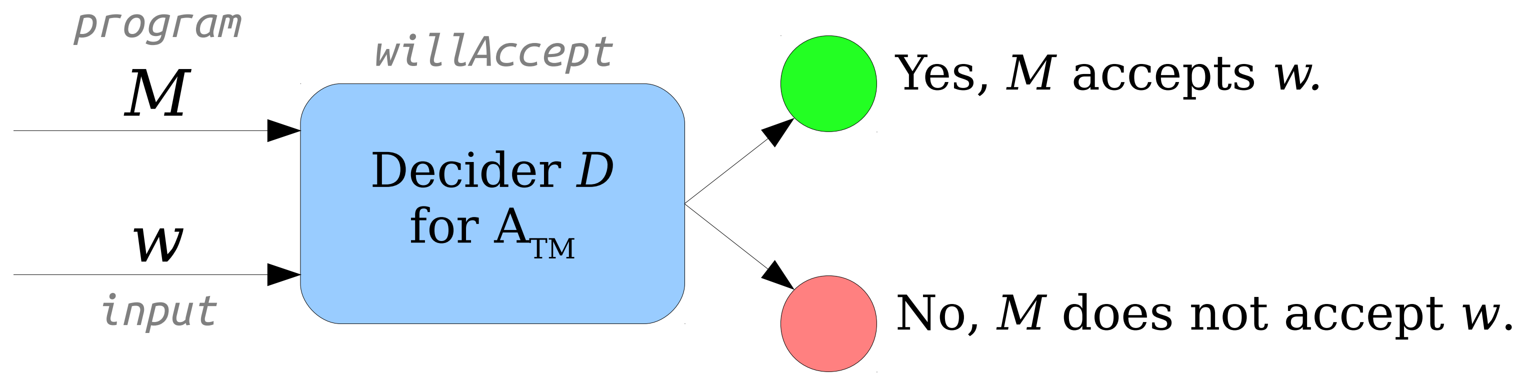
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

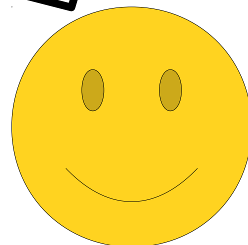
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {

  }
}
```

What about this part?



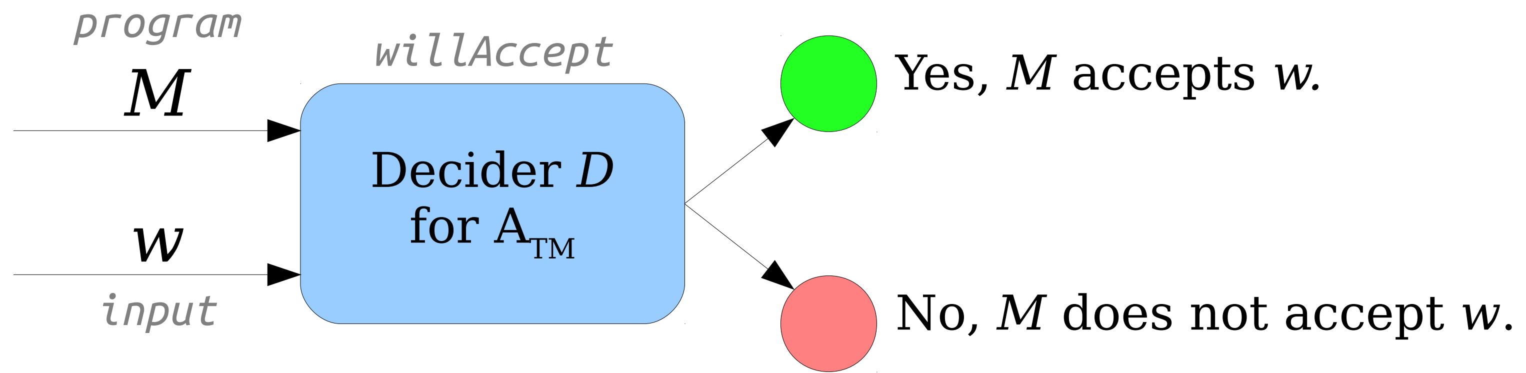
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

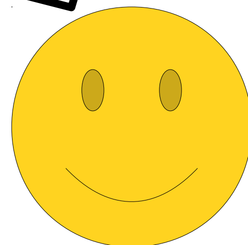
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {

  }
}
```

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



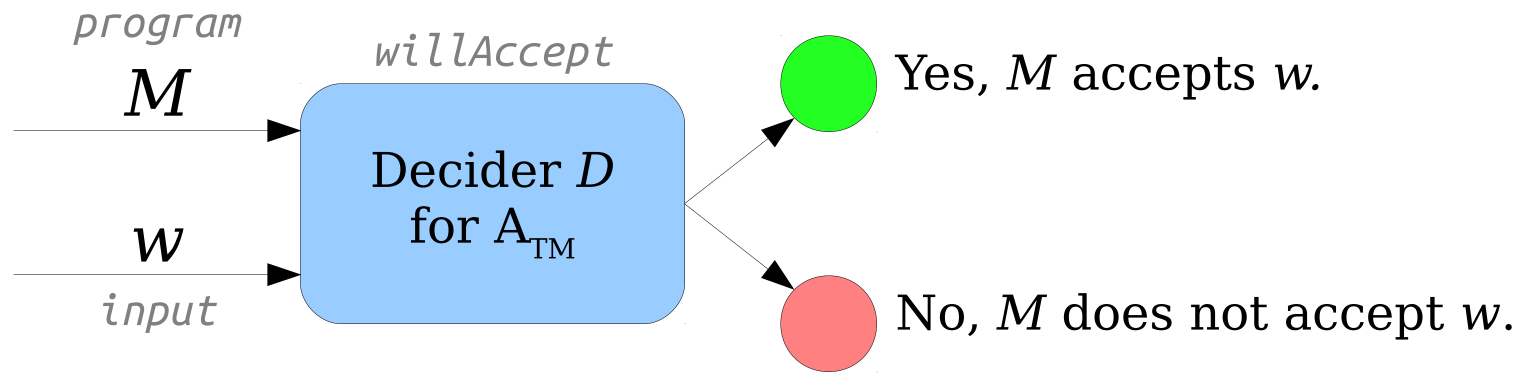
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

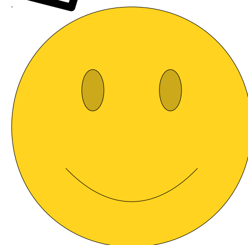
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```

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



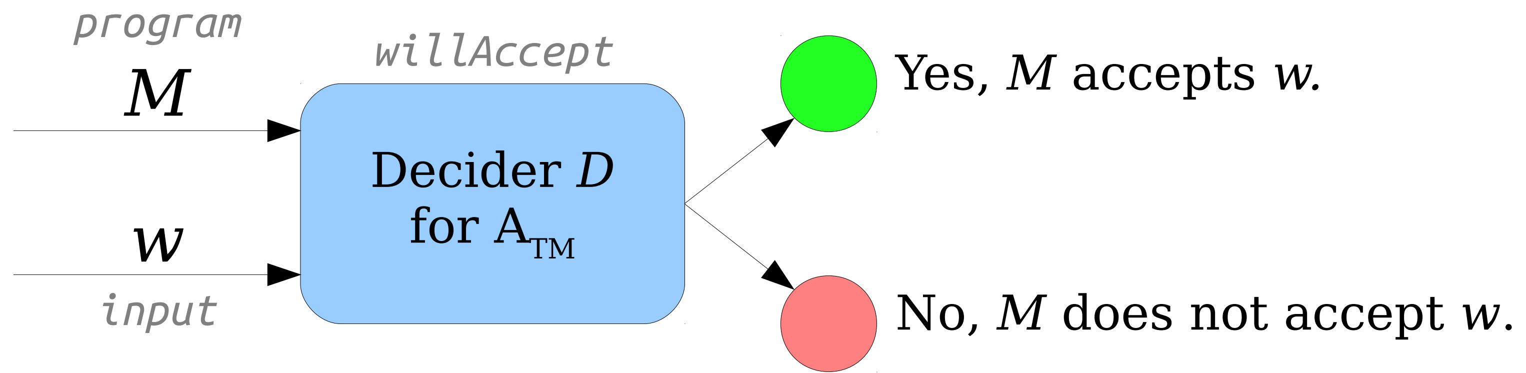
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

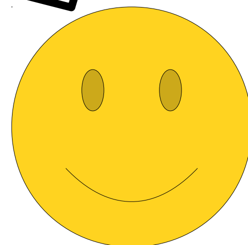
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```

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



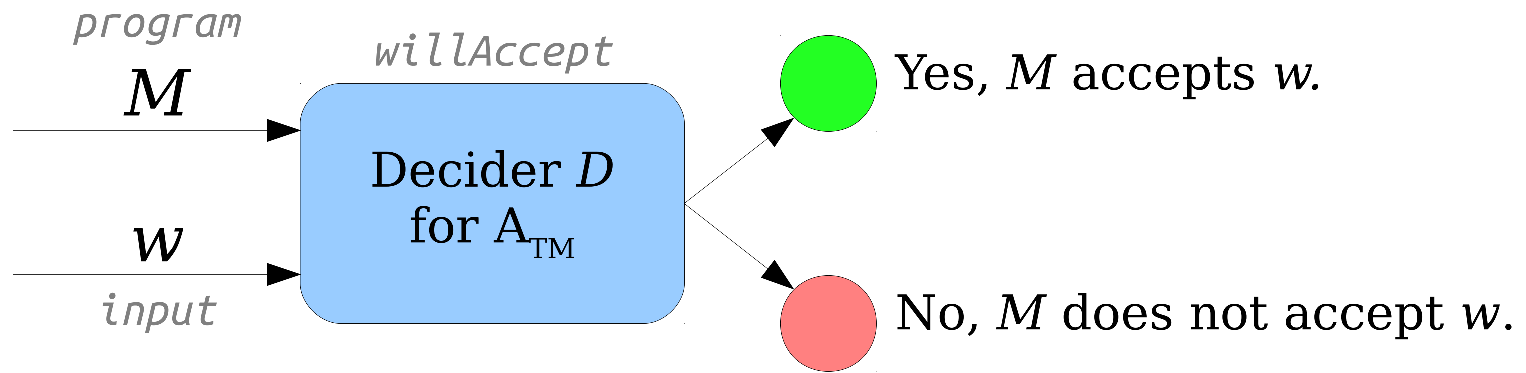
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

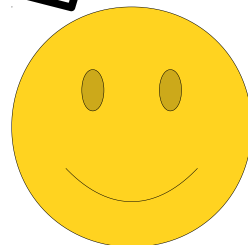
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```

So let's take a quick look over our program P.



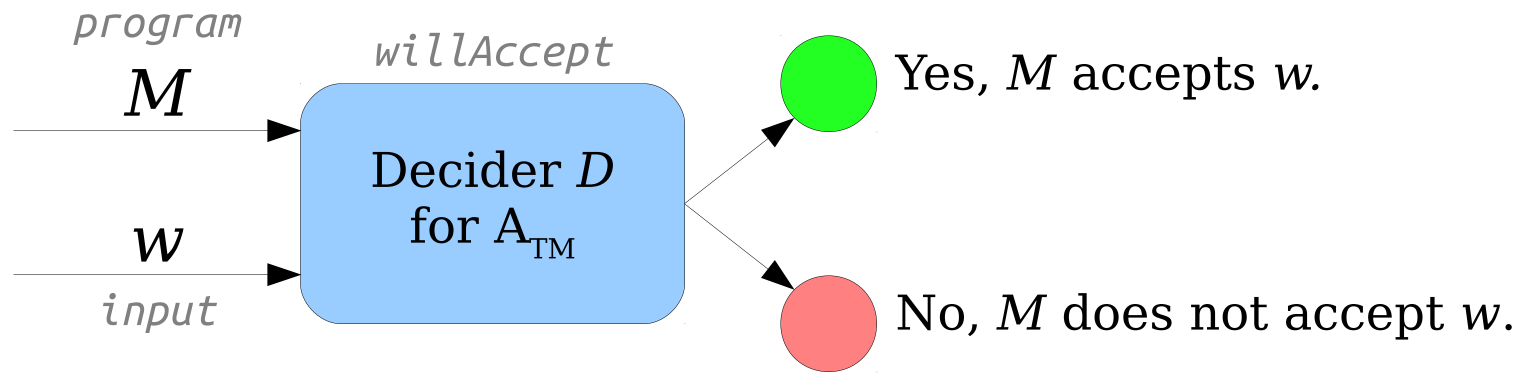
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

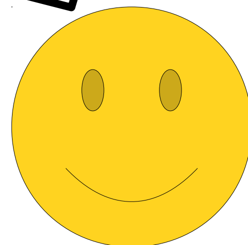
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```

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



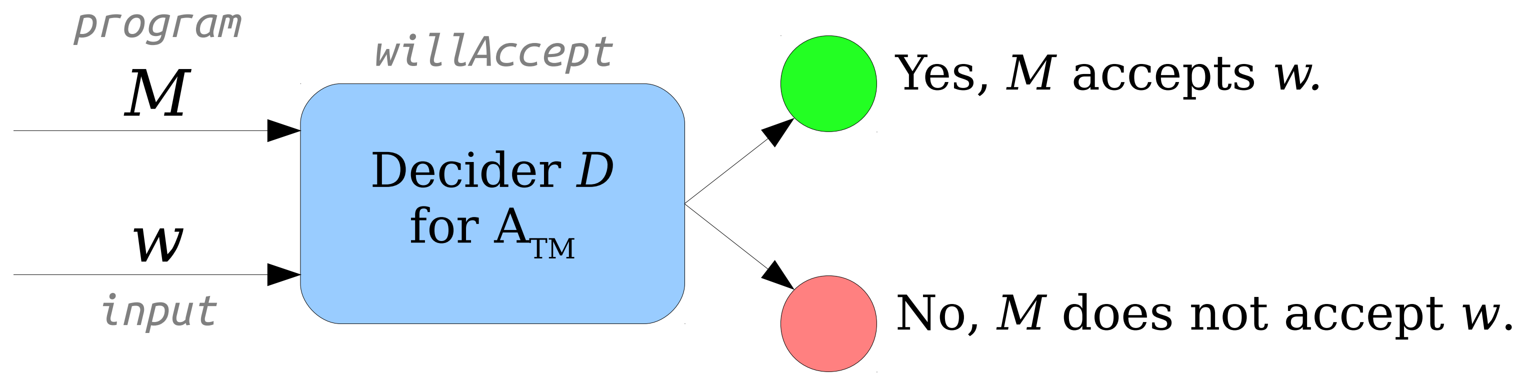
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

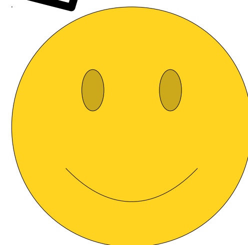
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```

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



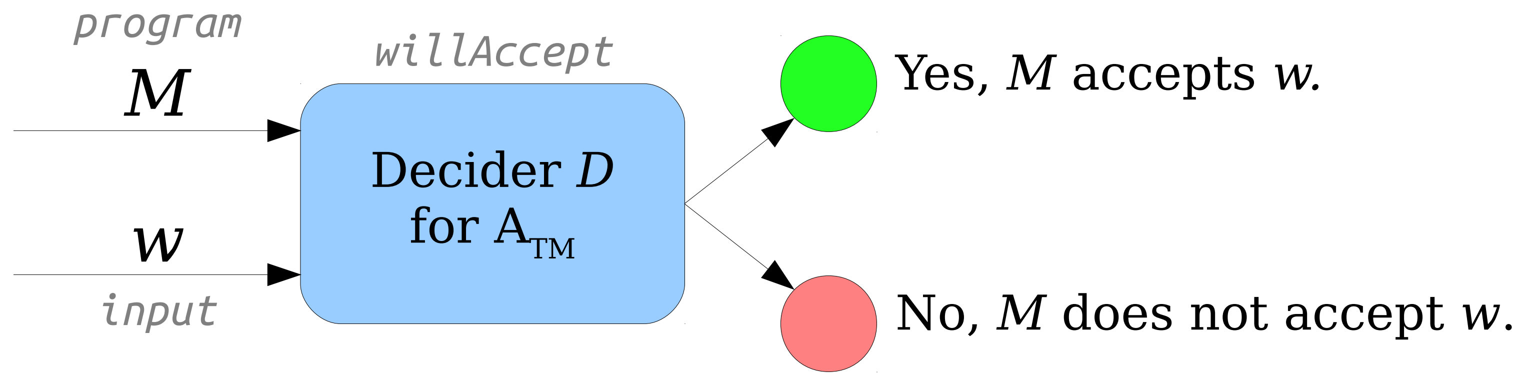
$A_{TM} \in \mathbf{R}$

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

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

Program *P* accepts its input if and only if program *P* does not accept its input

Contradiction!



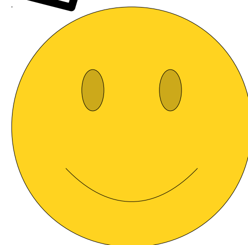

```
bool willAccept(string program, string input)
```

Program *P* design specification:

✓ If *P* accepts its input, then
P does not accept its input.

✓ If *P* does not accept its input, then
P accepts its input.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```



And, indeed, that's what we've done! There's a contradiction here because P accepts if and only if it doesn't accept.

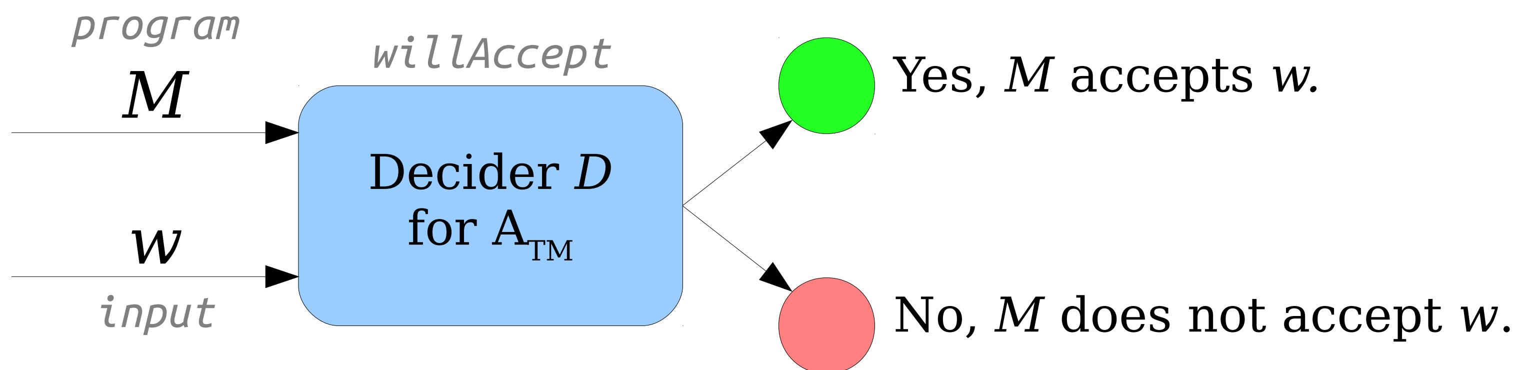
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

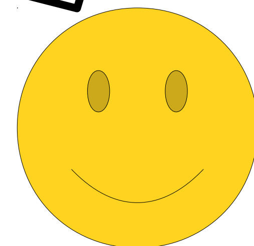
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```

So if you trace through the implications here...



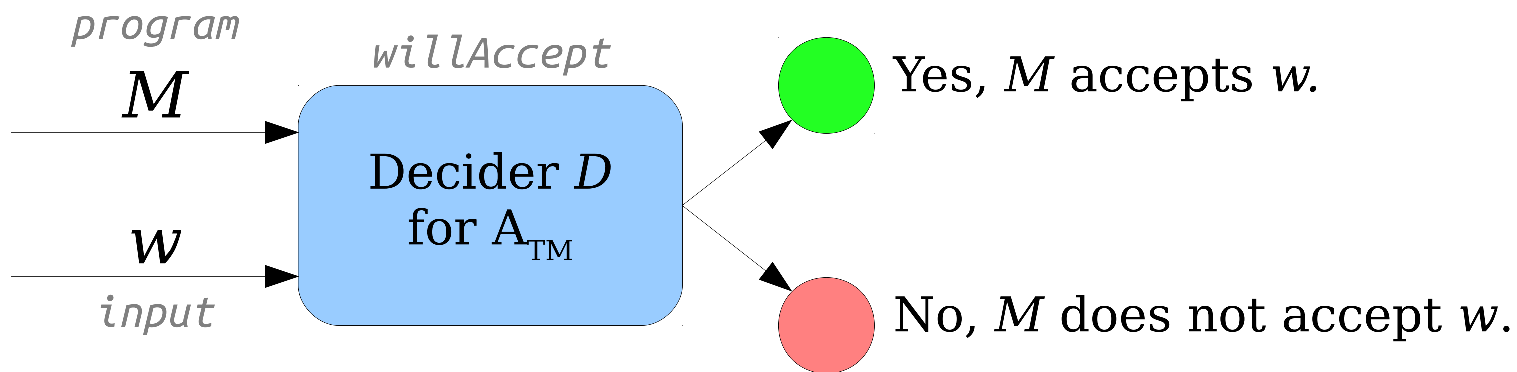
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

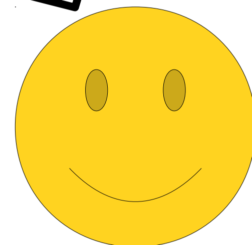
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```

So if you trace through the implications here…



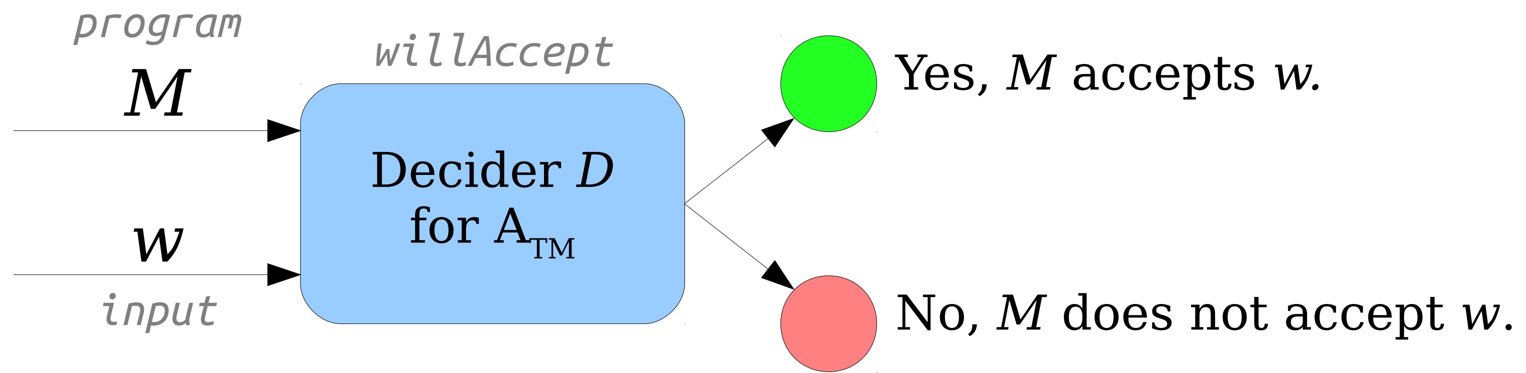
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

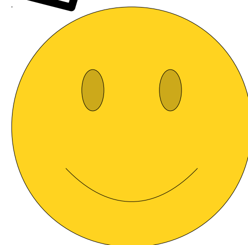
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```

So if you trace through the implications here…



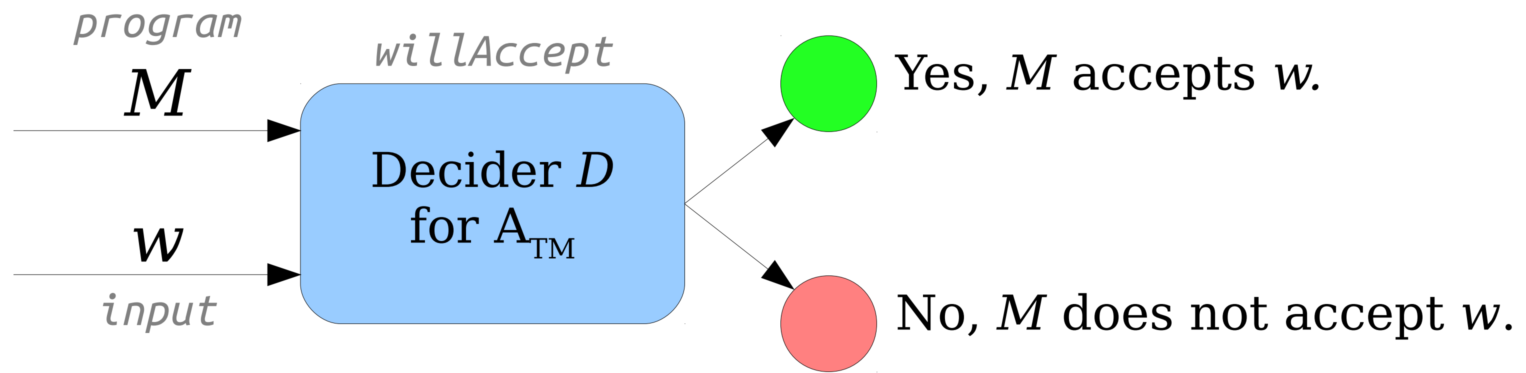
$A_{TM} \in \mathbf{R}$

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

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

Program *P* accepts its input if and only if program *P* does not accept its input

Contradiction!




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

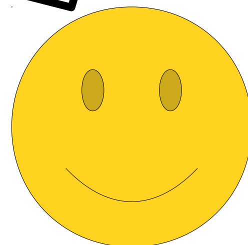
Program *P* design specification:

✓ If *P* accepts its input, then
P does not accept its input.

✓ If *P* does not accept its input, then
P accepts its input.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```

So if you trace through the implications here...



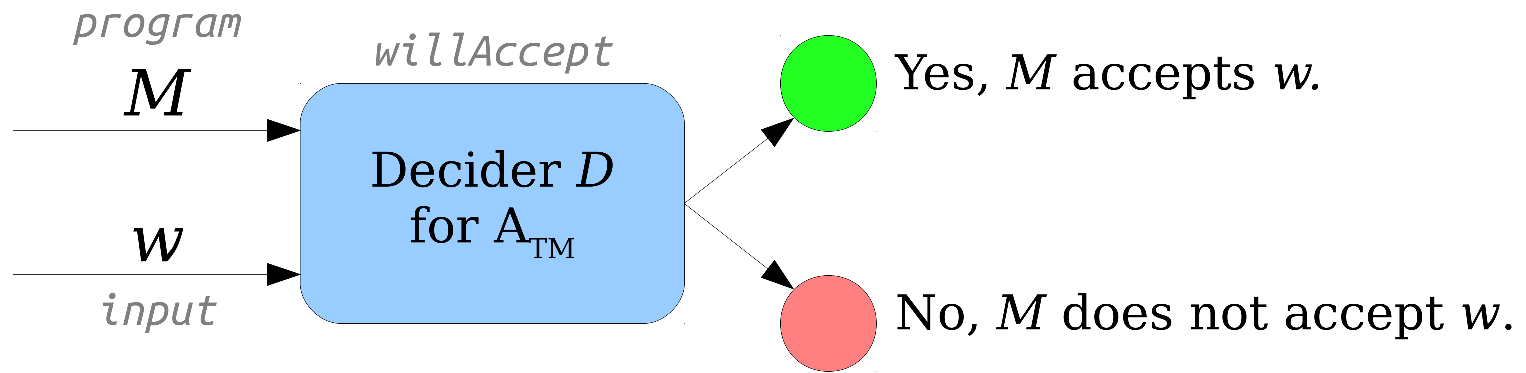
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

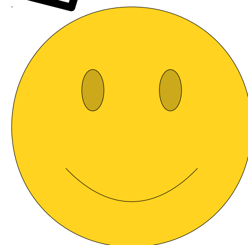
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```

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



What does this program do?

```
bool willAccept(string program, string input) {
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (willAccept(me, input)) {
      reject();
   } else {
      accept();
   }
}
```

Here's that initial lecture slide again.

Try running this program on any input. What happens if

… this program accepts its input?
It rejects the input!

… this program doesn't accept its input?
It accepts the input!

What does this program do?

```
bool willAccept(string program, string input) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

Take a look at it more closely.

Try running this program on any input.
What happens if

… this program accepts its input?
It rejects the input!

… this program doesn't accept its input?
It accepts the input!

What does this program do?

```
bool willAccept(string program, string input) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

Recognize this code? Now you know where it comes from!

Try running this program on any input. What happens if

… this program accepts its input?
It rejects the input!

… this program doesn't accept its input?
It accepts the input!

What does this program do?

```
bool willAccept(string program, string input) {
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

Try running this program on any input.
What happens if

… this program accepts its input?
It rejects the input!

… this program doesn't accept its input?
It accepts the input!

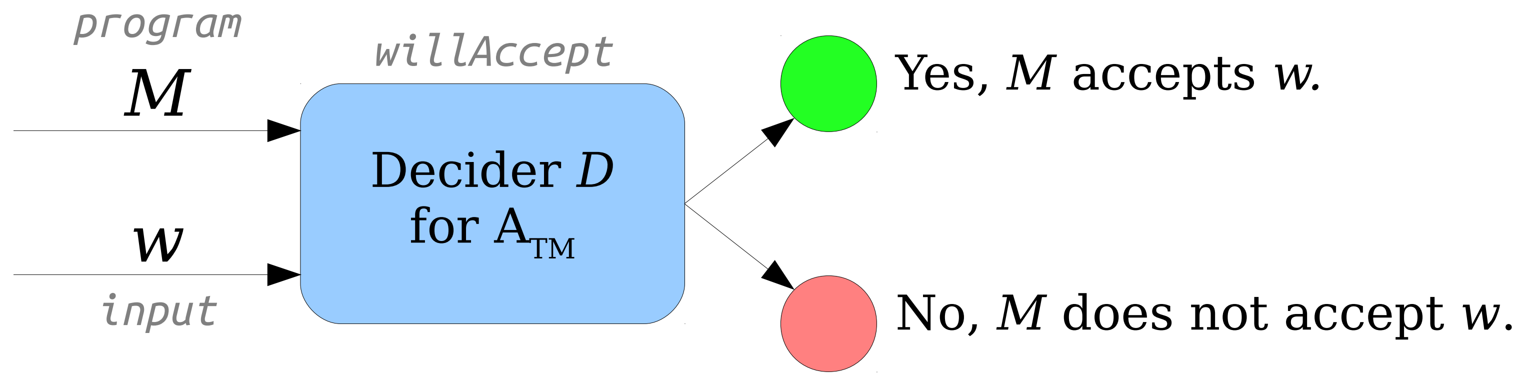
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

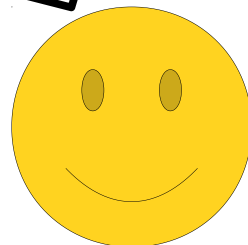
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```

This might seem like a lot – and in many ways it is.



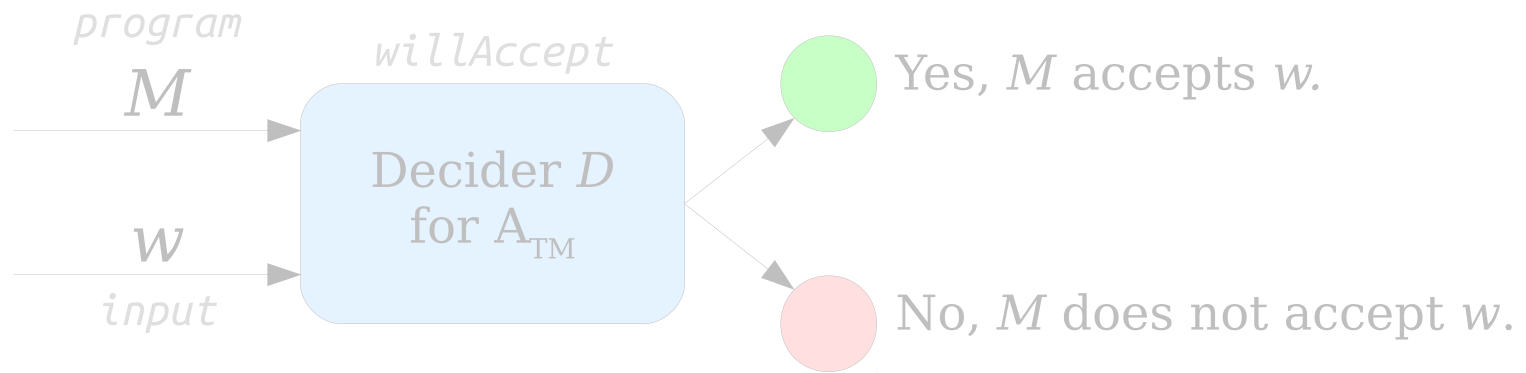
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

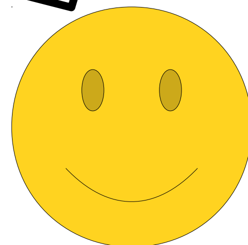
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```

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



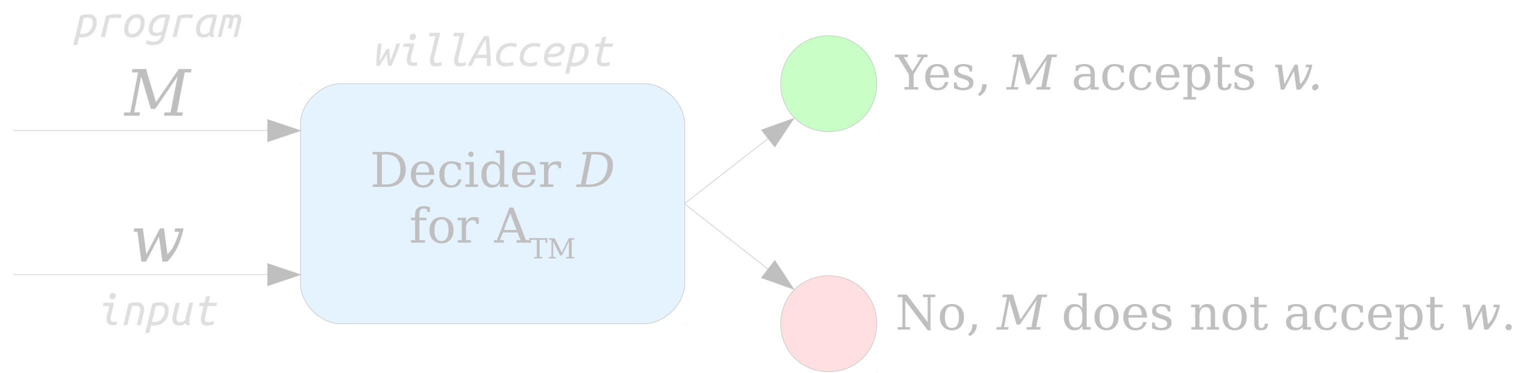
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

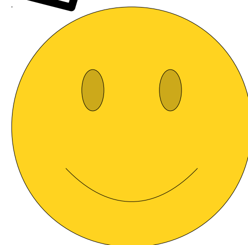
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```

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



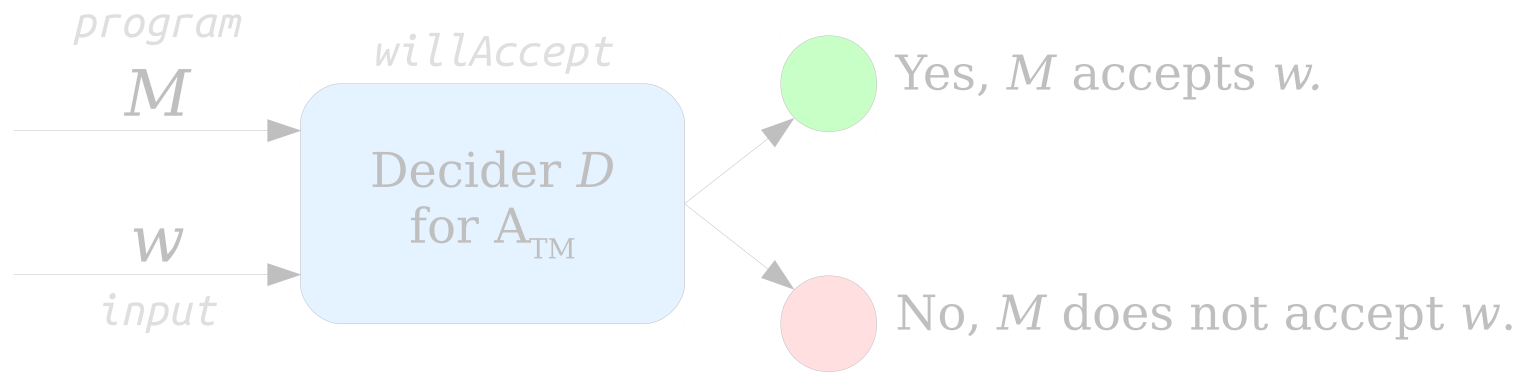
$A_{TM} \in \mathbf{R}$

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

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

Program *P* accepts its input if and only if program *P* does not accept its input

Contradiction!




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

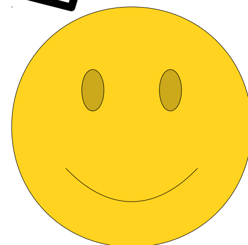
Program *P* design specification:

✓ If *P* accepts its input, then *P* does not accept its input.

✓ If *P* does not accept its input, then *P* accepts its input.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```

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



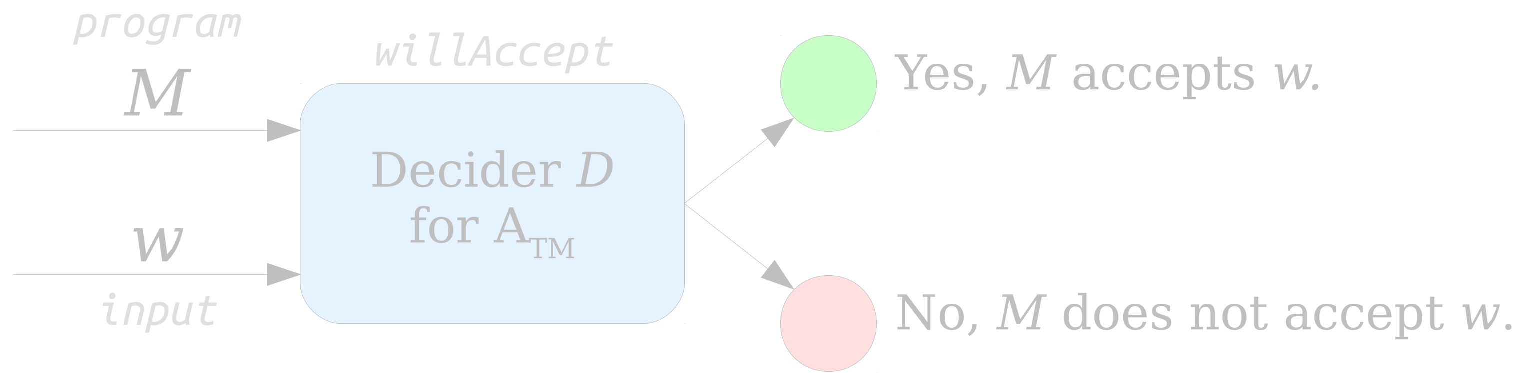
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!



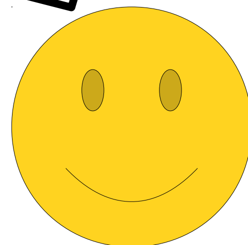

```
bool willAccept(string program, string input)
```

Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```



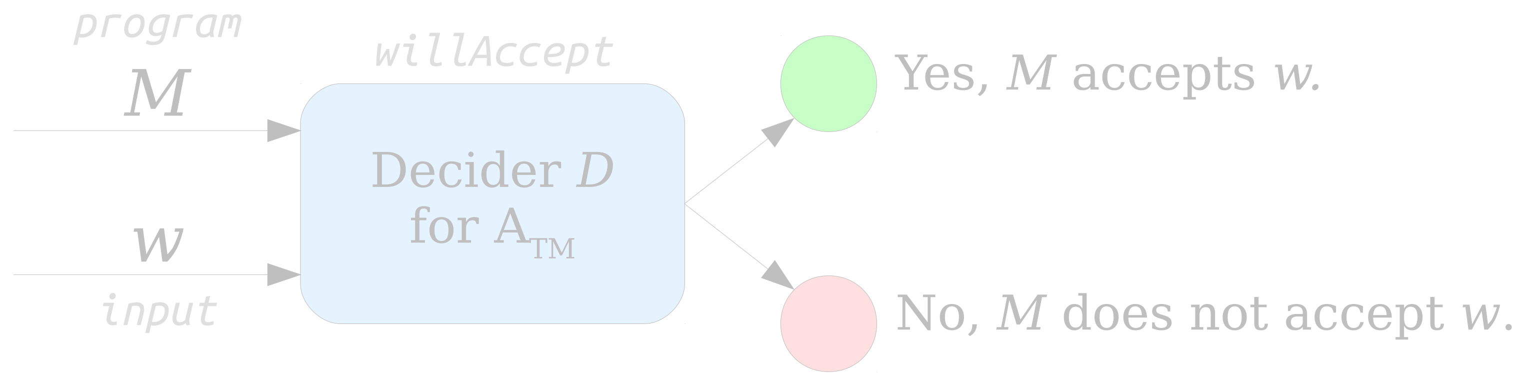
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!



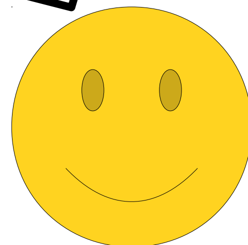

```
bool willAccept(string program, string input)
```

Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```



First, let's jump back to this part of the program P that we wrote.

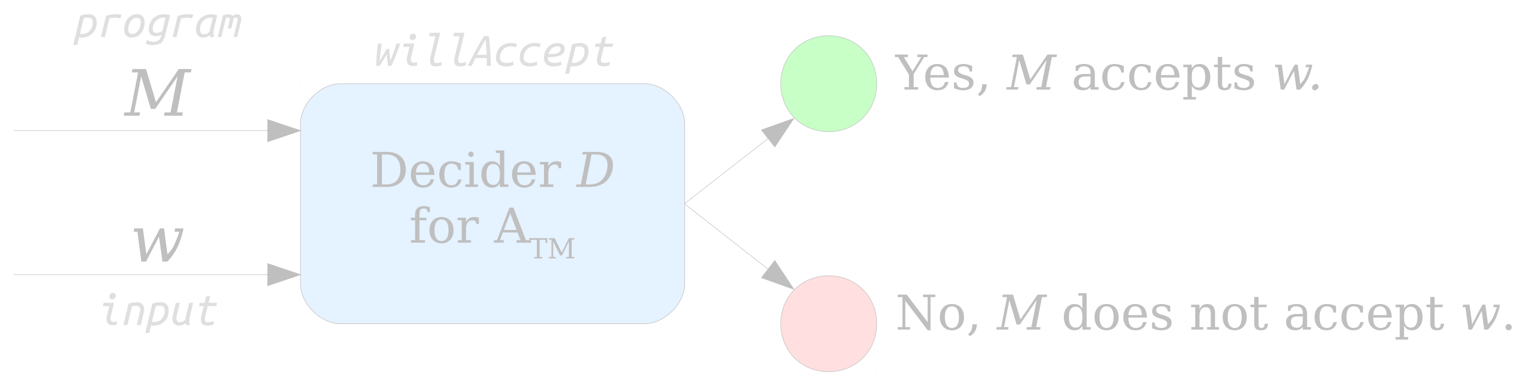
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

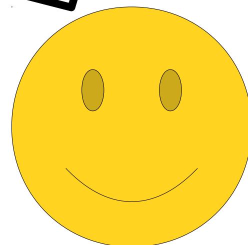
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```

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



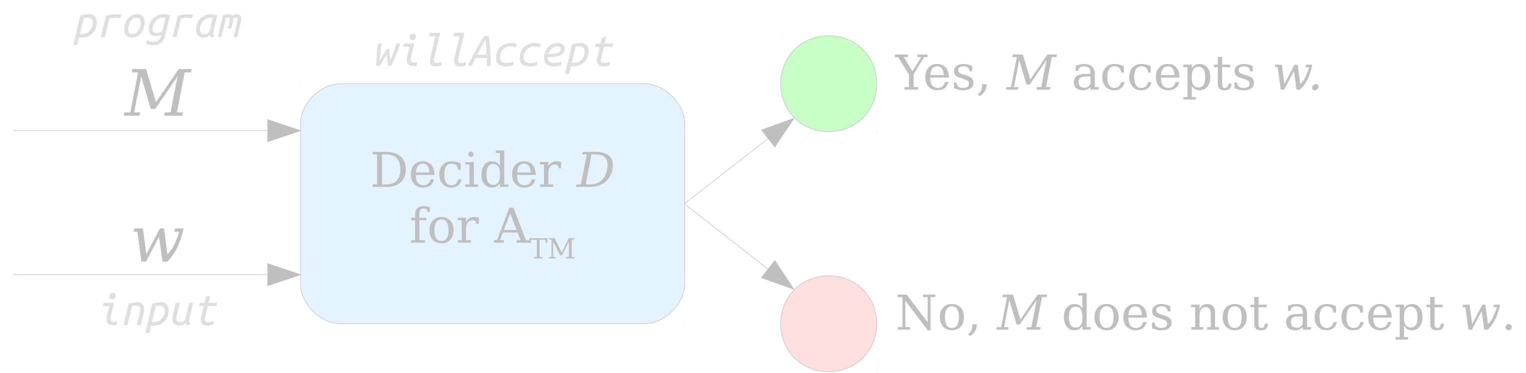
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

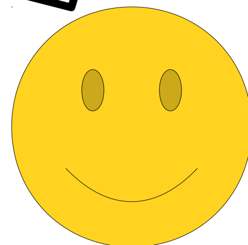
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```

Here, the specific way we ended up doing that was by having program P reject its input.



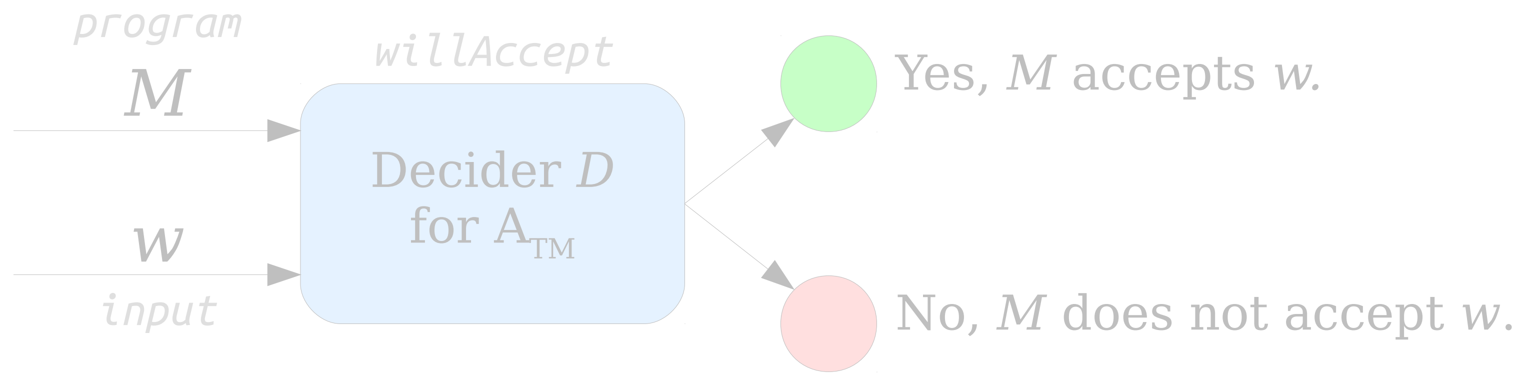
$A_{TM} \in \mathbf{R}$

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

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

Program *P* accepts its input if and only if program *P* does not accept its input

Contradiction!




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

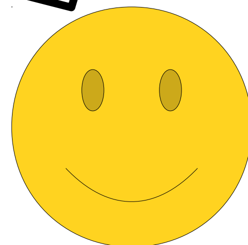
Program *P* design specification:

✓ If *P* accepts its input, then *P* does not accept its input.

✓ If *P* does not accept its input, then *P* accepts its input.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```

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



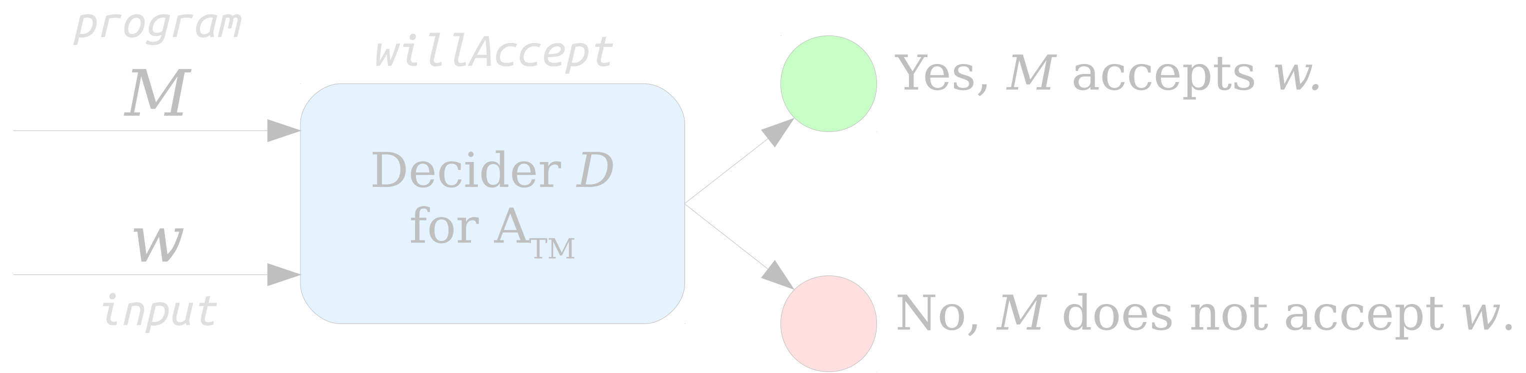
$A_{TM} \in \mathbf{R}$

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

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

Program *P* accepts its input if and only if program *P* does not accept its input

Contradiction!




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

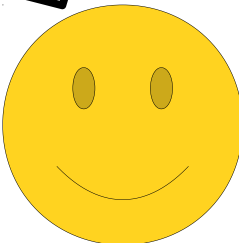
Program *P* design specification:

✓ If *P* accepts its input, then *P* does not accept its input.

✓ If *P* does not accept its input, then *P* accepts its input.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    while (true) { }
  } else {
    accept();
  }
}
```

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



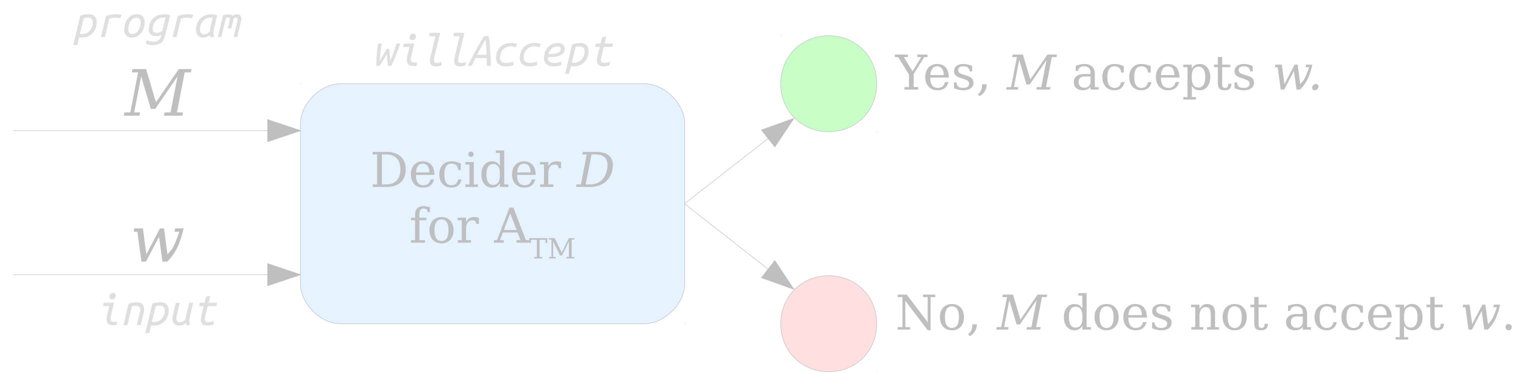
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

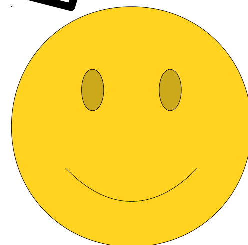
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    while (true) { }
  } else {
    accept();
  }
}
```

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



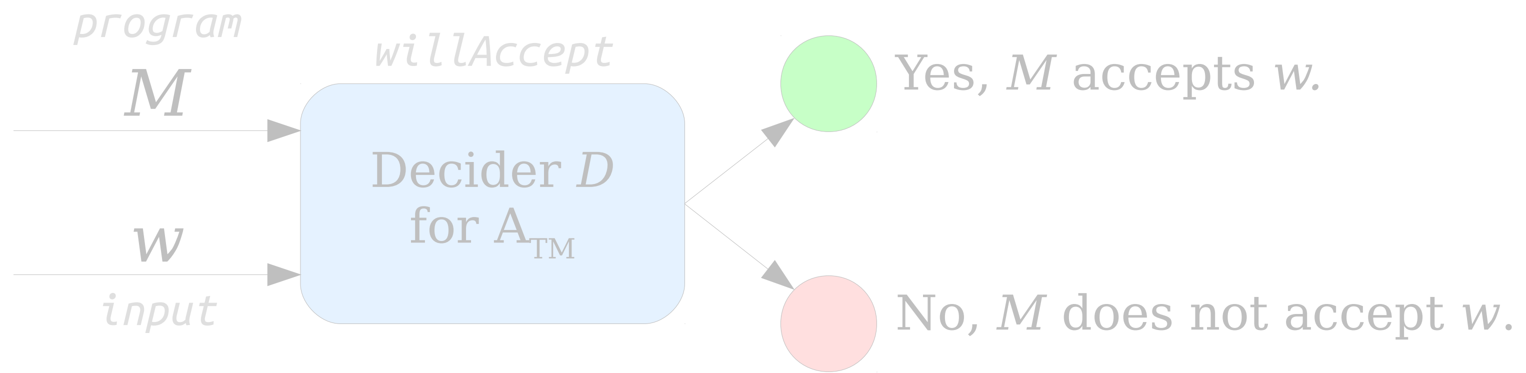
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

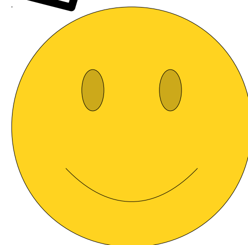
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    while (true) { }
  } else {
    accept();
  }
}
```

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



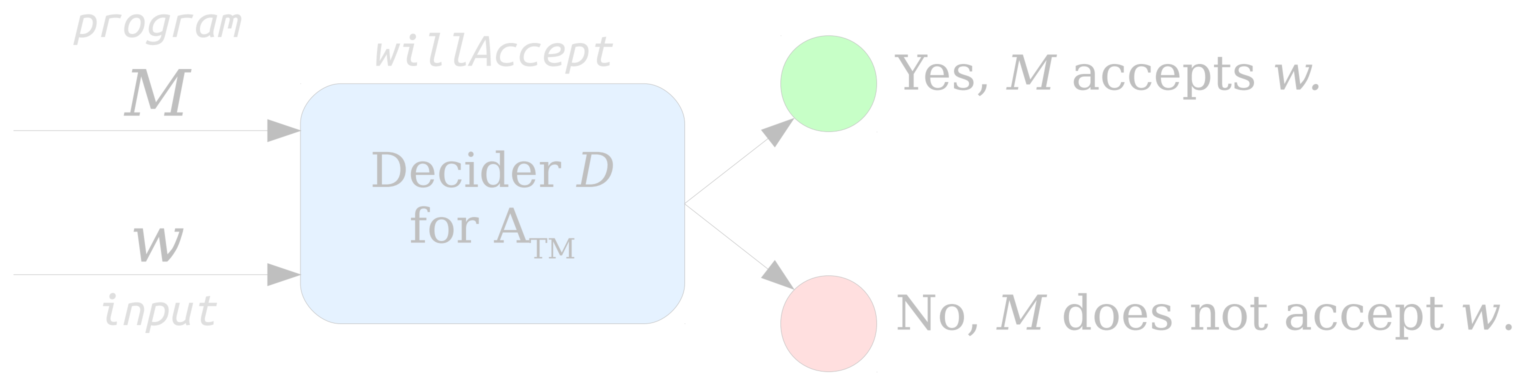
$A_{TM} \in \mathbf{R}$

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

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

Program *P* accepts its input if and only if program *P* does not accept its input

Contradiction!




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

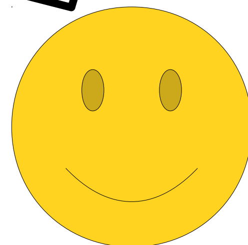
Program *P* design specification:

✓ If *P* accepts its input, then *P* does not accept its input.

✓ If *P* does not accept its input, then *P* accepts its input.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    while (true) { }
  } else {
    accept();
  }
}
```

Turns out, this is totally fine!



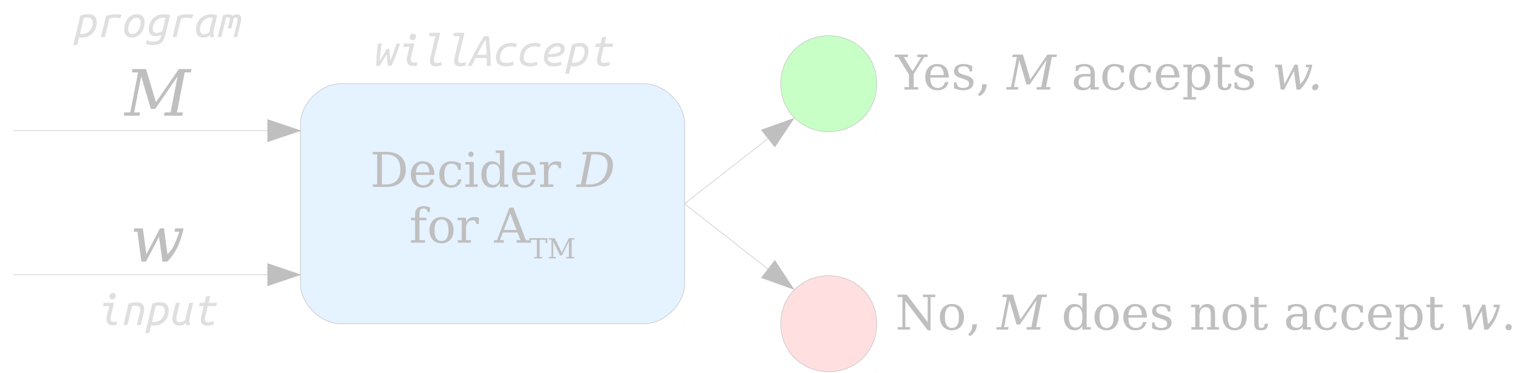
$A_{TM} \in \mathbf{R}$

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

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

Program *P* accepts its input if and only if program *P* does not accept its input

Contradiction!




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

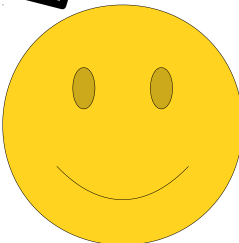
Program *P* design specification:

✓ If *P* accepts its input, then
P does not accept its input.

✓ If *P* does not accept its input, then
P accepts its input.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    while (true) { }
  } else {
    accept();
  }
}
```

There are two different programs here.



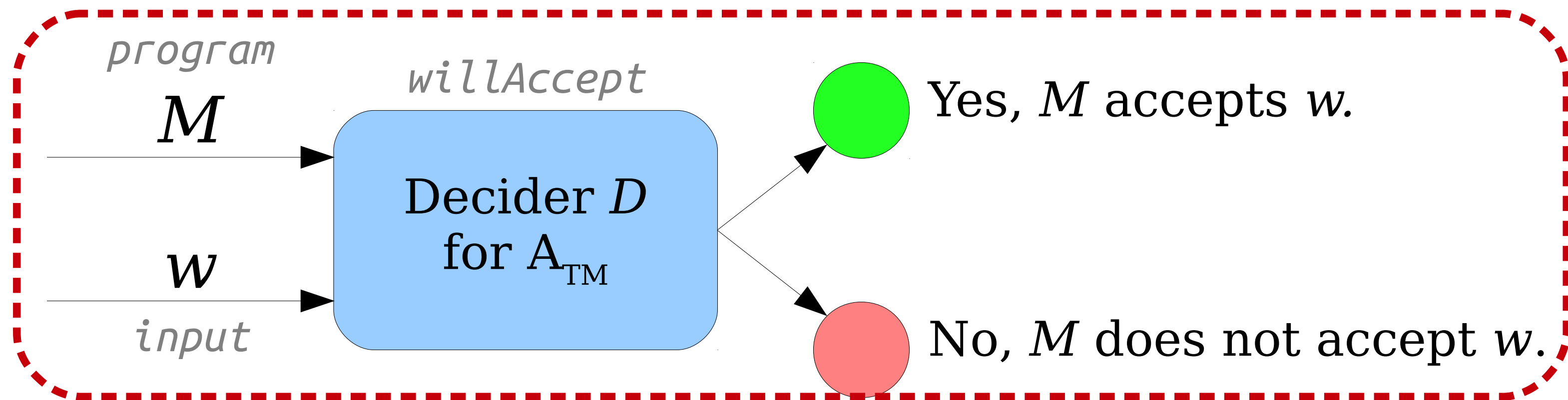
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

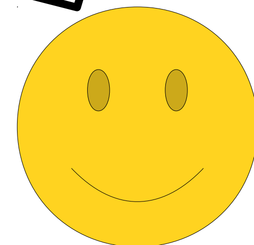
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    while (true) { }
  } else {
    accept();
  }
}
```

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



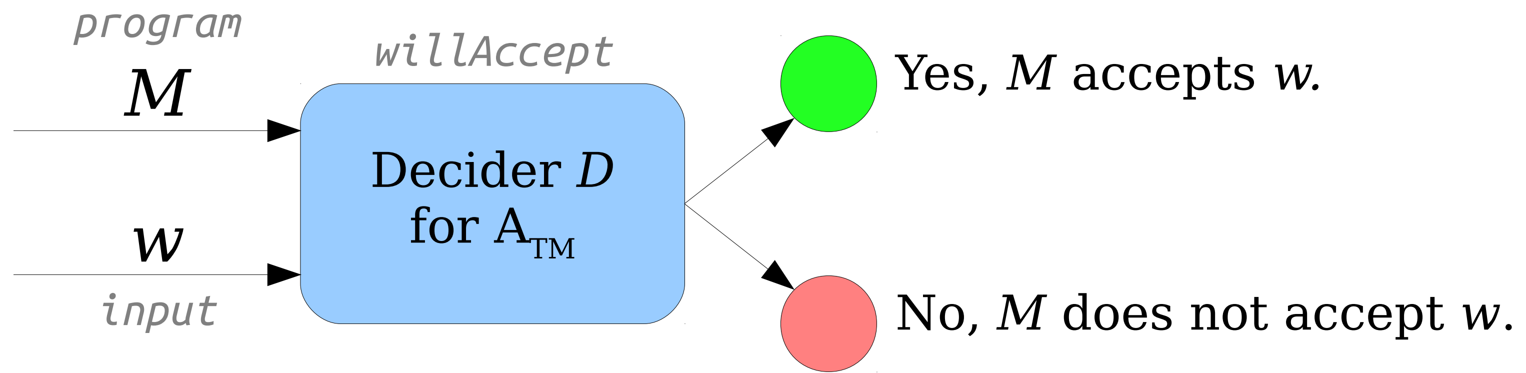
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

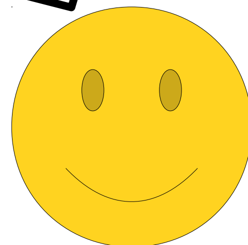
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    while (true) { }
  } else {
    accept();
  }
}
```

There's also this program P. Program P isn't the decider for A_{TM}, so it's not required to halt on all inputs.



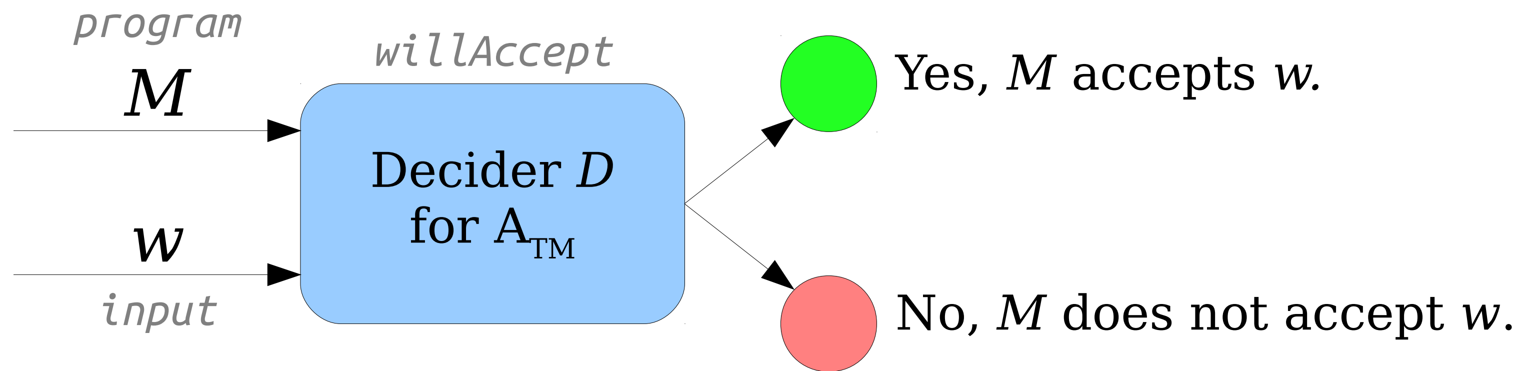
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

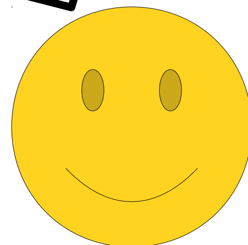
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    while (true) { }
  } else {
    accept();
  }
}
```

Going forward, remember that these proofs involve two different programs: the decider for the language, and the self-referential program.



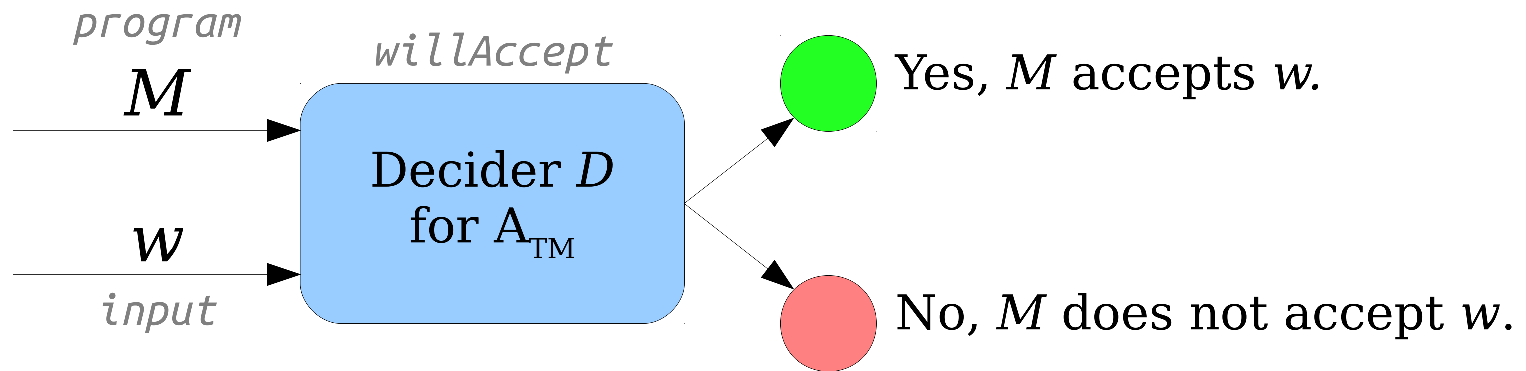
$A_{TM} \in \mathbf{R}$

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

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

Program *P* accepts its input if and only if program *P* does not accept its input

Contradiction!




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

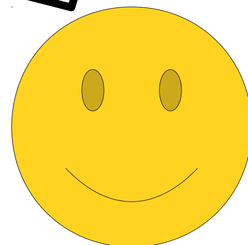
Program *P* design specification:

✓ If *P* accepts its input, then
P does not accept its input.

✓ If *P* does not accept its input, then
P accepts its input.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    while (true) { }
  } else {
    accept();
  }
}
```

The decider is always required to halt, but the program P is not.



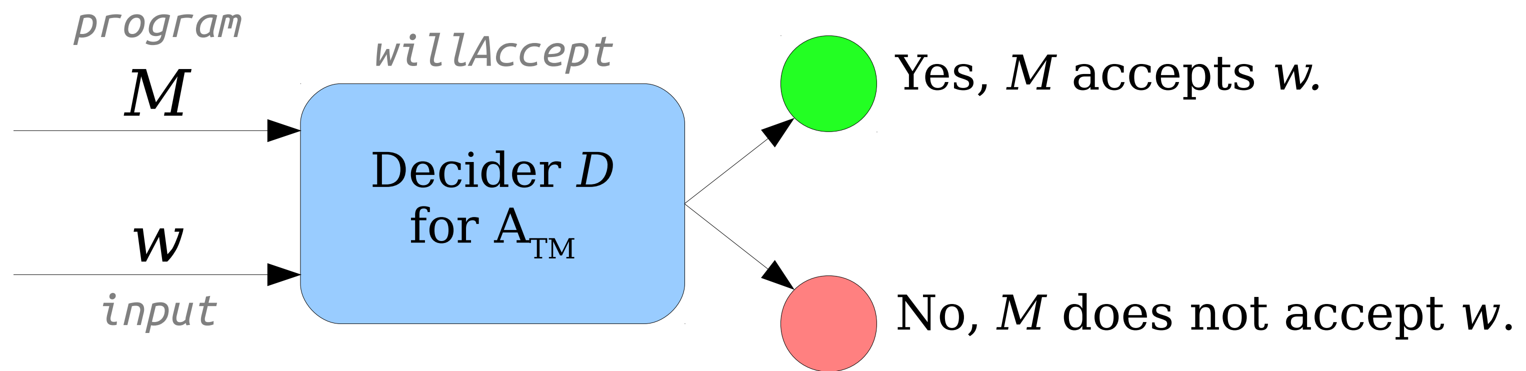
$A_{TM} \in \mathbf{R}$

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

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

Program *P* accepts its input if and only if program *P* does not accept its input

Contradiction!




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

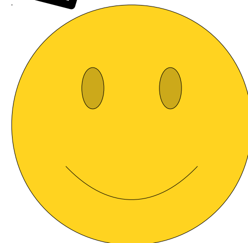
Program *P* design specification:

✓ If *P* accepts its input, then
P does not accept its input.

✓ If *P* does not accept its input, then
P accepts its input.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    while (true) { }
  } else {
    accept();
  }
}
```

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



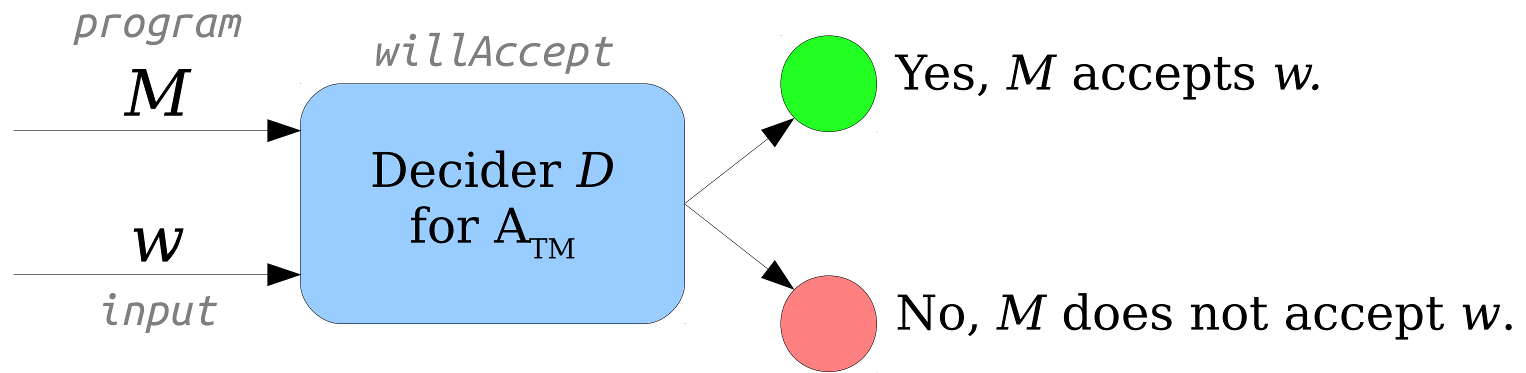
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

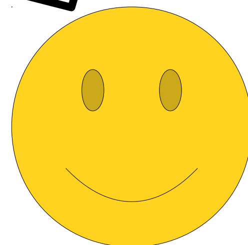
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```

Much better!



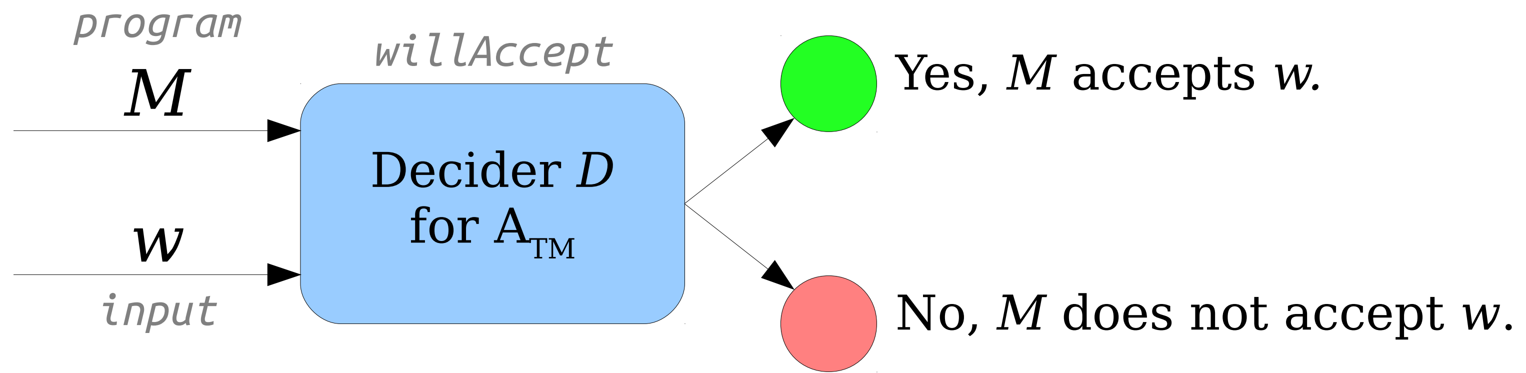
$A_{TM} \in \mathbf{R}$

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

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

Program *P* accepts its input if and only if program *P* does not accept its input

Contradiction!




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

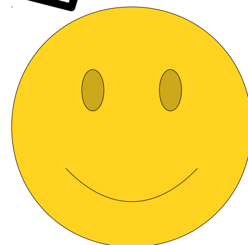
Program *P* design specification:

✓ If *P* accepts its input, then *P* does not accept its input.

✓ If *P* does not accept its input, then *P* accepts its input.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```

On to the next question.



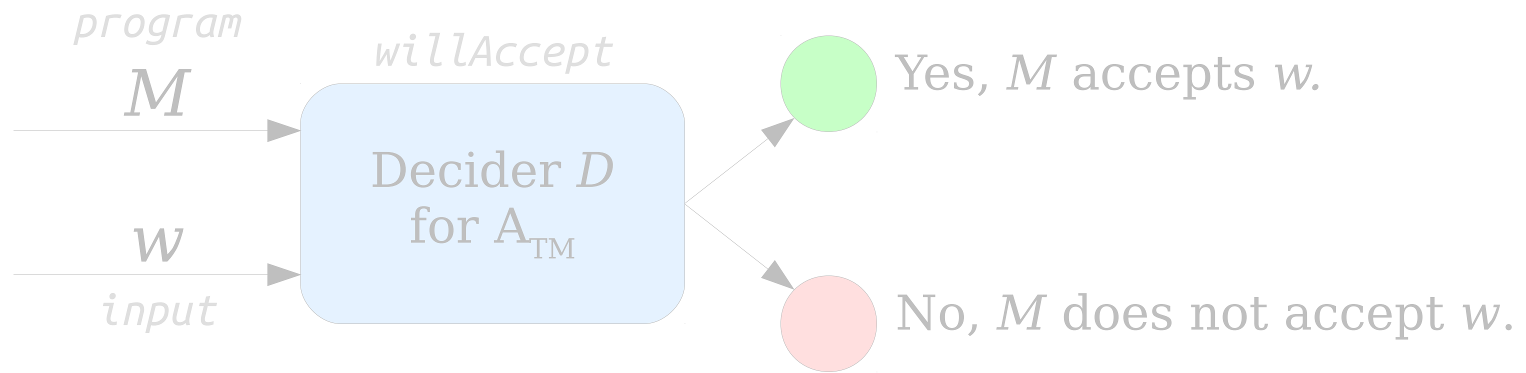
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!



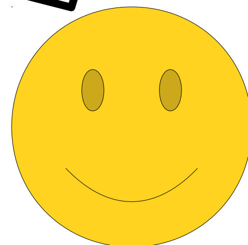

```
bool willAccept(string program, string input)
```

Program P design specification:

- ✓ If P accepts its input, then P does not accept its input.
- ✓ If P does not accept its input, then P accepts its input.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```

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



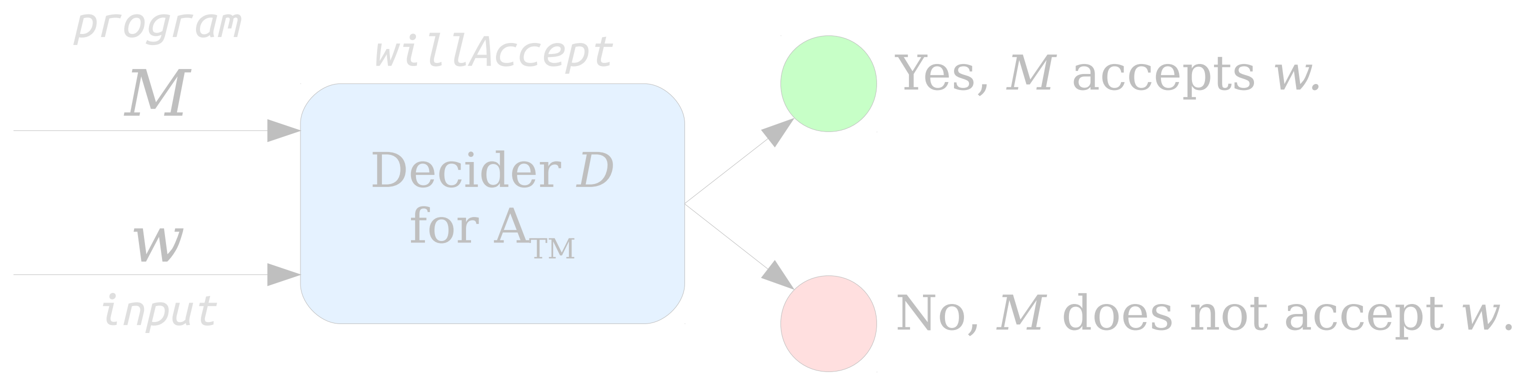
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

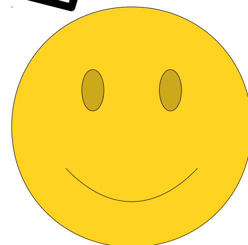
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```

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



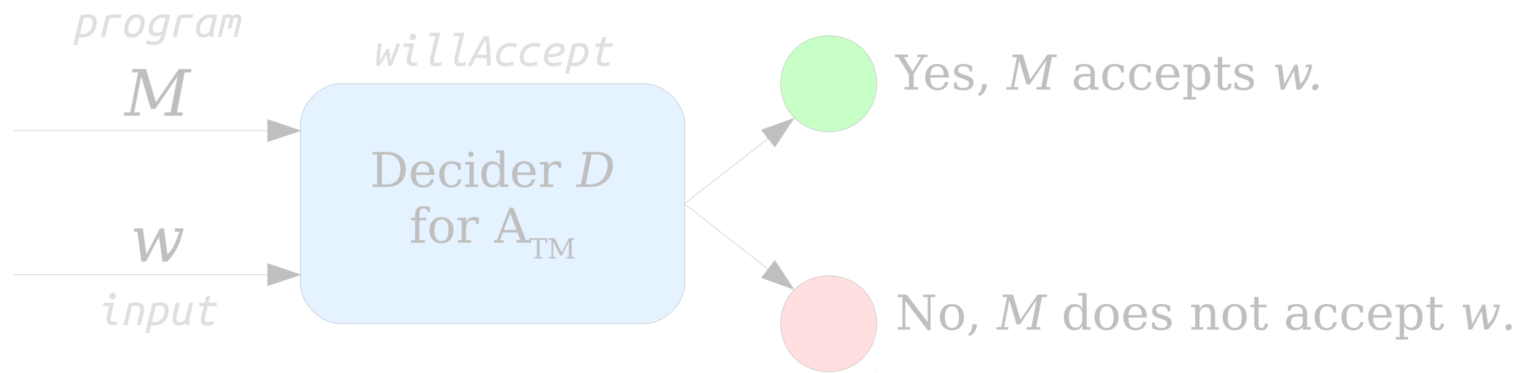
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

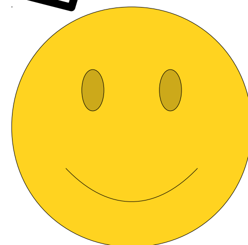
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    accept();
  } else {
    reject();
  }
}
```

… and swap them like this.



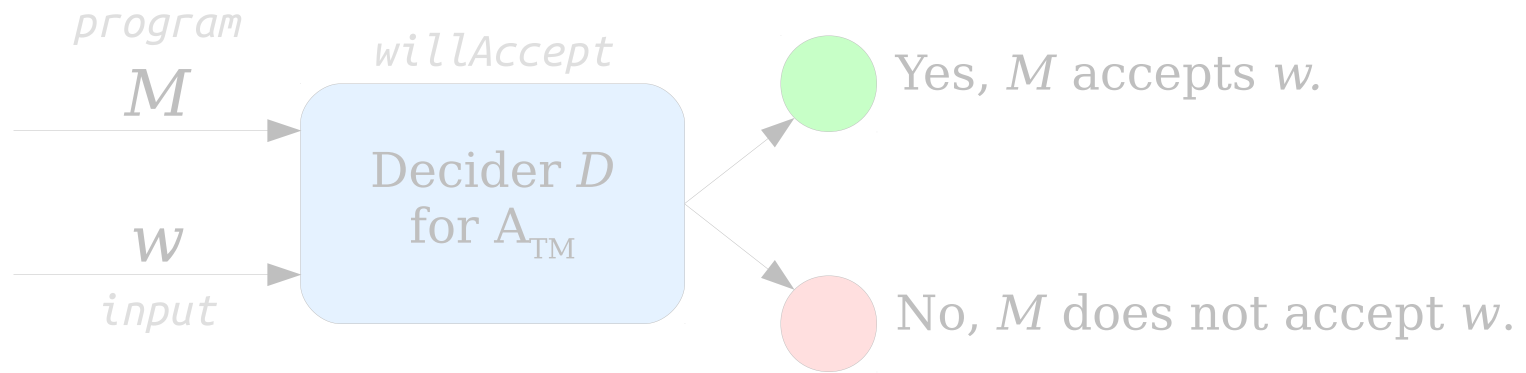
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

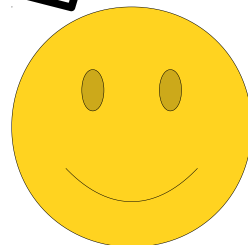
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    accept();
  } else {
    reject();
  }
}
```

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



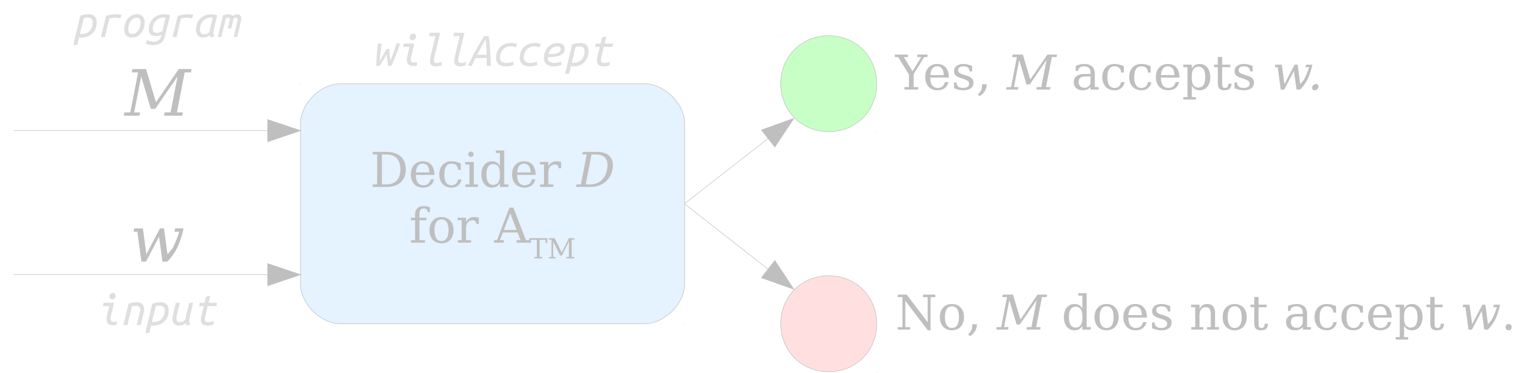
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

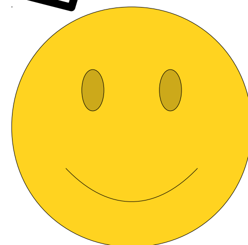
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    accept();
  } else {
    reject();
  }
}
```

Turns out, that doesn't work. Let's see why.



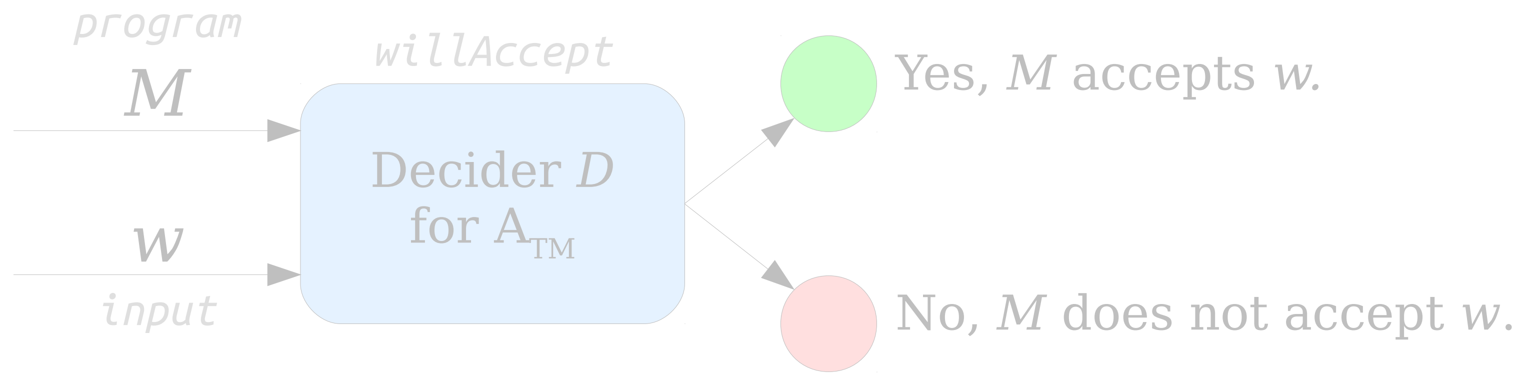
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

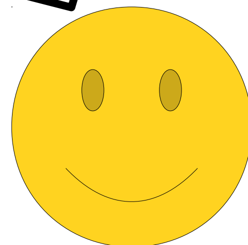
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    accept();
  } else {
    reject();
  }
}
```

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



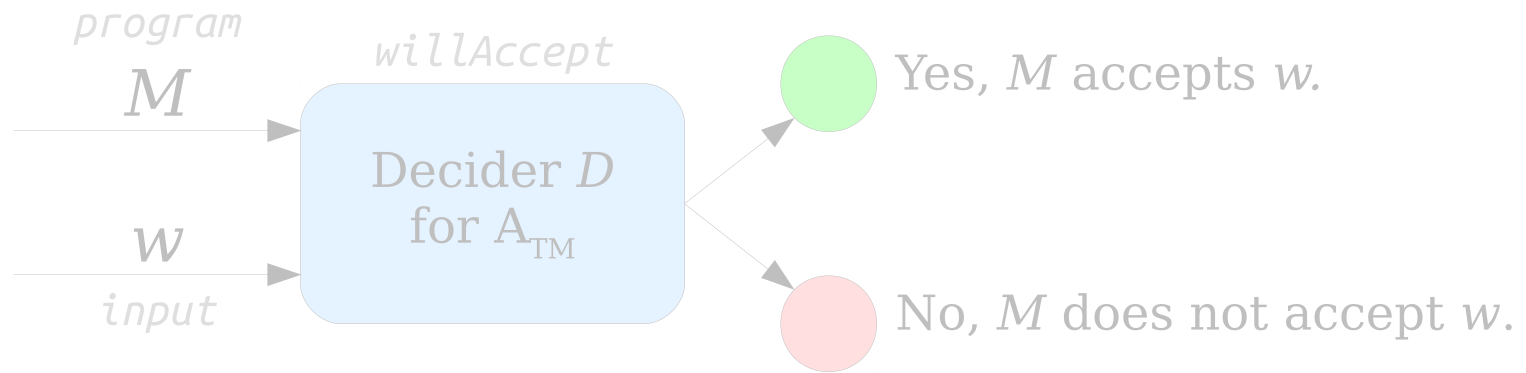
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does accept its input

Contradiction!




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

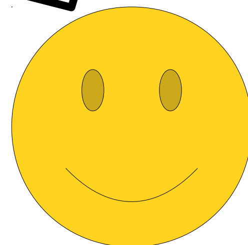
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    accept();
  } else {
    reject();
  }
}
```

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



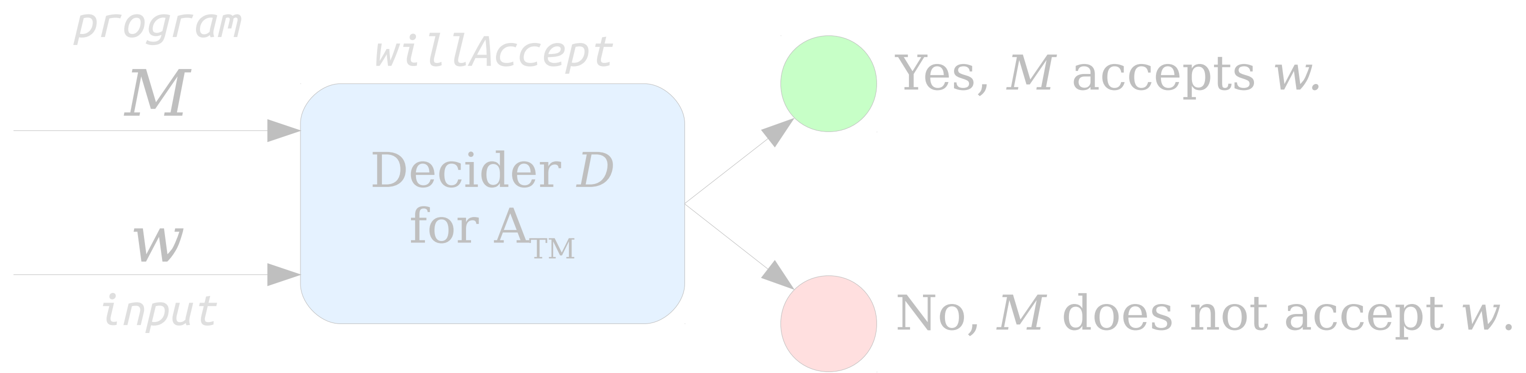
$A_{TM} \in \mathbf{R}$

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

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

Program *P* accepts its input if and only if program *P* does accept its input

Contradiction!




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

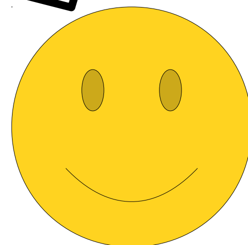
Program *P* design specification:

✓ If *P* accepts its input, then *P* does not accept its input.

✓ If *P* does not accept its input, then *P* accepts its input.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    accept();
  } else {
    reject();
  }
}
```

Notice that this is a true statement.



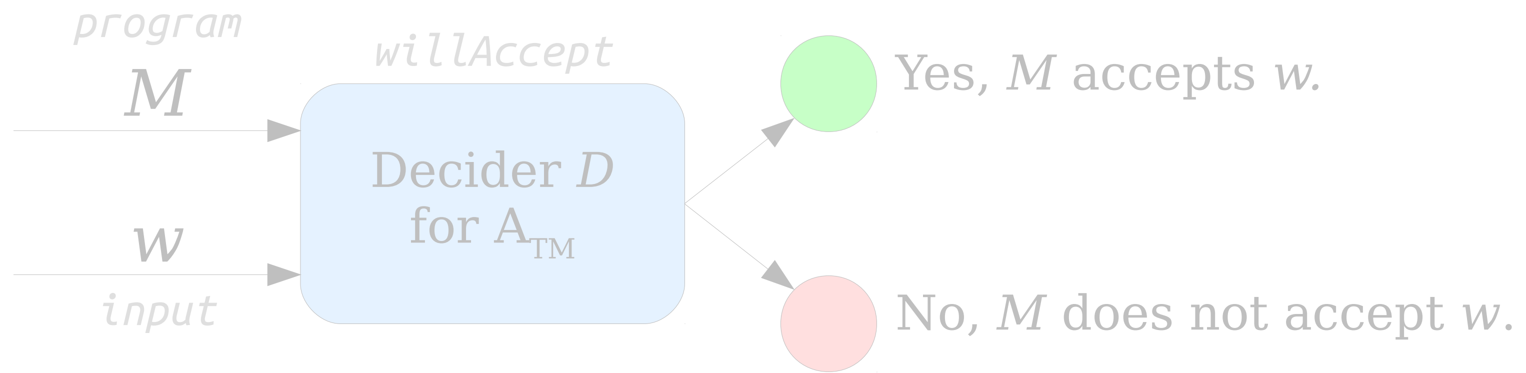
$A_{TM} \in \mathbf{R}$

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

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

Program *P* accepts its input if and only if program *P* does not accept its input

Contradiction!




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

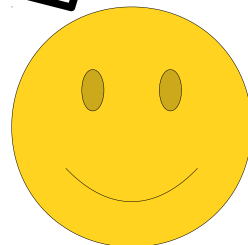
Program *P* design specification:

✓ If *P* accepts its input, then *P* does not accept its input.

✓ If *P* does not accept its input, then *P* accepts its input.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    accept();
  } else {
    reject();
  }
}
```

Originally, we got a contradiction here.



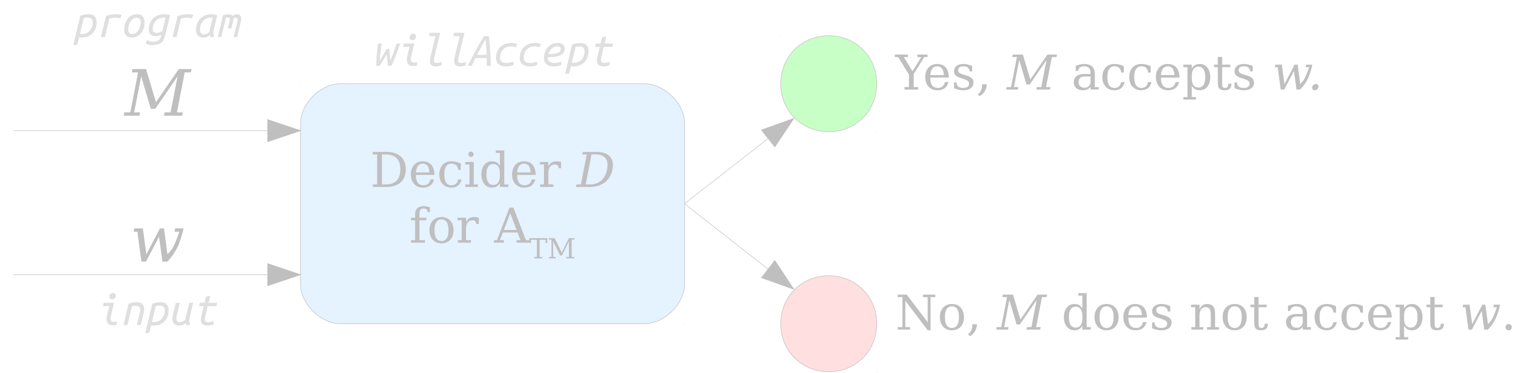
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

$\top$




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

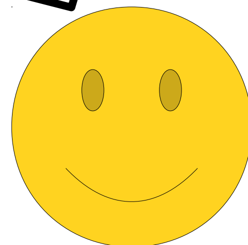
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    accept();
  } else {
    reject();
  }
}
```

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



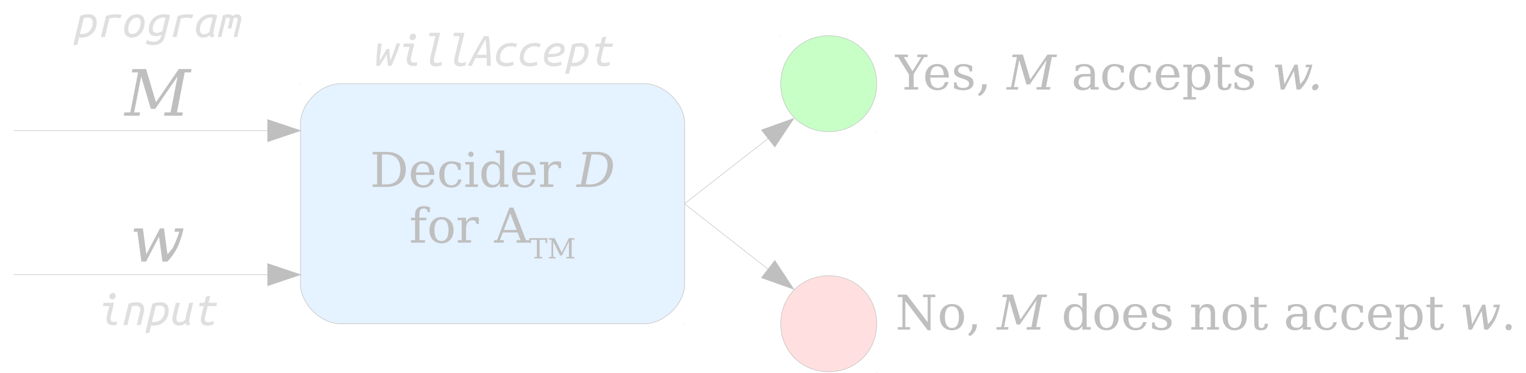
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does accept its input

$\top$




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

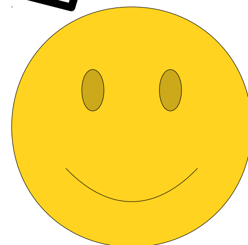
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    accept();
  } else {
    reject();
  }
}
```

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



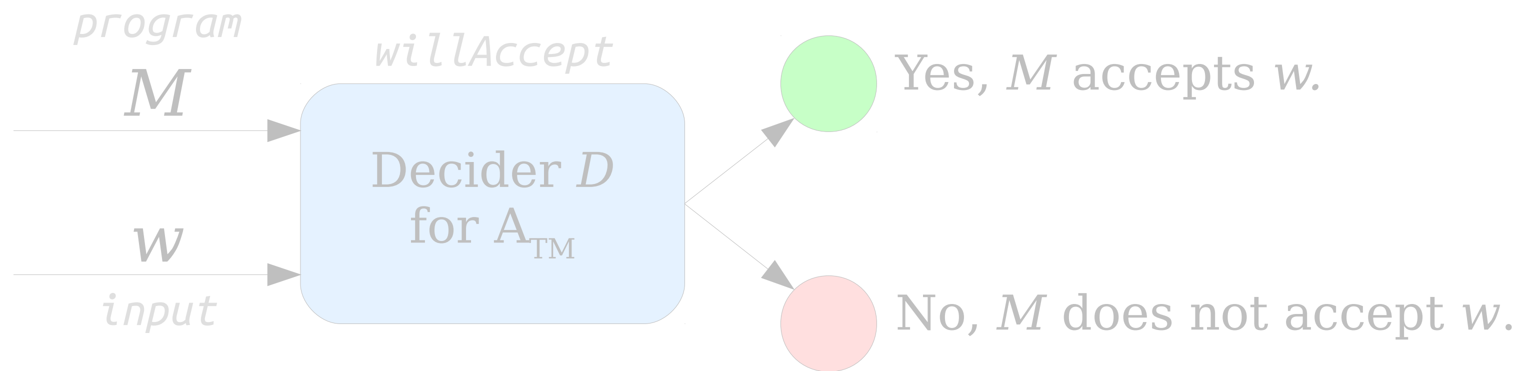
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

$\top$




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

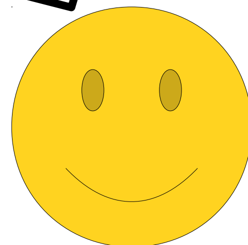
Program P design specification:

- ✓ If P accepts its input, then P does not accept its input.
- ✓ If P does not accept its input, then P accepts its input.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    accept();
  } else {
    reject();
  }
}
```

Overall, this shows that

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



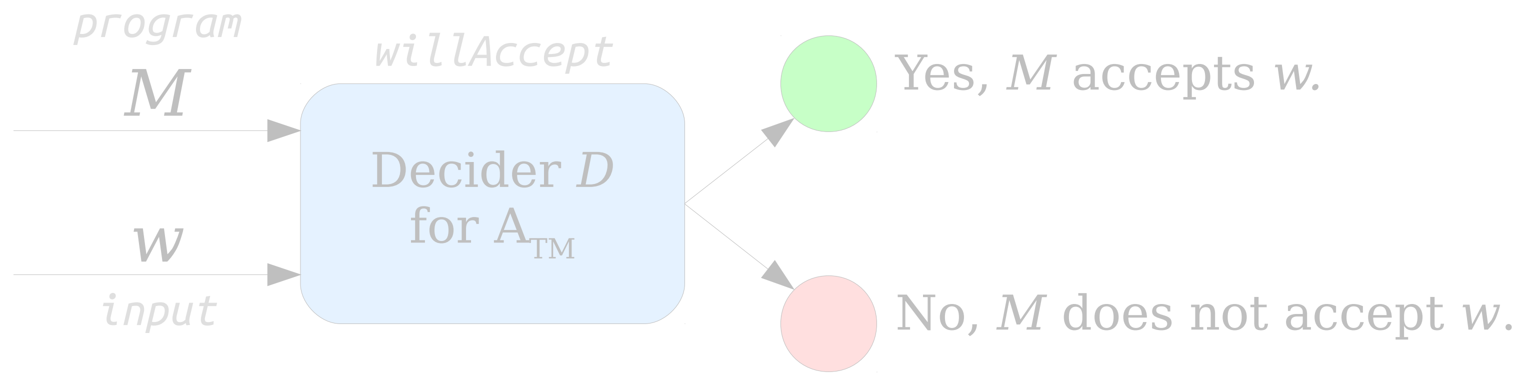
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

$\top$




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

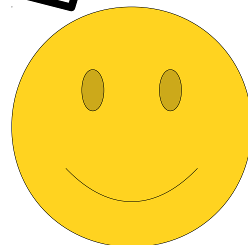
Program P design specification:

- ✓ If P accepts its input, then P does not accept its input.
- ✓ If P does not accept its input, then P accepts its input.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    accept();
  } else {
    reject();
  }
}
```

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

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



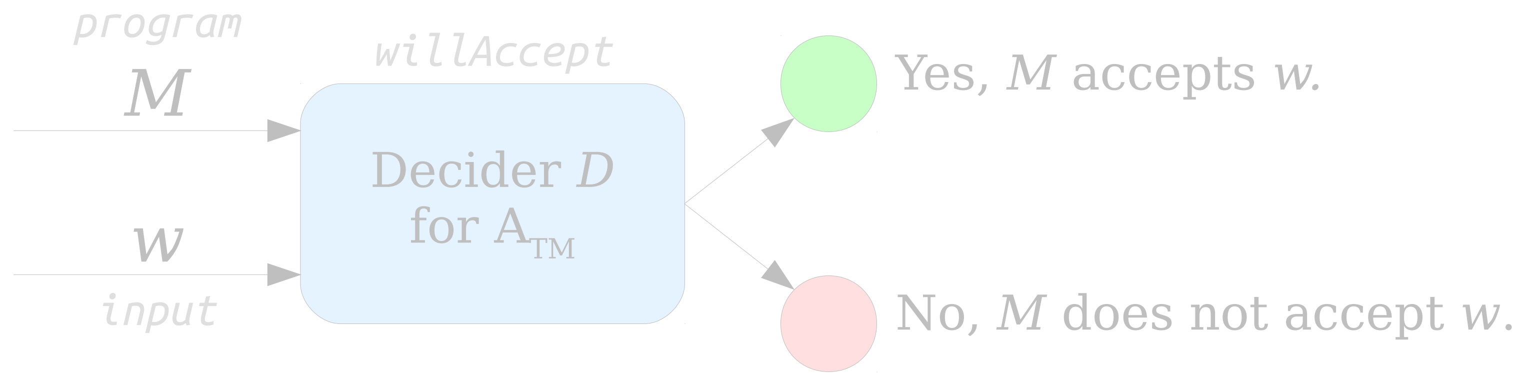
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

⊤




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

Program P design specification:

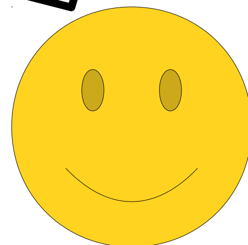
✓ If P accepts its input, then P does not accept its input.

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    accept();
  } else {
    reject();
  }
}
```

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

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



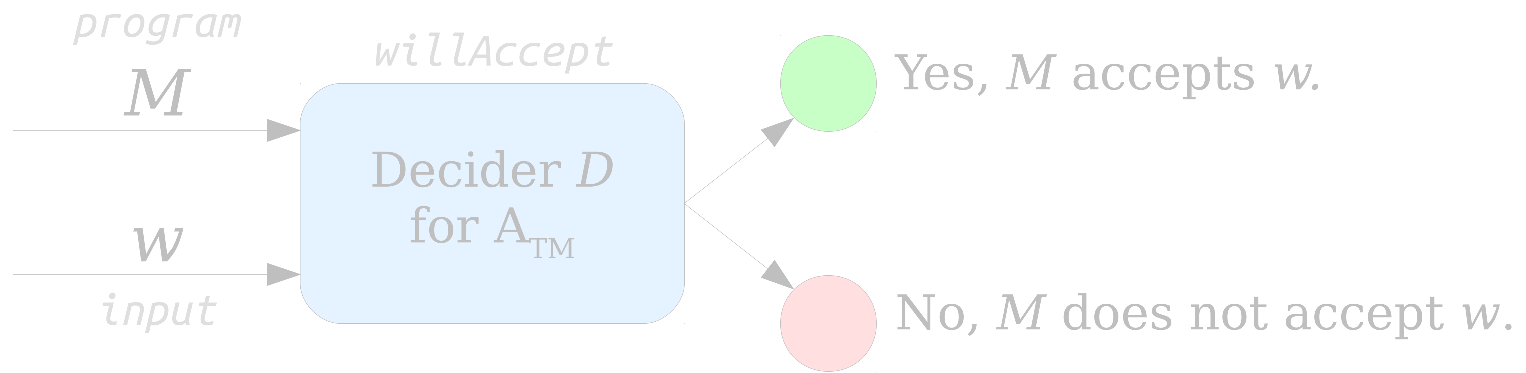
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

⊤




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

Program P design specification:

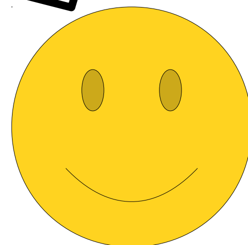
✓ If P accepts its input, then P does not accept its input.

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    accept();
  } else {
    reject();
  }
}
```

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

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



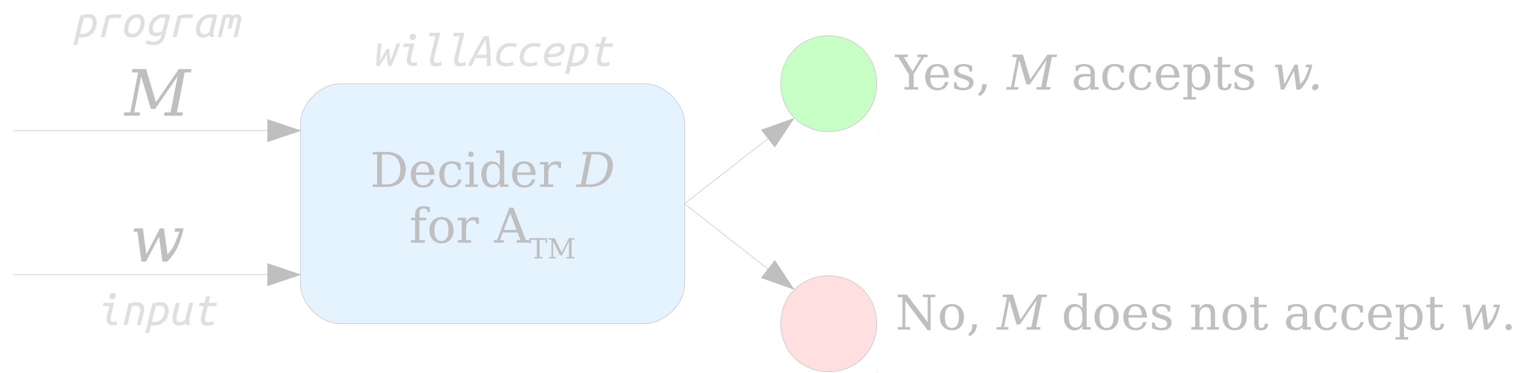
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

$\top$




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

Program P design specification:

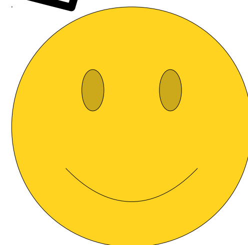
✓ If P accepts its input, then
P does not accept its input.

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    accept();
  } else {
    reject();
  }
}
```

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

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



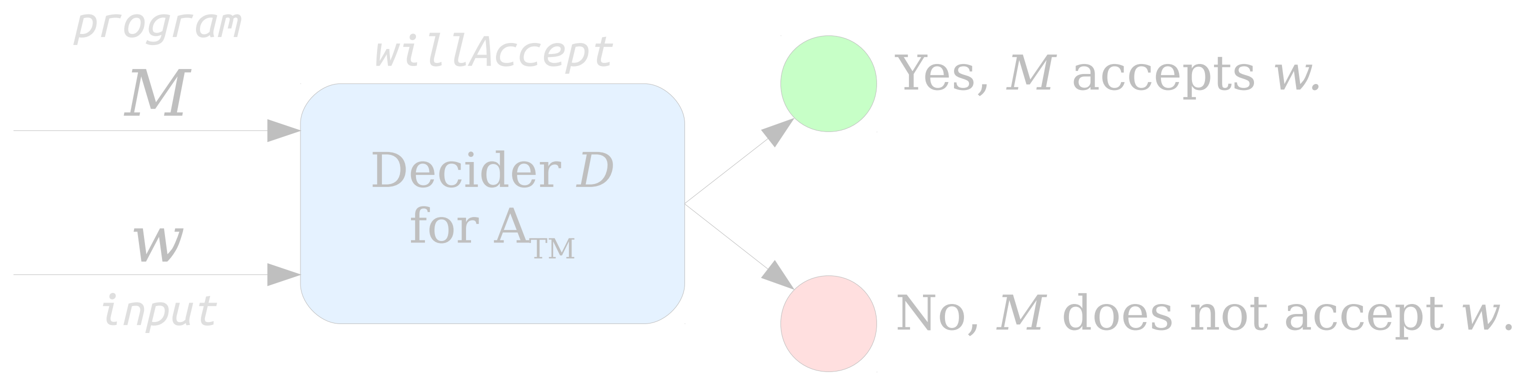
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

$\bot$




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

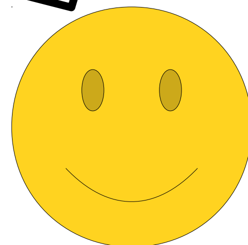
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    accept();
  } else {
    reject();
  }
}
```

Just so we don't get confused, let's reset everything back to how it used to be.



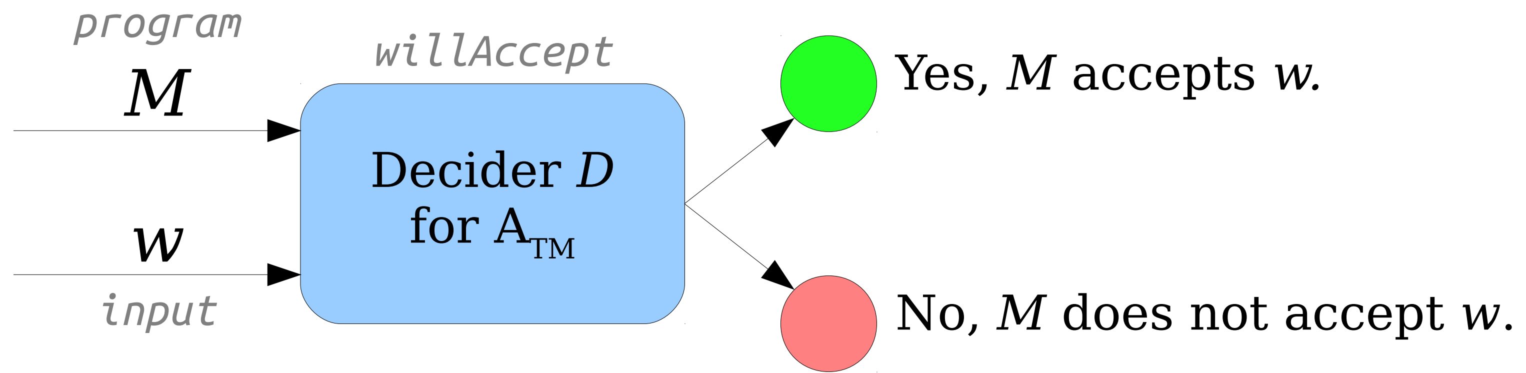
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!




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

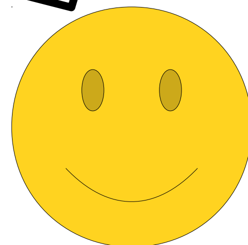
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```

Much better!



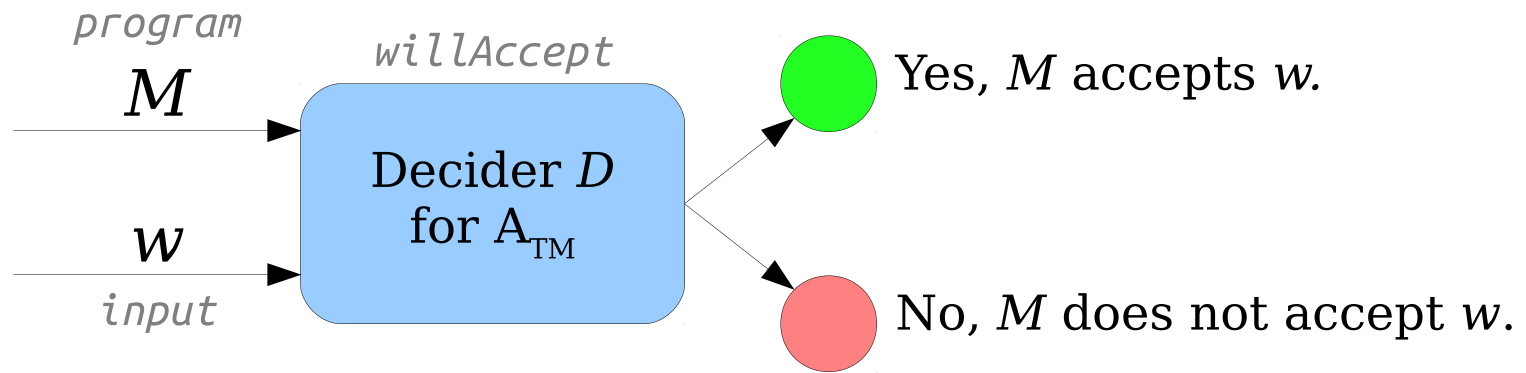
$A_{TM} \in \mathbf{R}$

There is a decider D for A_{TM}

We can write programs that use D as a helper method

Program P accepts its input if and only if program P does not accept its input

Contradiction!



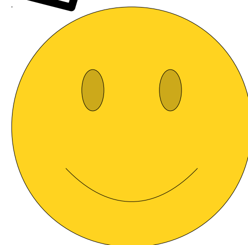

```
bool willAccept(string program, string input)
```

Program P design specification:

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

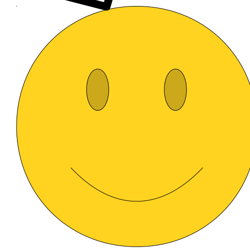
✓ If P does not accept its input, then
  P accepts its input.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (willAccept(me, input)) {
    reject();
  } else {
    accept();
  }
}
```



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?

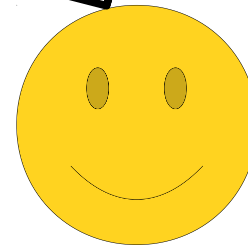


M is a secure voting machine

if and only if

$\mathscr{L}(M) = \{\ w \in \{\texttt{r}, \texttt{d}\}^* \mid w \text{ has more } \texttt{r}\text{'s than } \texttt{d}\text{'s}\ \}$

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

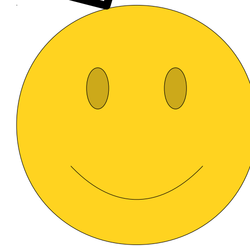


M is a secure voting machine

if and only if

$$\mathscr{L}(M) = \{\ w \in \{\texttt{r}, \texttt{d}\}^* \mid w \text{ has more } \texttt{r}\text{'s than } \texttt{d}\text{'s}\ \}$$

That's kind of a lot to take in at once.

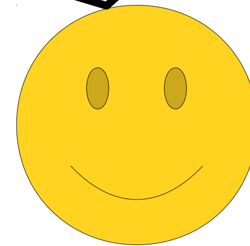


M is a secure voting machine

if and only if

$\mathscr{L}(M) = \{ w \in \{\texttt{r}, \texttt{d}\}^* \mid w$ has more $\texttt{r}$'s than $\texttt{d}$'s $\}$

Remember – the language of a TM is the set of all the strings it accepts.

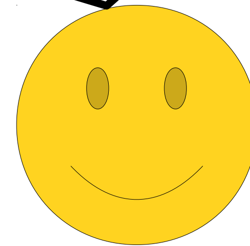


M is a secure voting machine

if and only if

$\mathscr{L}(M) = \{ w \in \{\texttt{r}, \texttt{d}\}^* \mid w \text{ has more } \texttt{r}\text{'s than } \texttt{d}\text{'s} \}$

So really this statement means that M accepts every string with more r's than d's and nothing else.

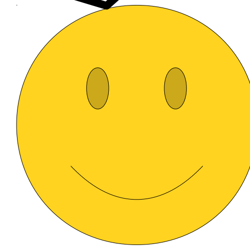


M is a secure voting machine

if and only if

$\mathscr{L}(M) = \{ w \in \{\texttt{r}, \texttt{d}\}^* \mid w$ has more $\texttt{r}$'s than $\texttt{d}$'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.

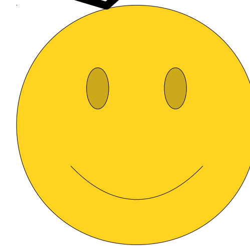


M is a secure voting machine

if and only if

$\mathscr{L}(M) = \{ w \in \{\texttt{r}, \texttt{d}\}^* \mid w \text{ has more } \texttt{r}\text{'s than } \texttt{d}\text{'s} \}$

Notice that our goal was ***not*** to show that you can't build a secure voting machine.

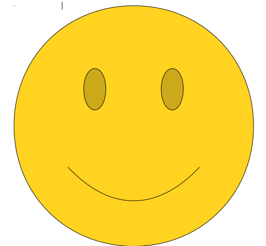


M is a secure voting machine

if and only if

$\mathscr{L}(M) = \{ w \in \{\texttt{r}, \texttt{d}\}^* \mid w \text{ has more } \texttt{r}\text{'s than } \texttt{d}\text{'s} \}$

It's absolutely possible to do that.



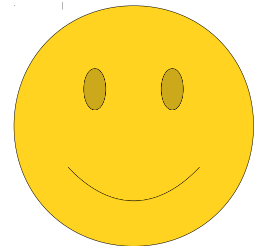
```
int main() {
  string input = getInput();
  if (countRs(input) > countDs(input)) {
    accept();
  } else {
    reject();
  }
}
```

M is a secure voting machine

if and only if

$\mathscr{L}(M) = \{ w \in \{\texttt{r}, \texttt{d}\}^* \mid w \text{ has more } \texttt{r}\text{'s than } \texttt{d}\text{'s} \}$

The hard part is being able to tell whether an arbitrary program is a secure voting machine.



```
int main() {
  string input = getInput();
  if (countRs(input) > countDs(input)) {
    accept();
  } else {
    reject();
  }
}
```

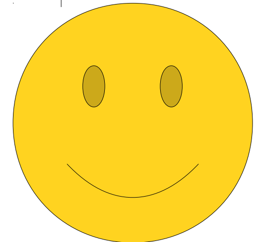
M is a secure voting machine

if and only if

$\mathscr{L}(M) = \{\ w \in \{r, d\}^* \mid w$ has more r's than d's $\}$

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

```
int main() {
  string input = getInput();

  int n = countRs(input);
  while (n > 1) {
    if (n % 2 == 0) n = n / 2;
    else n = 3*n + 1;
  }

  if (countRs(input) > countDs(input)) {
    accept();
  } else {
    reject();
  }
}
```

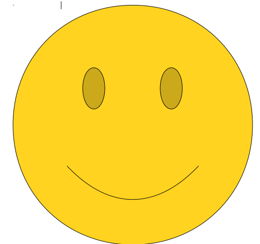


M is a secure voting machine

if and only if

$\mathscr{L}(M) = \{\ w \in \{r, d\}^* \mid w$ has more r's than d's $\}$

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



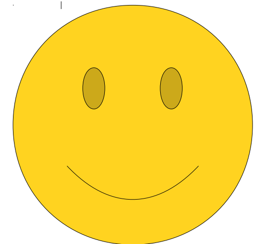
```
int main() {
  string input = getInput();

  int n = countRs(input);
  while (n > 1) {
    if (n % 2 == 0) n = n / 2;
    else n = 3*n + 1;
  }

  if (countRs(input) > countDs(input)) {
    accept();
  } else {
    reject();
  }
}
```

M is a secure voting machine

if and only if

$\mathscr{L}(M) = \{ w \in \{\texttt{r}, \texttt{d}\}^* \mid w$ has more $\texttt{r}$'s than $\texttt{d}$'s $\}$

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



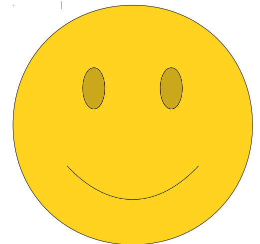
```
int main() {
  string input = getInput();

  int n = countRs(input);
  while (n > 1) {
    if (n % 2 == 0) n = n / 2;
    else n = 3*n + 1;
  }

  if (countRs(input) > countDs(input)) {
    accept();
  } else {
    reject();
  }
}
```

M is a secure voting machine

if and only if

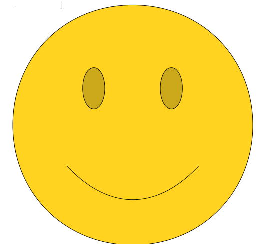
$\mathscr{L}(M) = \{ w \in \{$`r`, `d`$\}^* \mid w$ has more `r`'s than `d`'s $\}$

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



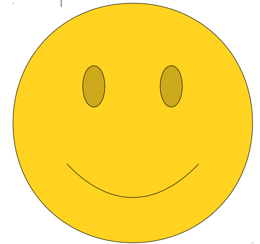
```
int main() {
  string input = getInput();

  int n = countRs(input);
  while (n > 1) {
    if (n % 2 == 0) n = n / 2;
    else n = 3*n + 1;
  }

  if (countRs(input) > countDs(input)) {
    accept();
  } else {
    reject();
  }
}
```

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.

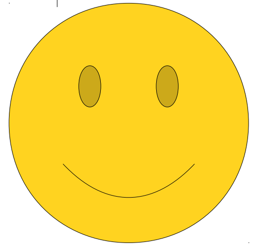
Following our pattern from before, we'll assume that the secure voting problem is decidable.



The secure voting problem is decidable.

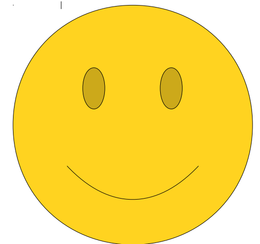
Contradiction!

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.

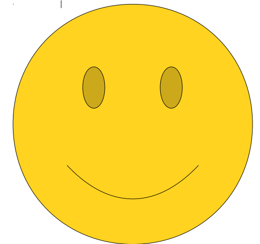


Contradiction!

The secure voting problem is decidable.

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

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



Contradiction!

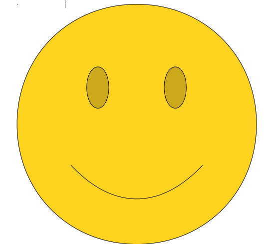
The secure voting problem is decidable.

There is a decider D for the secure voting problem

Contradiction!

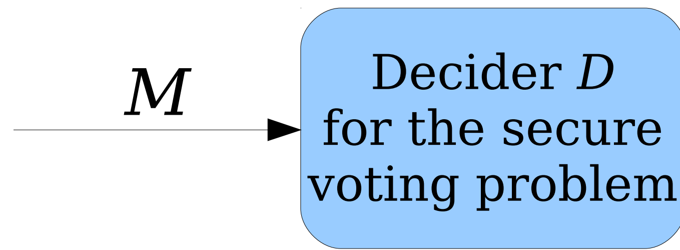
Decider D for the secure voting problem

So what does that look like?

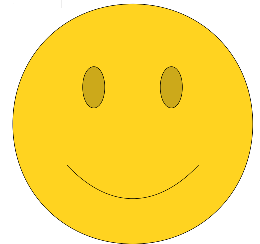


The secure voting problem is decidable.

There is a decider D for the secure voting problem



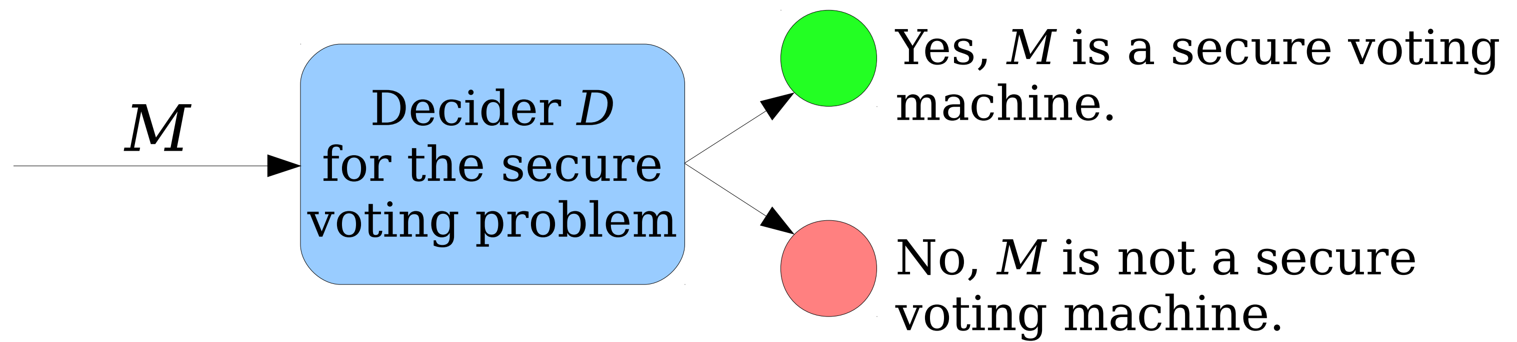
A decider for the secure voting problem will take in some TM M, which is the machine we want to specifically check.



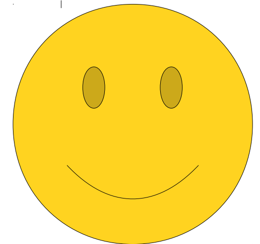
Contradiction!

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.



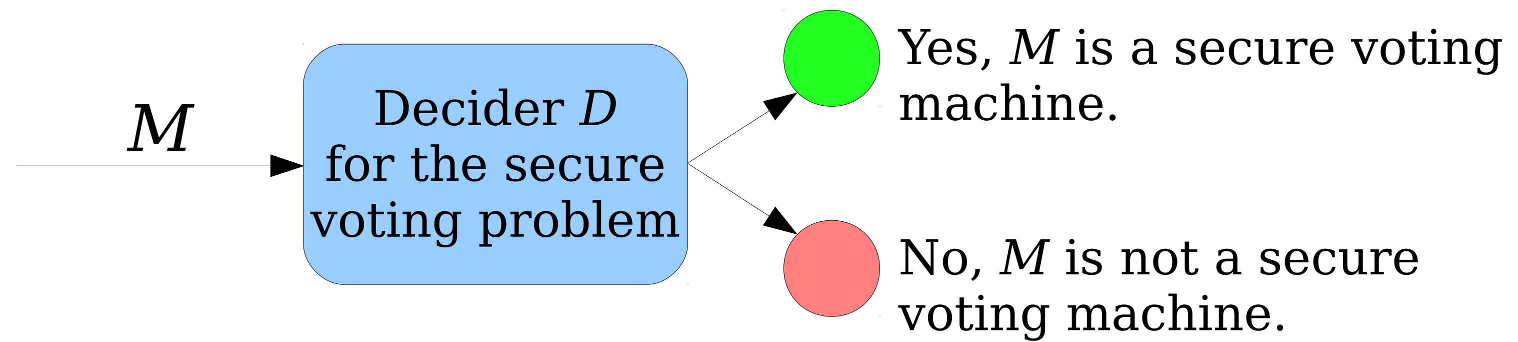
Contradiction!

The secure voting problem is decidable.

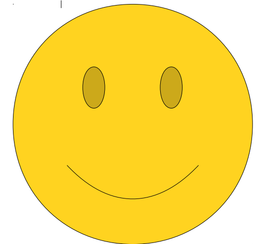
There is a decider D for the secure voting problem

We can write programs that use D as a helper method

Contradiction!



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

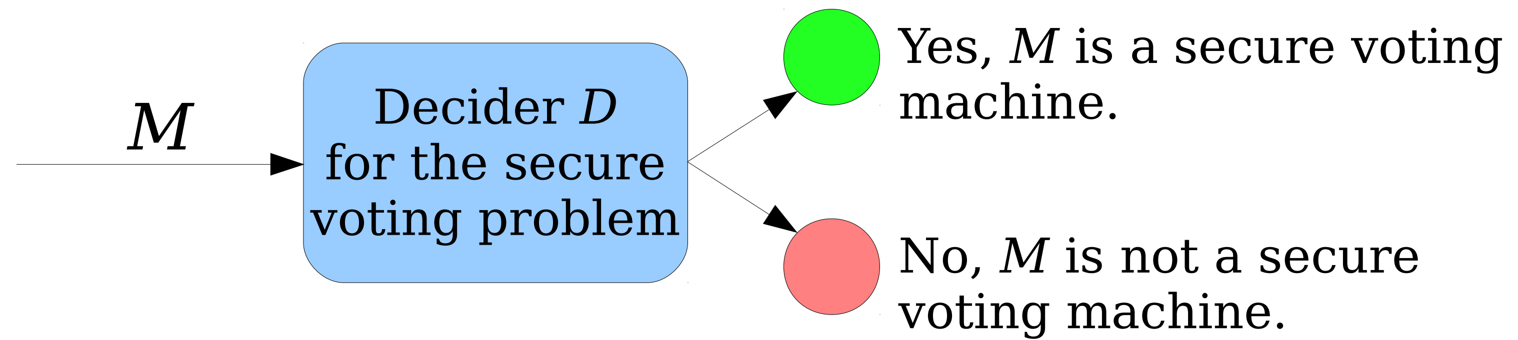


The secure voting problem is decidable.

There is a decider D for the secure voting problem

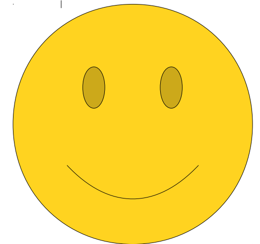
We can write programs that use D as a helper method

Contradiction!




```
bool isSecure(string program)
```

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

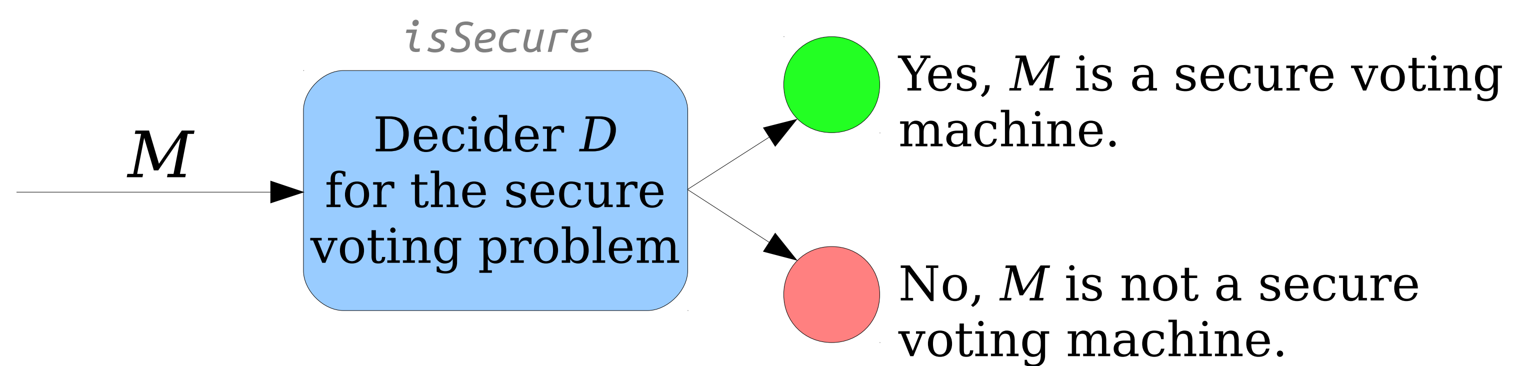


The secure voting problem is decidable.

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

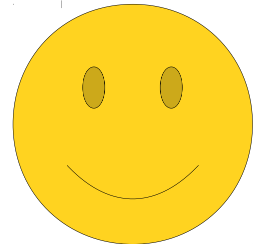
We can write programs that use *D* as a helper method

Contradiction!




```
bool isSecure(string program)
```

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

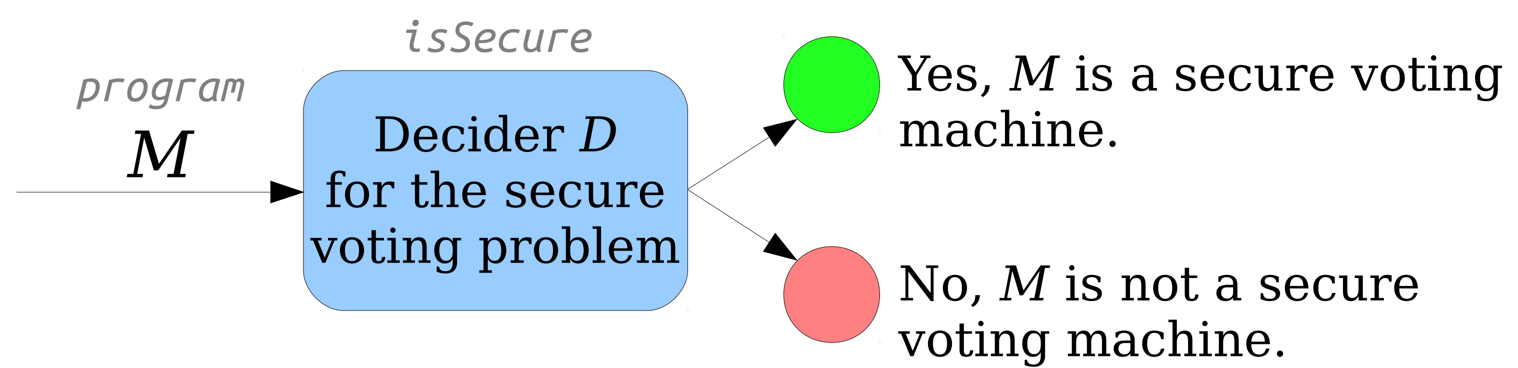


The secure voting problem is decidable.

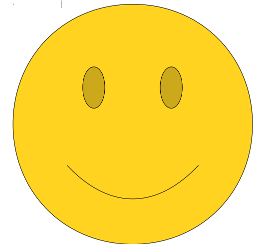
There is a decider D for the secure voting problem

We can write programs that use D as a helper method

Contradiction!




```
bool isSecure(string program)
```

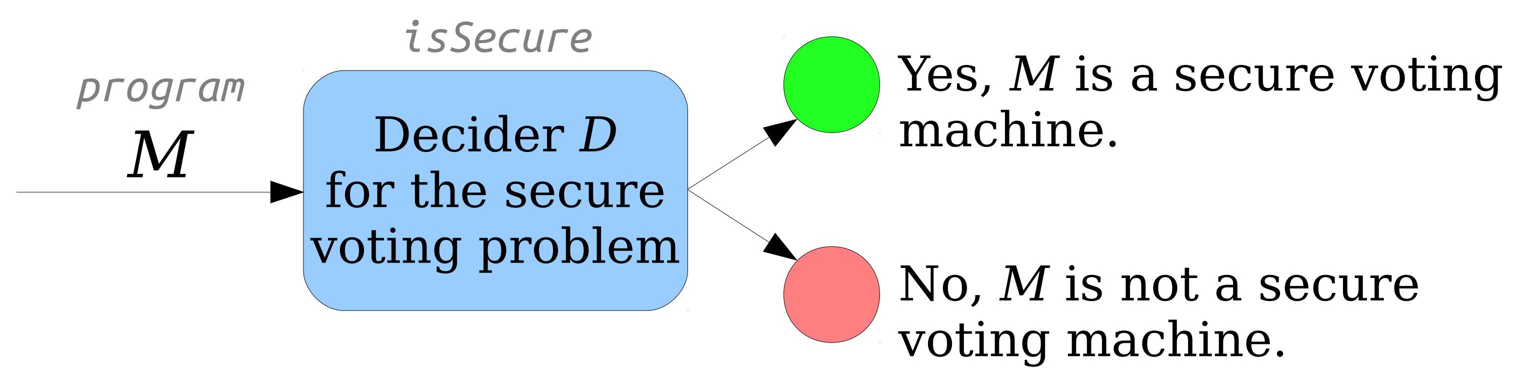


The secure voting problem is decidable.

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

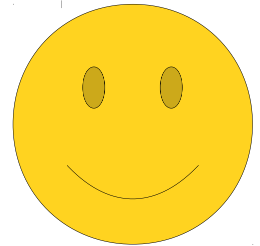
We can write programs that use *D* as a helper method

Contradiction!




```
bool isSecure(string program)
```

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

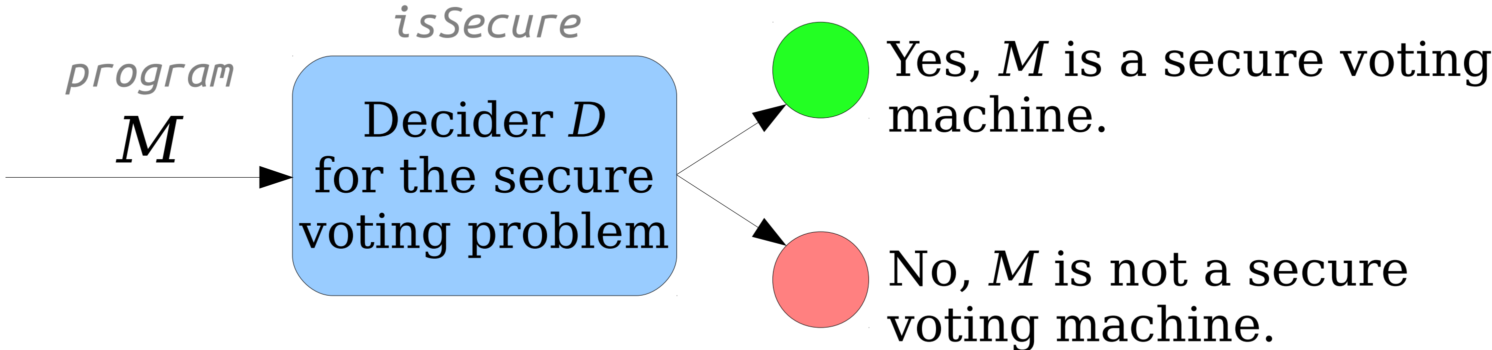


The secure voting problem is decidable.

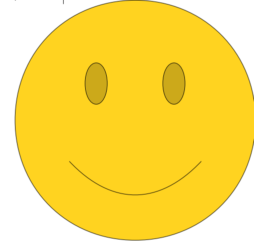
There is a decider D for the secure voting problem

We can write programs that use D as a helper method

Contradiction!




```
bool isSecure(string program)
```

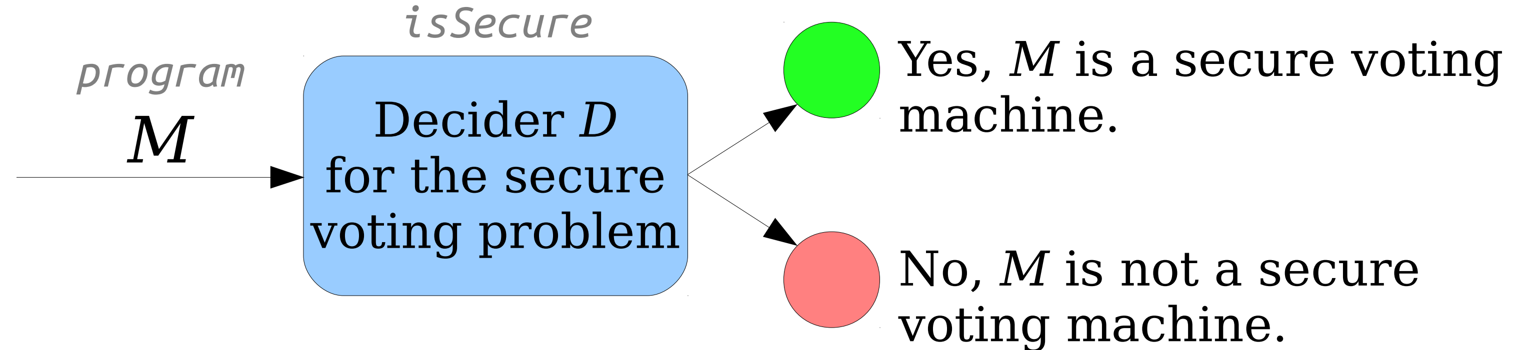


The secure voting problem is decidable.

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

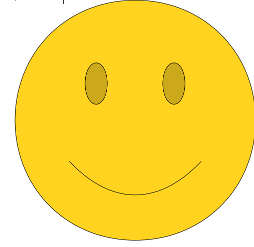
We can write programs that use *D* as a helper method

Contradiction!

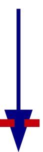



```
bool isSecure(string program)
```

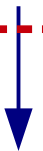
That was a great contradiction to get when we had a decider that would tell us whether a program would accept a given input.



The secure voting problem is decidable.

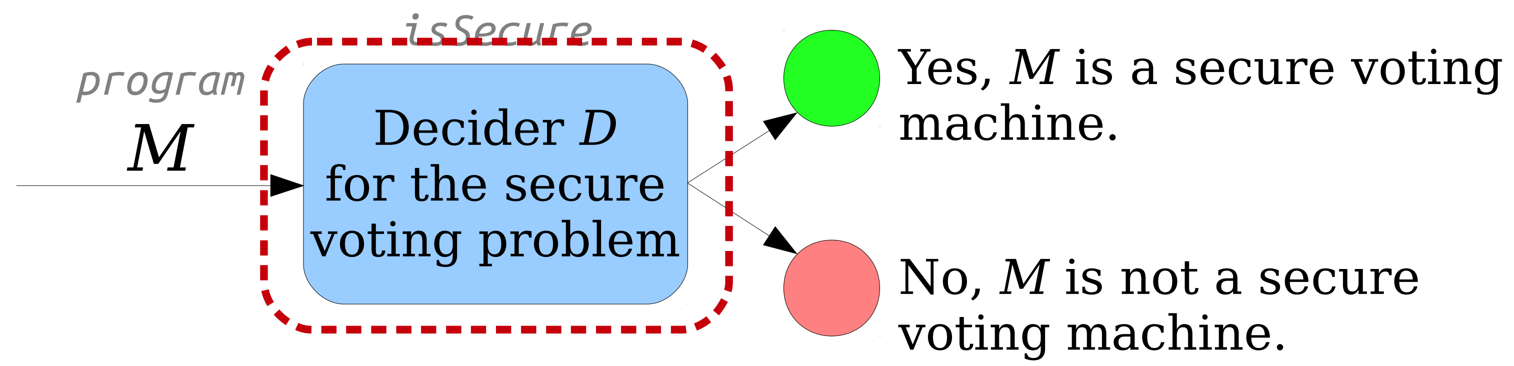


There is a decider D for the secure voting problem



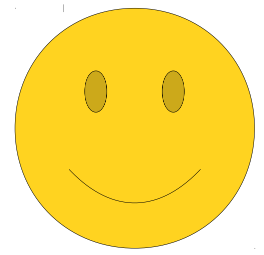
We can write programs that use D as a helper method

Contradiction!




```
bool isSecure(string program)
```

The problem here is that our decider doesn't do that. Instead, it tells us whether a program is a secure voting machine.

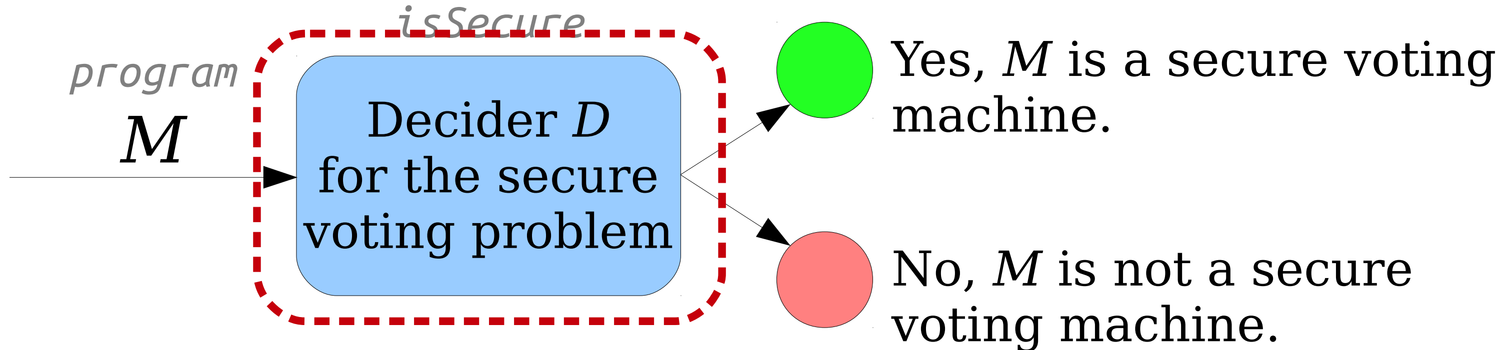


The secure voting problem is decidable.

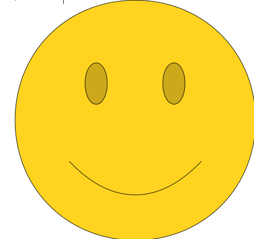
There is a decider D for the secure voting problem

We can write programs that use D as a helper method

Contradiction!




```
bool isSecure(string program)
```



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

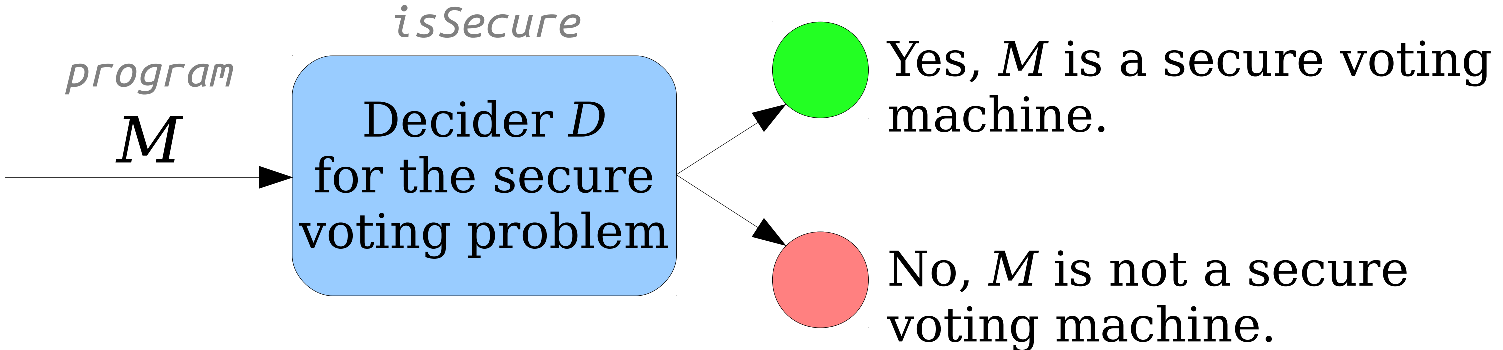
The secure voting problem is decidable.

There is a decider D for the secure voting problem

We can write programs that use D as a helper method

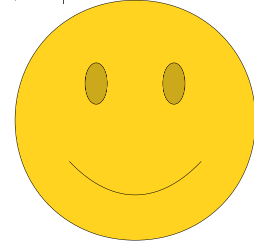
Program P is secure if and only if program P is not secure.

Contradiction!




```
bool isSecure(string program)
```

Specifically, we're going to build a program P that is a secure voting machine if and only if it's not a secure voting machine.



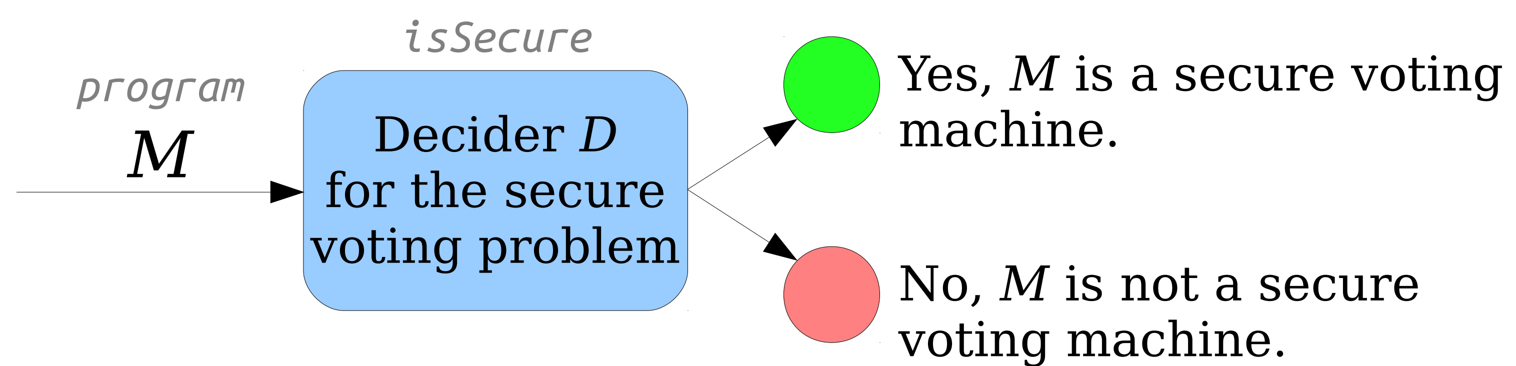
The secure voting problem is decidable.

There is a decider D for the secure voting problem

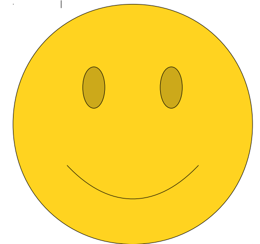
We can write programs that use D as a helper method

Program P is secure if and only if program P is not secure.

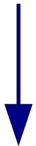
Contradiction!



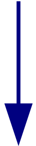

```
bool isSecure(string program)
```



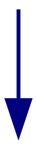
The secure voting problem is decidable.



There is a decider D for the secure voting problem

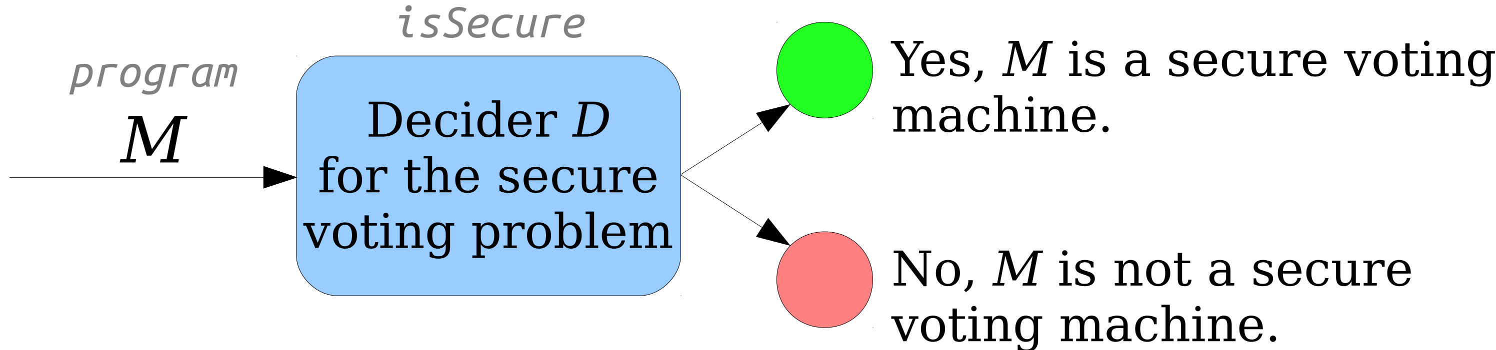


We can write programs that use D as a helper method

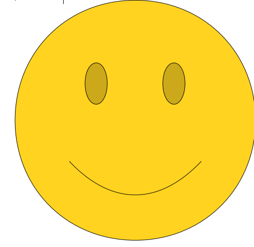


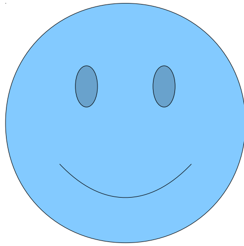
Program P is secure if and only if program P is not secure.

Contradiction!




```
bool isSecure(string program)
```



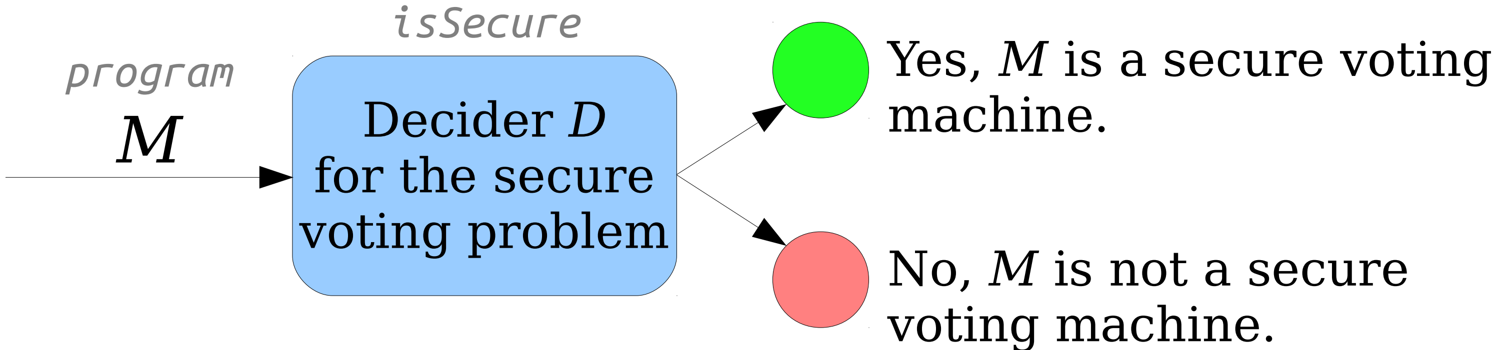
The secure voting problem is decidable.

There is a decider D for the secure voting problem

We can write programs that use D as a helper method

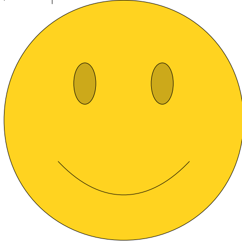
Program P is secure if and only if program P is not secure.

Contradiction!




```
bool isSecure(string program)
```

So now we have to figure out how to write this program P.



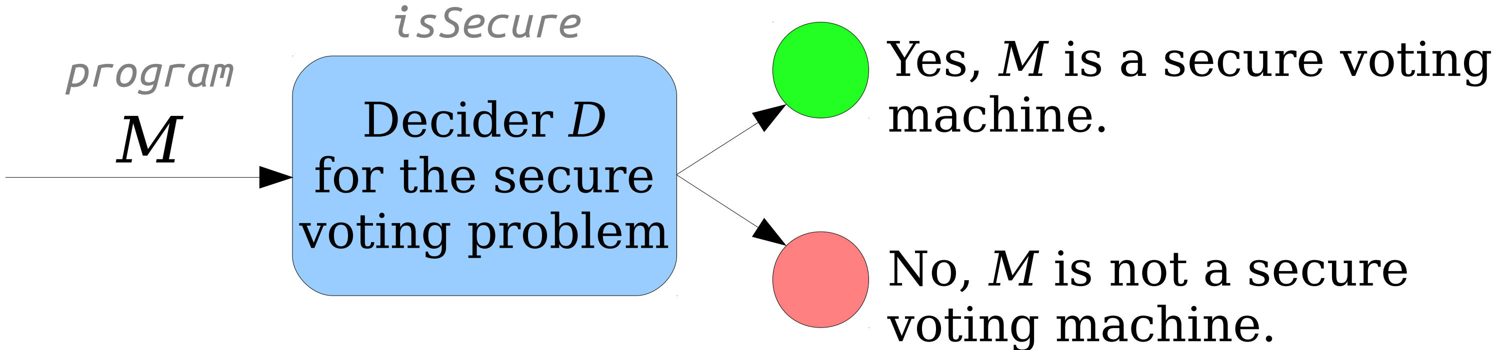
The secure voting problem is decidable.

There is a decider D for the secure voting problem

We can write programs that use D as a helper method

Program P is secure if and only if program P is not secure.

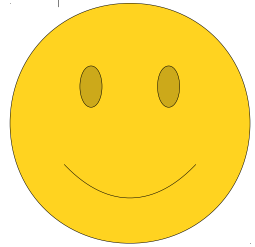
Contradiction!



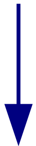

```
bool isSecure(string program)
```

Program P design specification:

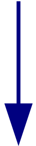
As before, let's start by writing out a design specification for what it's supposed to do.



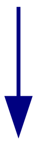
The secure voting problem is decidable.



There is a decider D for the secure voting problem

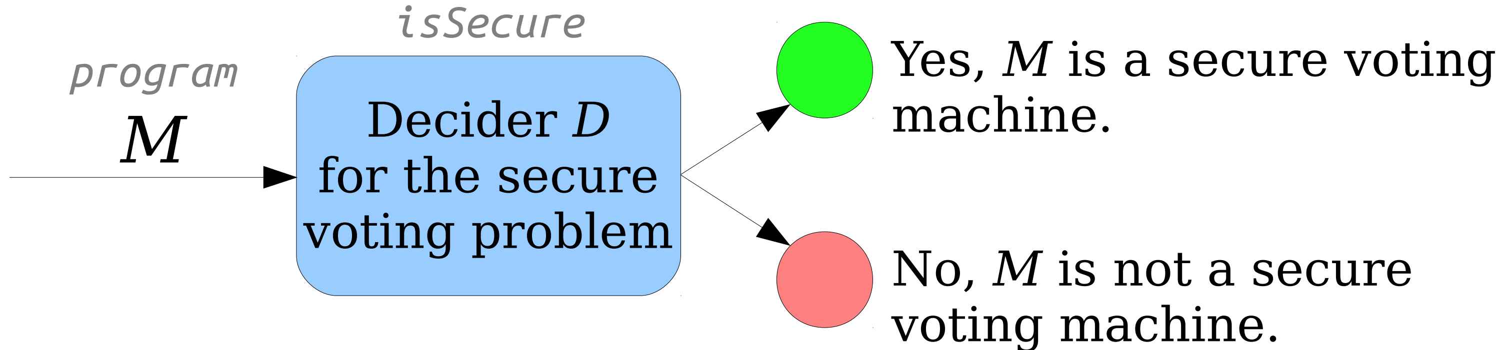


We can write programs that use D as a helper method



Program P is secure if and only if program P is not secure.

Contradiction!



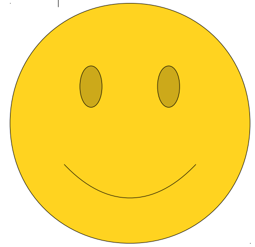

```
bool isSecure(string program)
```

Program P design specification:

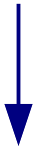
If P is a secure voting machine, then

P is not a secure voting machine.

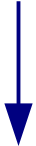
This first part takes care of the first half of the biconditional.



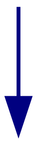
The secure voting problem is decidable.



There is a decider D for the secure voting problem

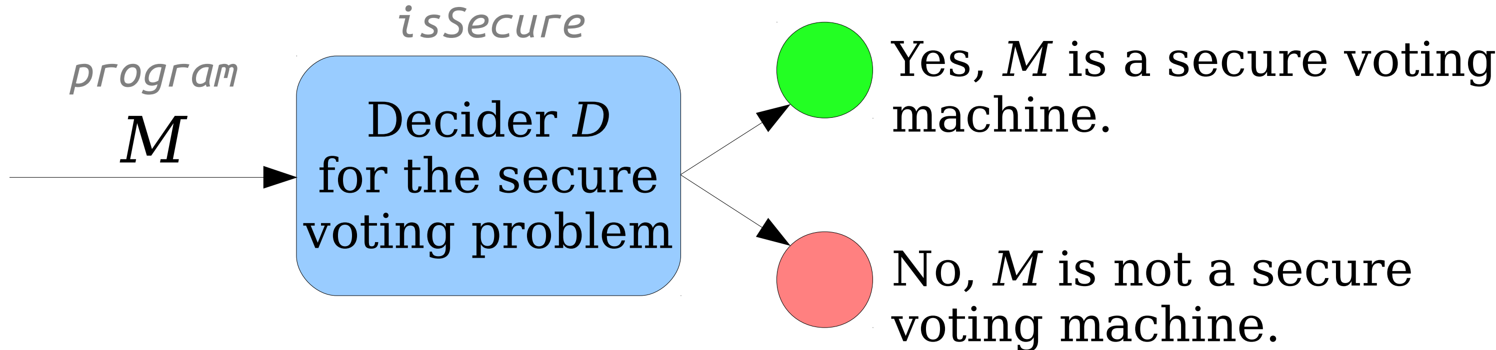


We can write programs that use D as a helper method



Program P is secure if and only if program P is not secure.

Contradiction!

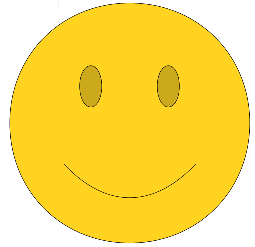



```
bool isSecure(string program)
```

Program P design specification:

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

This second part takes care of the other direction.



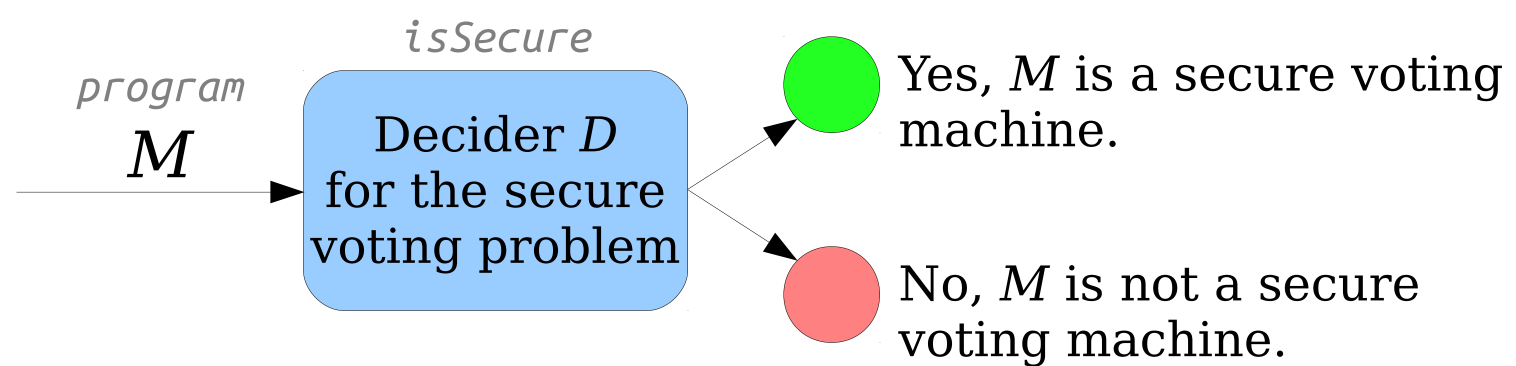
The secure voting problem is decidable.

There is a decider D for the secure voting problem

We can write programs that use D as a helper method

Program P is secure if and only if program P is not secure.

Contradiction!

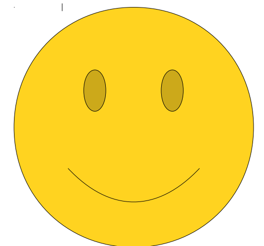



```
bool isSecure(string program)
```

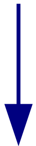
Program P design specification:

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

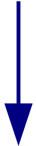
At this point, we have written out a spec for what we want P to do. All that's left to do now is to code it up!



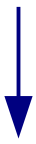
The secure voting problem is decidable.



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

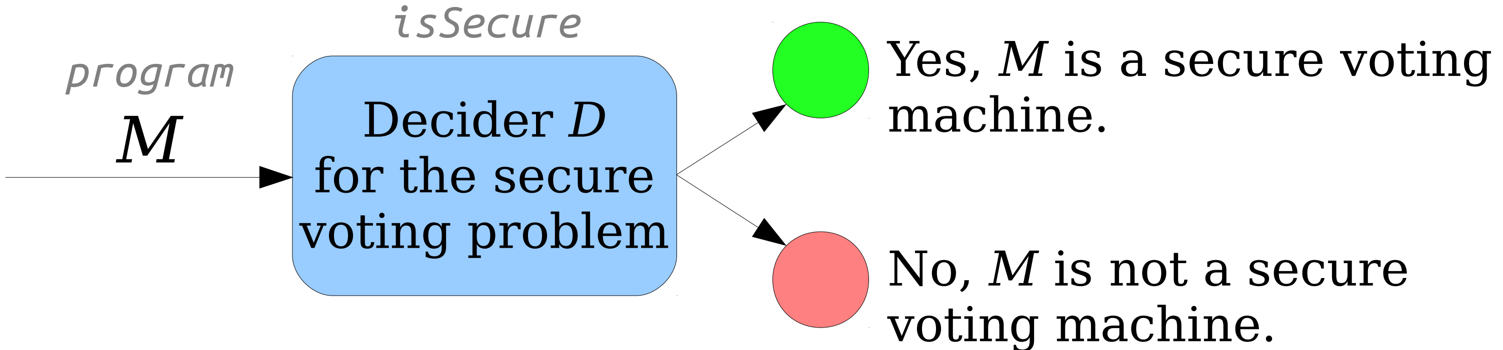


We can write programs that use *D* as a helper method



Program *P* is secure if and only if program *P* is not secure.

Contradiction!

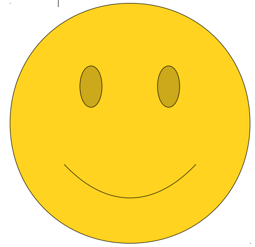



```
bool isSecure(string program)
```

Program *P* design specification:

If *P* is a secure voting machine, then
P is not a secure voting machine.
If *P* is not a secure voting machine, then
P 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.



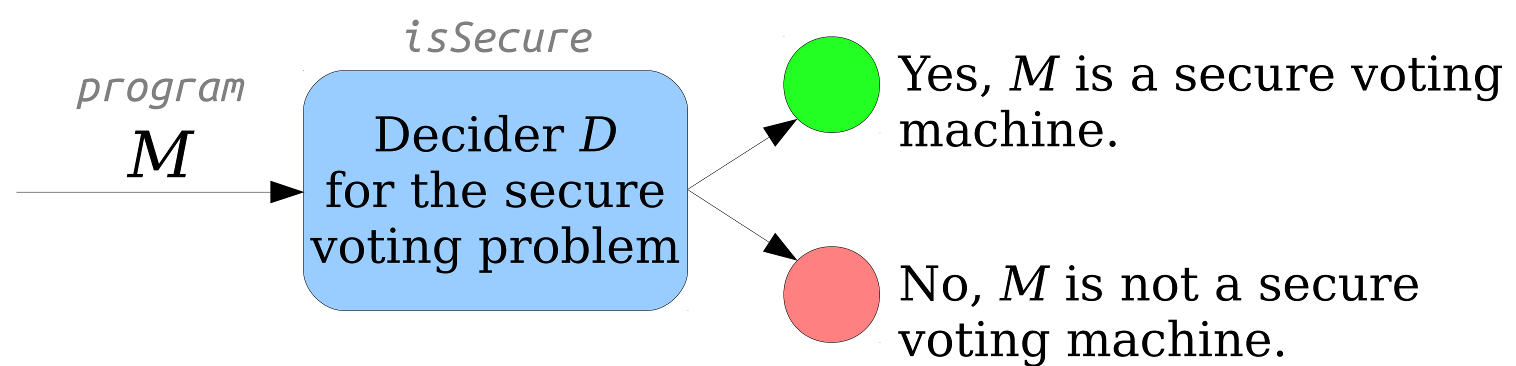
The secure voting problem is decidable.

There is a decider D for the secure voting problem

We can write programs that use D as a helper method

Program P is secure if and only if program P is not secure.

Contradiction!




```
bool isSecure(string program)
```

Program P design specification:

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

```
// Program P

int main() {

}
```

Our program starts off in main().

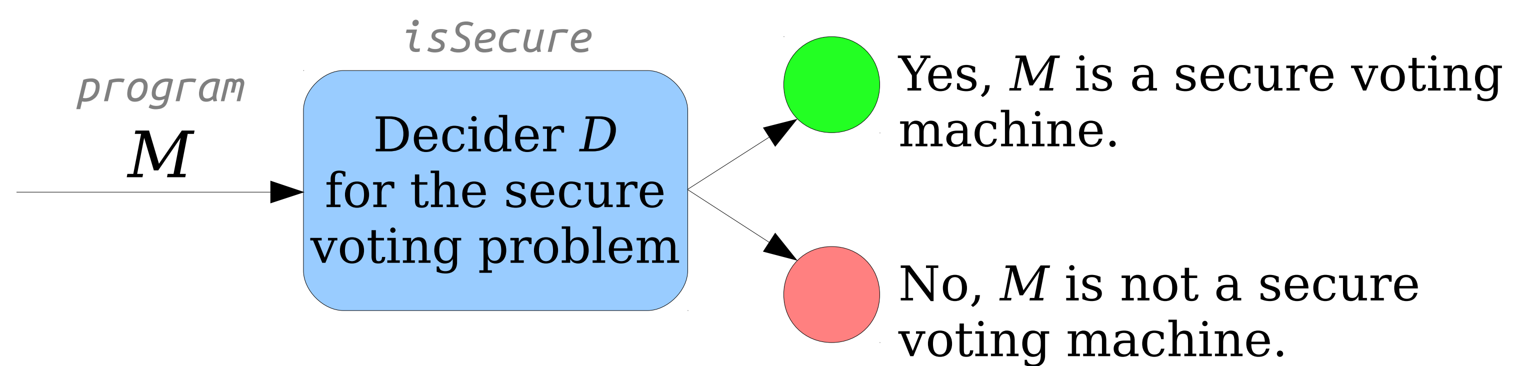
The secure voting problem is decidable.

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

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

Program *P* is secure if and only if program *P* is not secure.

Contradiction!



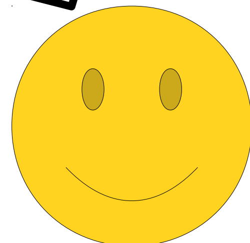

```
bool isSecure(string program)
```

Program *P* design specification:

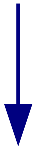
If *P* is a secure voting machine, then
P is not a secure voting machine.
If *P* is not a secure voting machine, then
P is a secure voting machine.

```
// Program P

int main() {

}
```

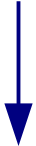
Ultimately, we need to figure out if we're a secure voting machine or not.



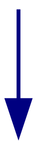
The secure voting problem is decidable.



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

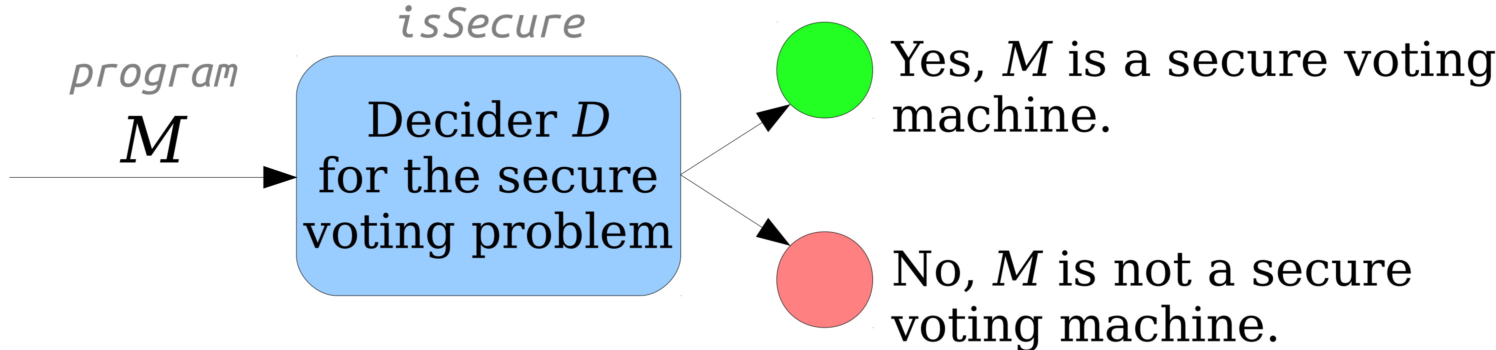


We can write programs that use *D* as a helper method



Program *P* is secure if and only if program *P* is not secure.

Contradiction!




```
bool isSecure(string program)
```

Program *P* design specification:

If *P* is a secure voting machine, then
P is not a secure voting machine.
If *P* is not a secure voting machine, then
P is a secure voting machine.

```
// Program P

int main() {

  string me = mySource();

}
```

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

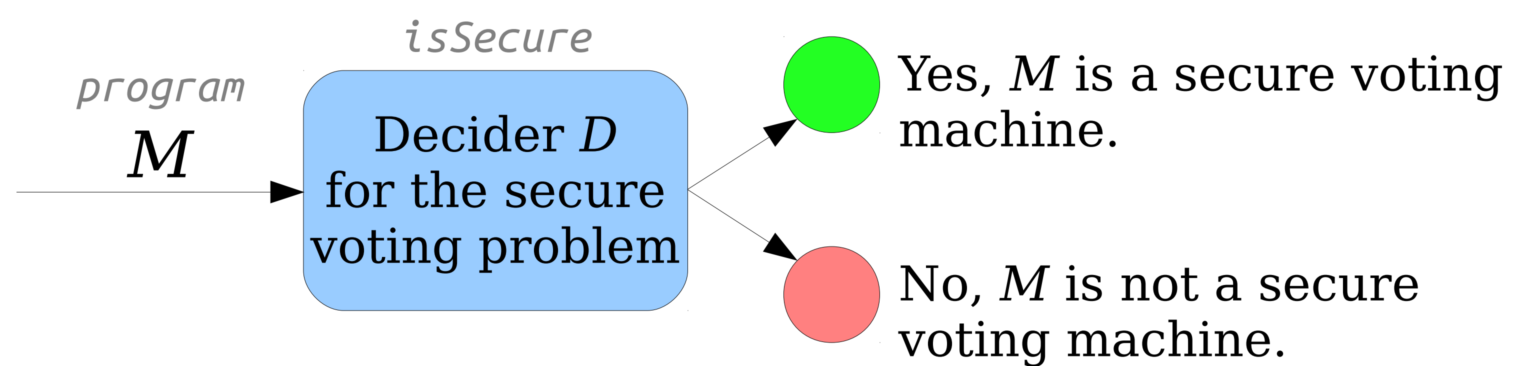
The secure voting problem is decidable.

There is a decider D for the secure voting problem

We can write programs that use D as a helper method

Program P is secure if and only if program P is not secure.

Contradiction!

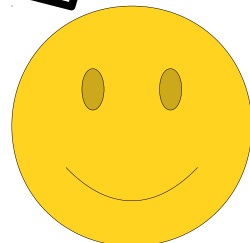



```
bool isSecure(string program)
```

Program P design specification:

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

```
// Program P

int main() {

  string me = mySource();

}
```



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

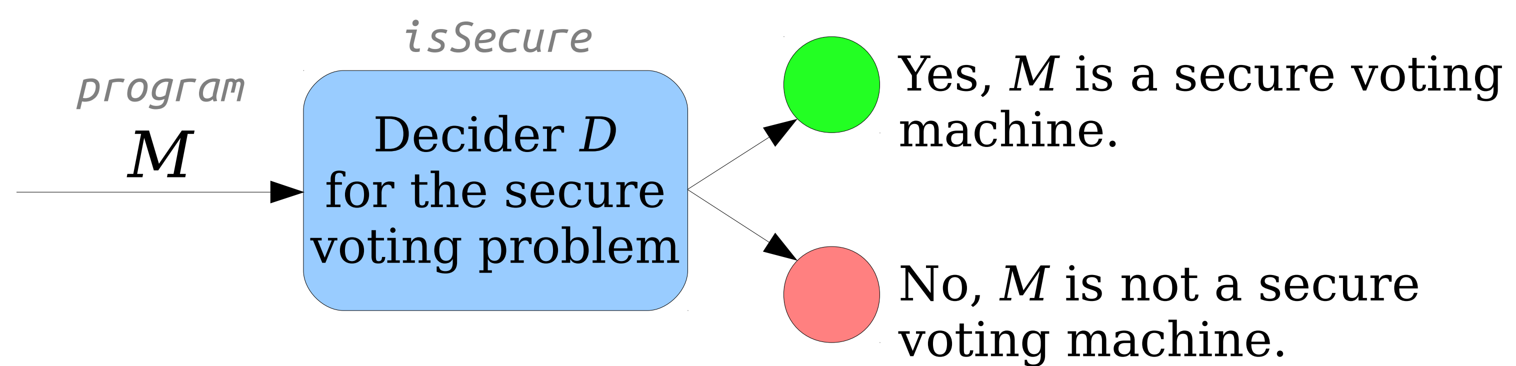
The secure voting problem is decidable.

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

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

Program *P* is secure if and only if program *P* is not secure.

Contradiction!



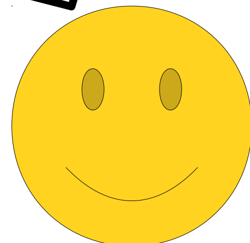

```
bool isSecure(string program)
```

Program *P* design specification:

If *P* is a secure voting machine, then
P is not a secure voting machine.
If *P* is not a secure voting machine, then
P is a secure voting machine.

```
// Program P

int main() {

  string me = mySource();

  if (isSecure(me)) {

  } else {


  }
}
```

Specifically, let's have program P ask what it's going to do.



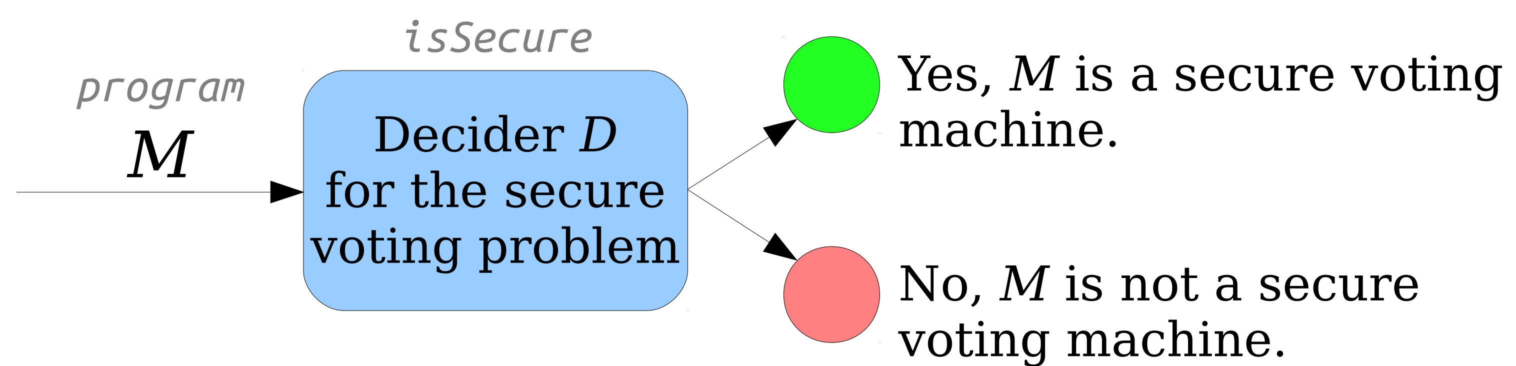
The secure voting problem is decidable.

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

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

Program *P* is secure if and only if program *P* is not secure.

Contradiction!

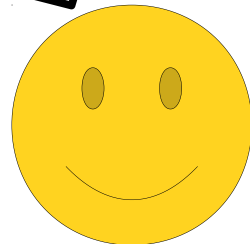



```
bool isSecure(string program)
```

Program *P* design specification:

If *P* is a secure voting machine, then
P is not a secure voting machine.
If *P* is not a secure voting machine, then
P is a secure voting machine.

```
// Program P

int main() {

  string me = mySource();

  if (isSecure(me)) {

  } else {


  }
}
```



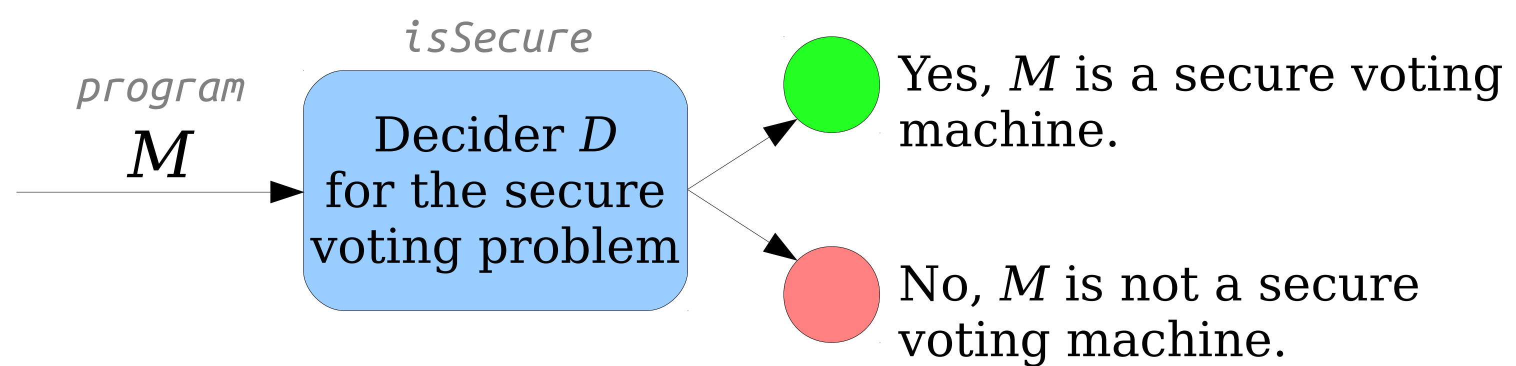
The secure voting problem is decidable.

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

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

Program *P* is secure if and only if program *P* is not secure.

Contradiction!

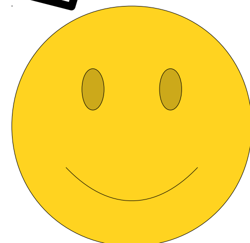



```
bool isSecure(string program)
```

Program *P* design specification:

If *P* is a secure voting machine, then
P is not a secure voting machine.
If *P* is not a secure voting machine, then
P is a secure voting machine.

```
// Program P

int main() {

  string me = mySource();

  if (isSecure(me)) {

  } else {


  }
}
```



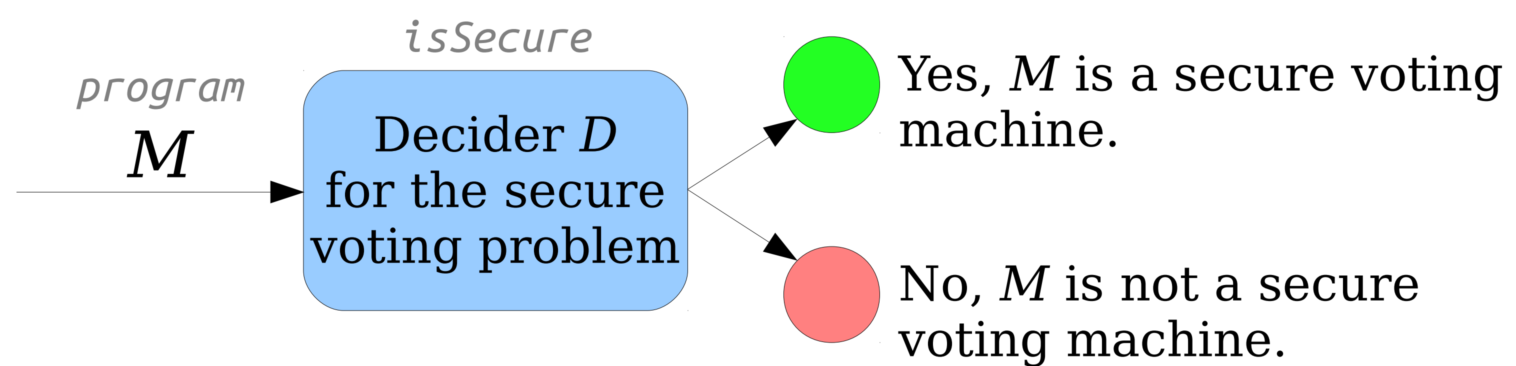
The secure voting problem is decidable.

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

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

Program *P* is secure if and only if program *P* is not secure.

Contradiction!



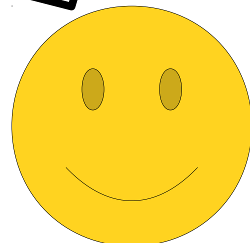

```
bool isSecure(string program)
```

Program *P* design specification:

If *P* is a secure voting machine, then
P is not a secure voting machine.

If *P* is not a secure voting machine, then
P is a secure voting machine.

```
// Program P

int main() {

  string me = mySource();

  if (isSecure(me)) {

  } else {


  }
}
```



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

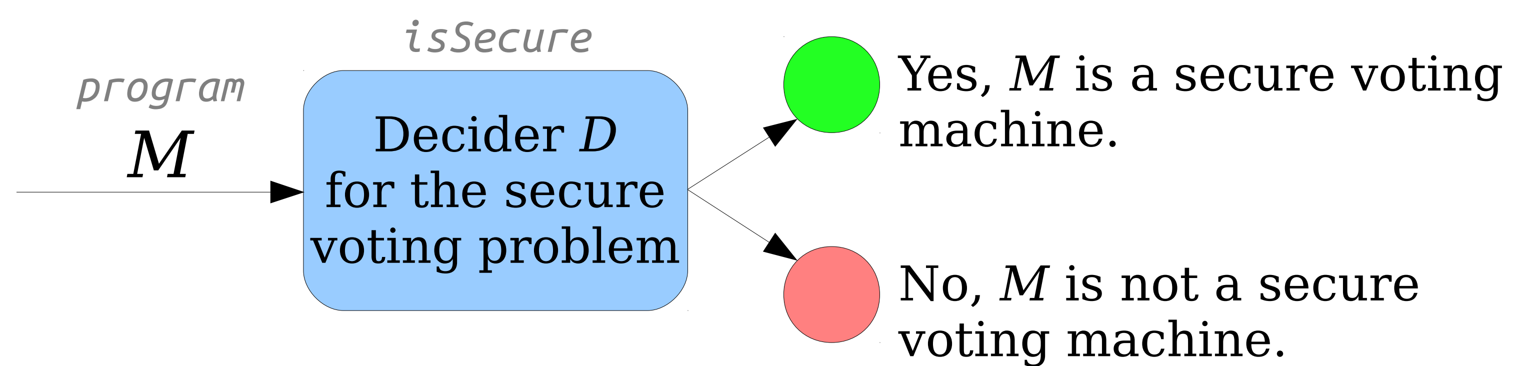
The secure voting problem is decidable.

There is a decider D for the secure voting problem

We can write programs that use D as a helper method

Program P is secure if and only if program P is not secure.

Contradiction!




```
bool isSecure(string program)
```

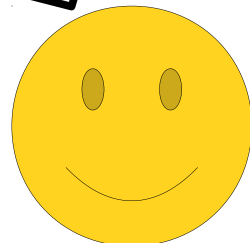
Program P design specification:

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

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

```
// Program P

int main() {

  string me = mySource();

  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.



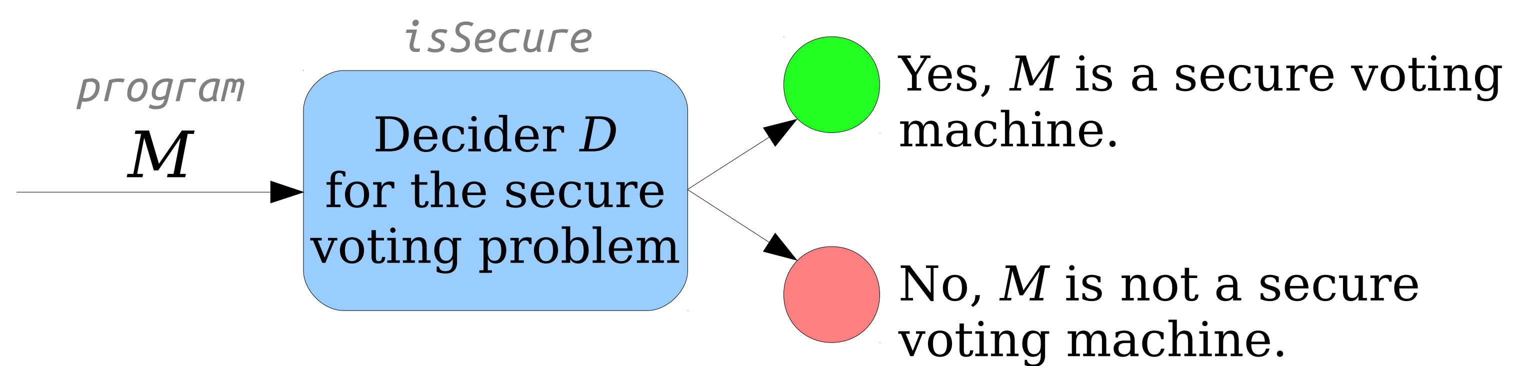
The secure voting problem is decidable.

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

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

Program *P* is secure if and only if program *P* is not secure.

Contradiction!

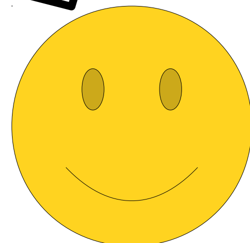



```
bool isSecure(string program)
```

Program *P* design specification:

If *P* is a secure voting machine, then
P is not a secure voting machine.
If *P* is not a secure voting machine, then
P is a secure voting machine.

```
// Program P

int main() {
   string me = mySource();
   if (isSecure(me)) {
   } else {

   }
}
```



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

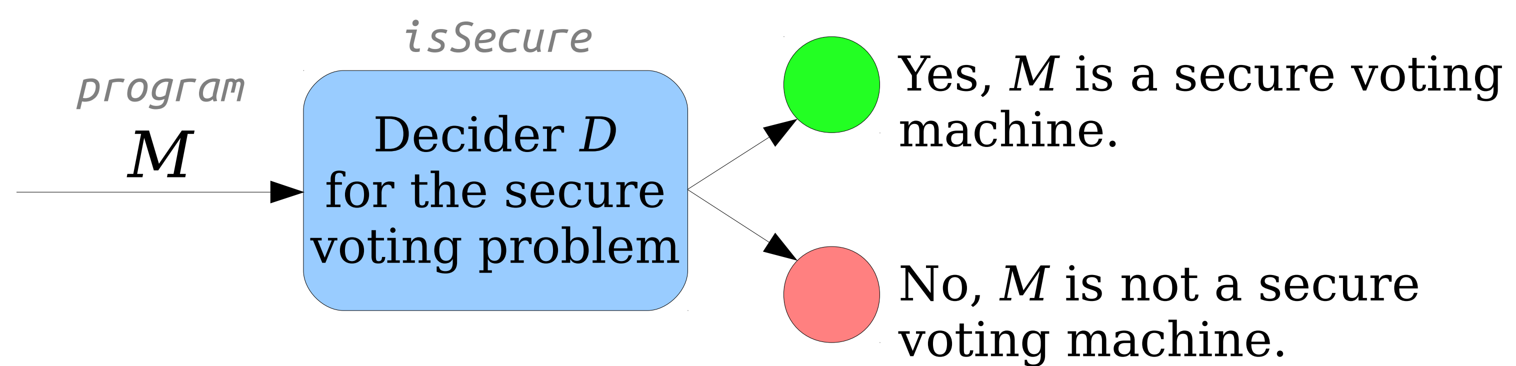
The secure voting problem is decidable.

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

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

Program *P* is secure if and only if program *P* is not secure.

Contradiction!



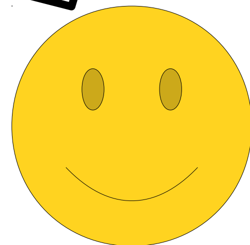

```
bool isSecure(string program)
```

Program *P* design specification:

If *P* is a secure voting machine, then
P is not a secure voting machine.
If *P* is not a secure voting machine, then
P is a secure voting machine.

```
// Program P

int main() {

  string me = mySource();

  if (isSecure(me)) {

  } else {


  }
}
```

That means we need to make P accept all strings with more r's than d's and not accept anything else.



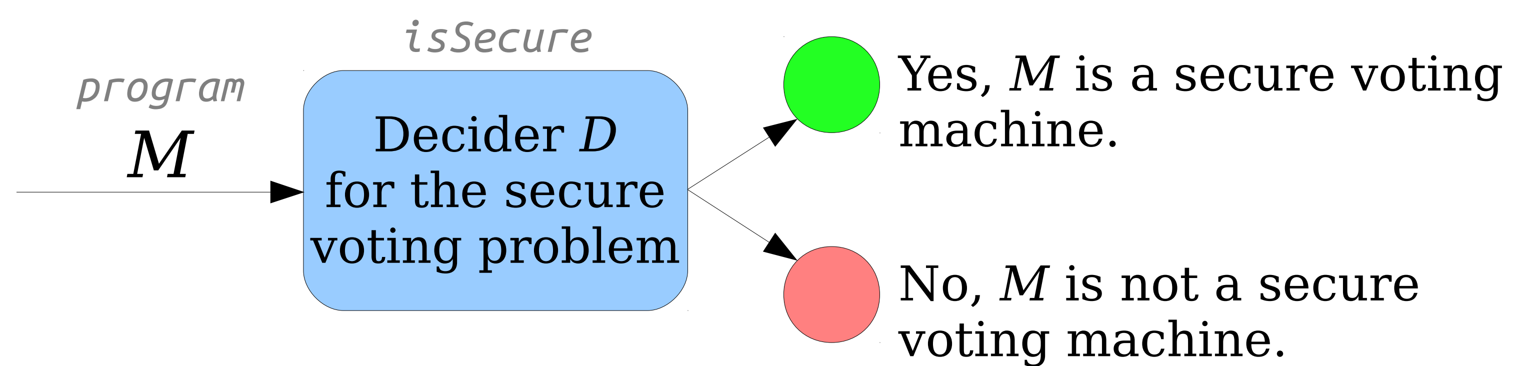
The secure voting problem is decidable.

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

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

Program *P* is secure if and only if program *P* is not secure.

Contradiction!




```
bool isSecure(string program)
```

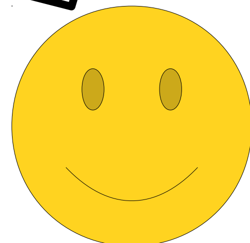
Program *P* design specification:

If *P* is a secure voting machine, then *P* is not a secure voting machine.

If *P* is not a secure voting machine, then *P* is a secure voting machine.

```
// Program P

int main() {

  string me = mySource();

  if (isSecure(me)) {

  } else {


  }
}
```

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



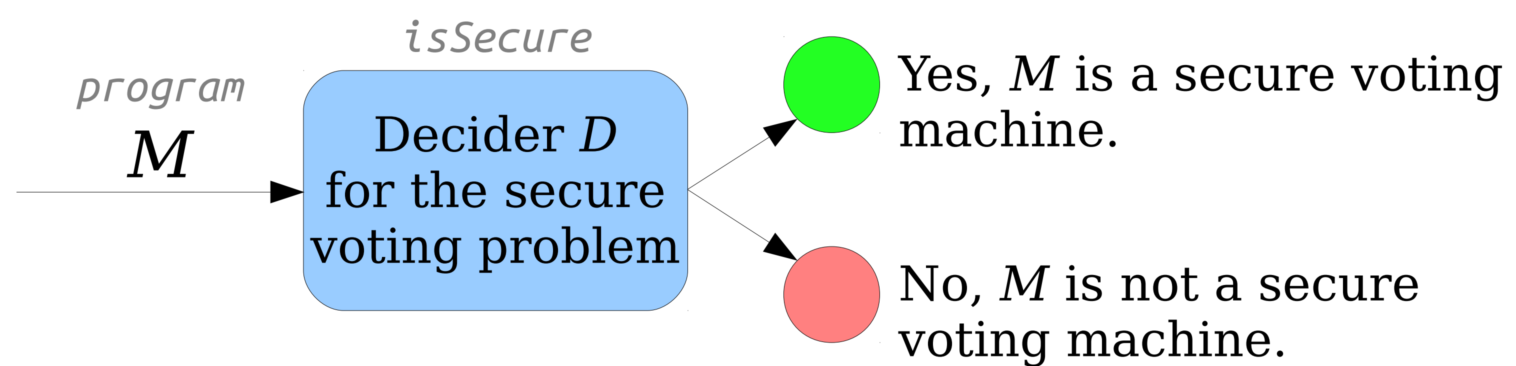
The secure voting problem is decidable.

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

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

Program *P* is secure if and only if program *P* is not secure.

Contradiction!




```
bool isSecure(string program)
```

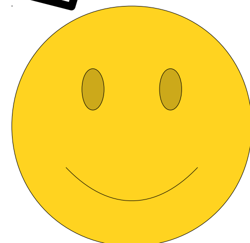
Program *P* design specification:

If *P* is a secure voting machine, then
P is not a secure voting machine.

If *P* is not a secure voting machine, then
P is a secure voting machine.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (isSecure(me)) {

  } else {
    if (countRs(input) > countDs(input)) accept();
    else reject();
  }
}
```

The code looks something like this.



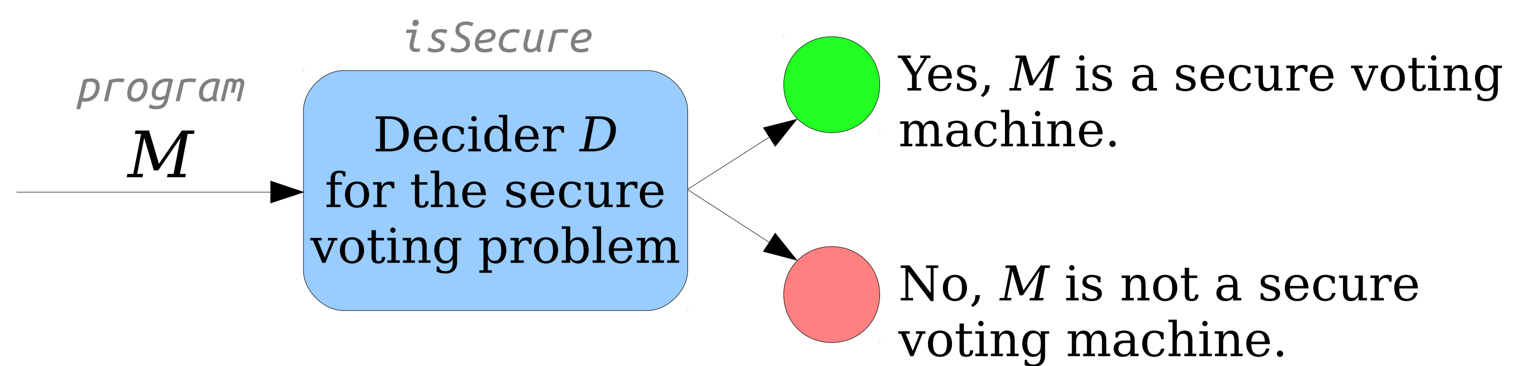
The secure voting problem is decidable.

There is a decider D for the secure voting problem

We can write programs that use D as a helper method

Program P is secure if and only if program P is not secure.

Contradiction!




```
bool isSecure(string program)
```

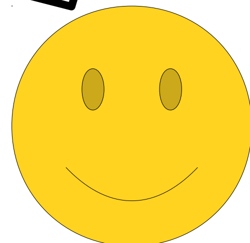
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (isSecure(me)) {

  } else {
    if (countRs(input) > countDs(input)) accept();
    else reject();
  }
}
```

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.



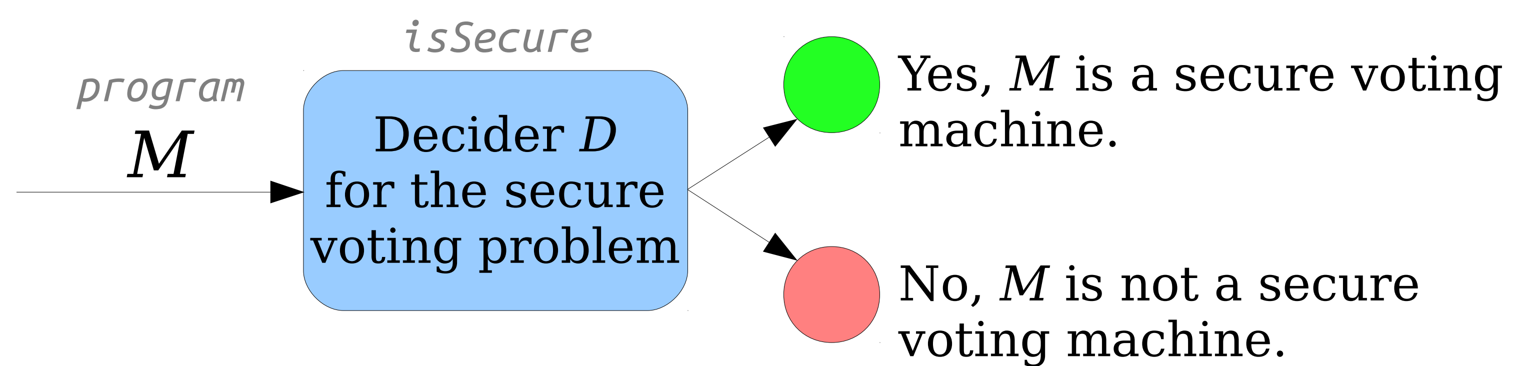
The secure voting problem is decidable.

There is a decider D for the secure voting problem

We can write programs that use D as a helper method

Program P is secure if and only if program P is not secure.

Contradiction!




```
bool isSecure(string program)
```

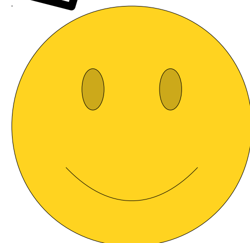
Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (isSecure(me)) {

  } else {
    if (countRs(input) > countDs(input)) accept();
    else reject();
  }
}
```

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



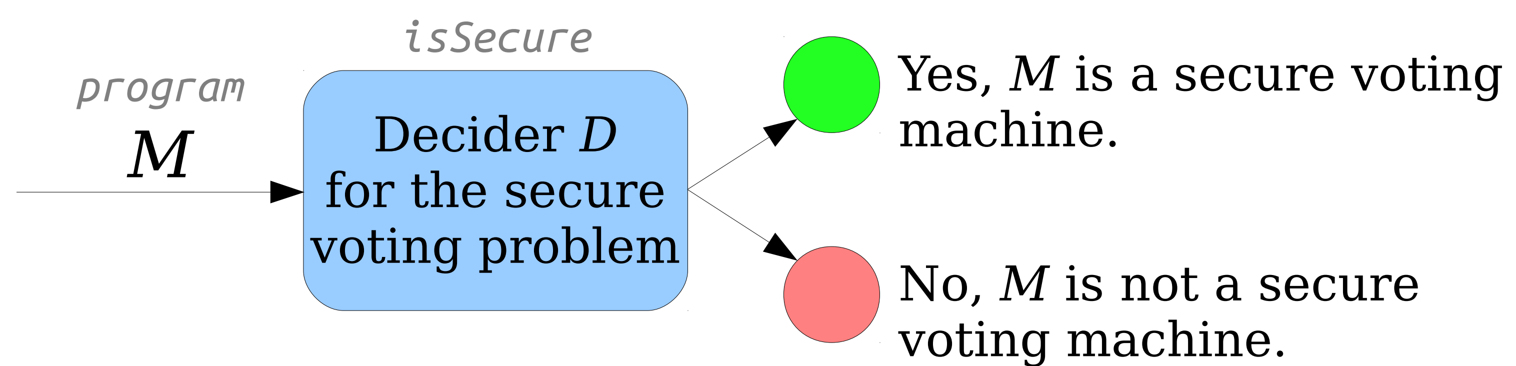
The secure voting problem is decidable.

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

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

Program *P* is secure if and only if program *P* is not secure.

Contradiction!




```
bool isSecure(string program)
```

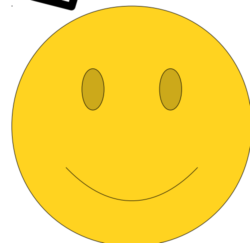
Program *P* design specification:

If *P* is a secure voting machine, then *P* is not a secure voting machine.

✓ If *P* is not a secure voting machine, then *P* is a secure voting machine.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (isSecure(me)) {

  } else {
    if (countRs(input) > countDs(input)) accept();
    else reject();
  }
}
```

Let's move on to the other one.



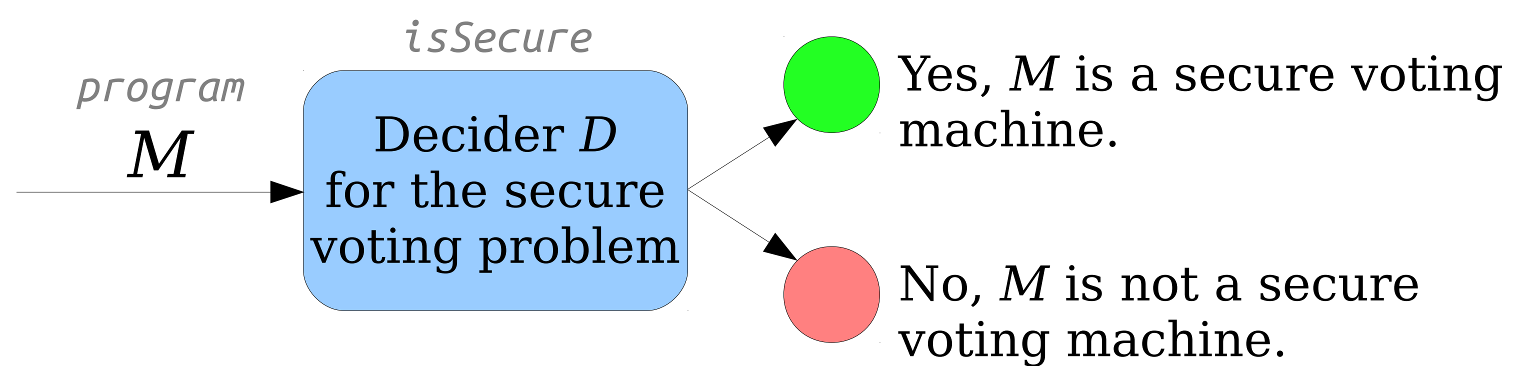
The secure voting problem is decidable.

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

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

Program *P* is secure if and only if program *P* is not secure.

Contradiction!




```
bool isSecure(string program)
```

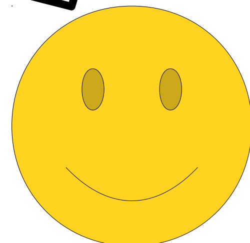
Program *P* design specification:

If *P* is a secure voting machine, then
P is not a secure voting machine.

✓ If *P* is not a secure voting machine, then
P is a secure voting machine.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (isSecure(me)) {

  } else {
    if (countRs(input) > countDs(input)) accept();
    else reject();
  }
}
```

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



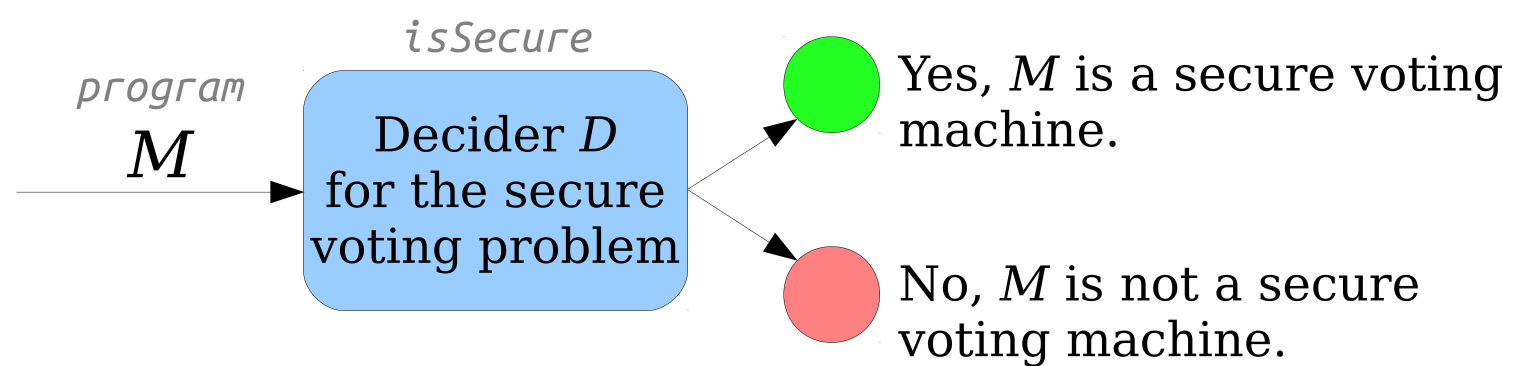
The secure voting problem is decidable.

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

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

Program *P* is secure if and only if program *P* is not secure.

Contradiction!




```
bool isSecure(string program)
```

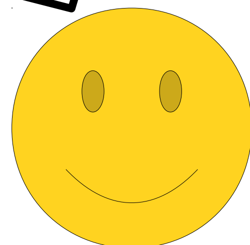
Program *P* design specification:

If *P* is a secure voting machine, then
P is not a secure voting machine.

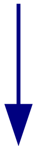
✓ If *P* is not a secure voting machine, then
P is a secure voting machine.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (isSecure(me)) {

  } else {
    if (countRs(input) > countDs(input)) accept();
    else reject();
  }
}
```

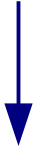
There are a lot of ways to get P to not be a secure voting machine.



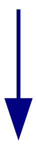
The secure voting problem is decidable.



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

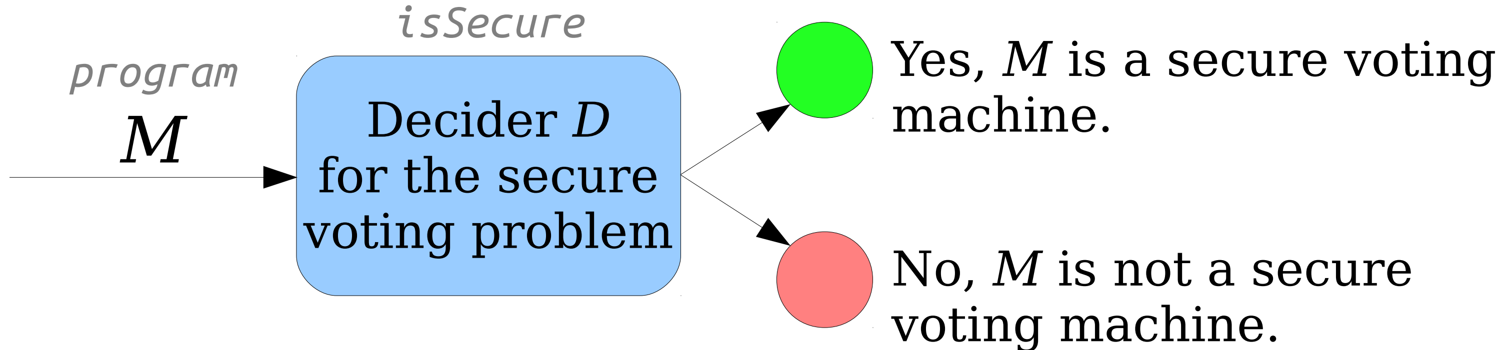


We can write programs that use *D* as a helper method



Program *P* is secure if and only if program *P* is not secure.

Contradiction!




```
bool isSecure(string program)
```

Program *P* design specification:

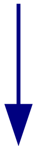
If *P* is a secure voting machine, then
P is not a secure voting machine.

✓ If *P* is not a secure voting machine, then
P is a secure voting machine.

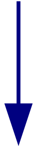
```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (isSecure(me)) {

  } else {
    if (countRs(input) > countDs(input)) accept();
    else reject();
  }
}
```

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

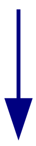
The secure voting problem is decidable.



There is a decider D for the secure voting problem

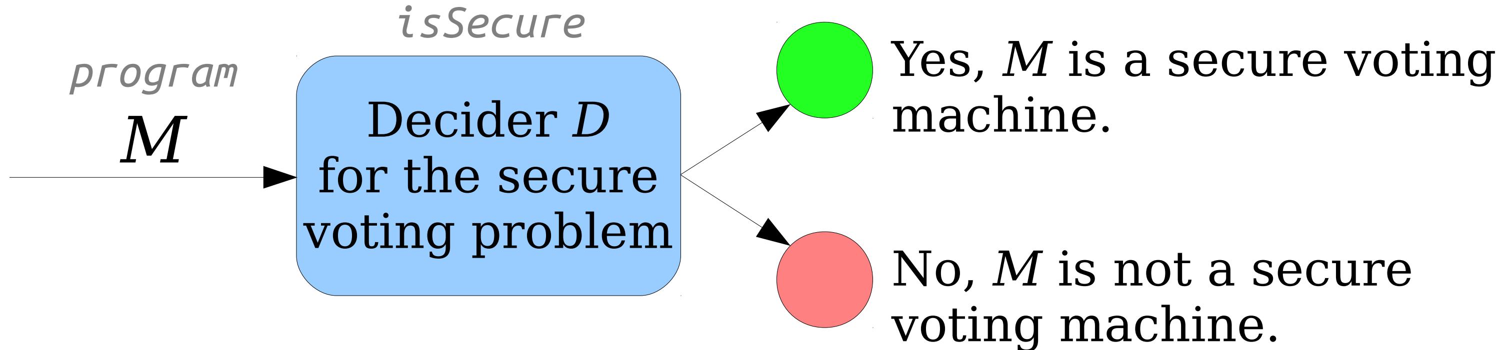


We can write programs that use D as a helper method



Program P is secure if and only if program P is not secure.

Contradiction!




```
bool isSecure(string program)
```

Program P design specification:

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

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (isSecure(me)) {
    accept();
  } else {
    if (countRs(input) > countDs(input)) accept();
    else reject();
  }
}
```

Among the many things we can do that falls into the "literally anything else" camp would be to just accept everything.

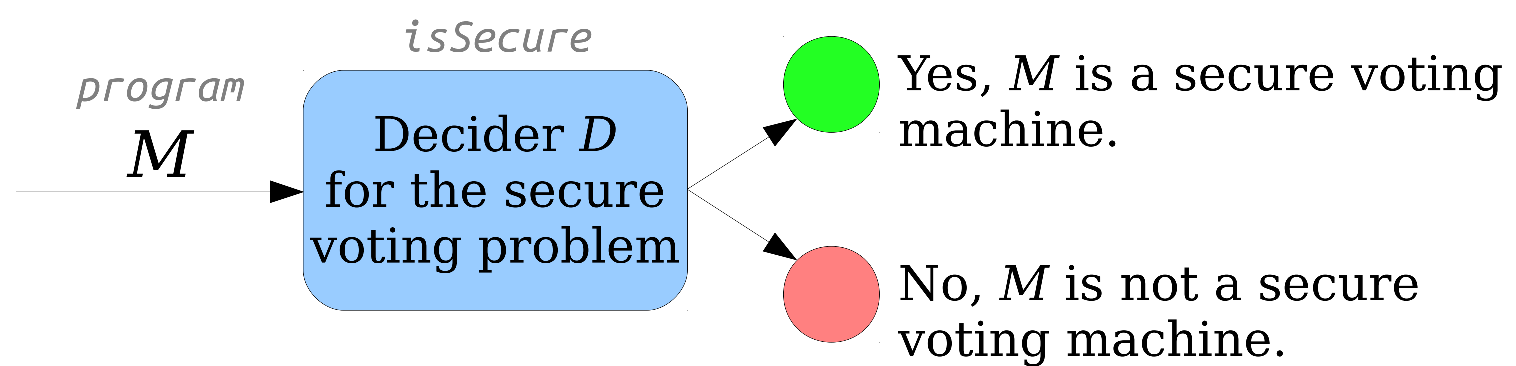
The secure voting problem is decidable.

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

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

Program *P* is secure if and only if program *P* is not secure.

Contradiction!




```
bool isSecure(string program)
```

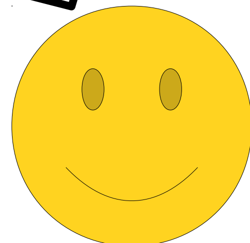
Program *P* design specification:

If *P* is a secure voting machine, then *P* is not a secure voting machine.

✓ If *P* is not a secure voting machine, then *P* is a secure voting machine.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (isSecure(me)) {
    accept();
  } else {
    if (countRs(input) > countDs(input)) accept();
    else reject();
  }
}
```

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



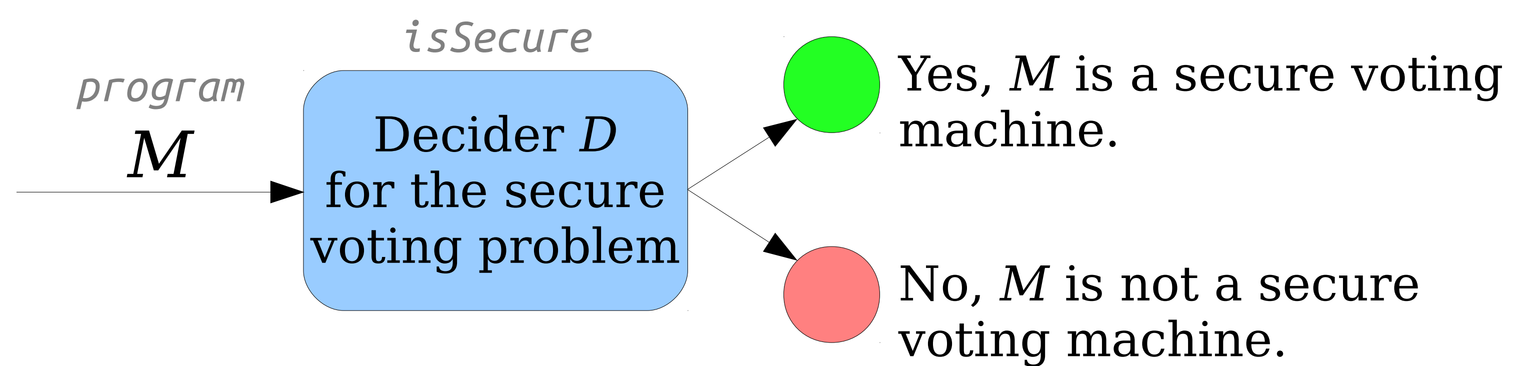
The secure voting problem is decidable.

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

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

Program *P* is secure if and only if program *P* is not secure.

Contradiction!




```
bool isSecure(string program)
```

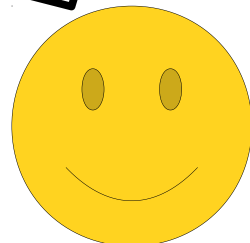
Program *P* design specification:

✓ If *P* is a secure voting machine, then *P* is not a secure voting machine.

✓ If *P* is not a secure voting machine, then *P* is a secure voting machine.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (isSecure(me)) {
    accept();
  } else {
    if (countRs(input) > countDs(input)) accept();
    else reject();
  }
}
```

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



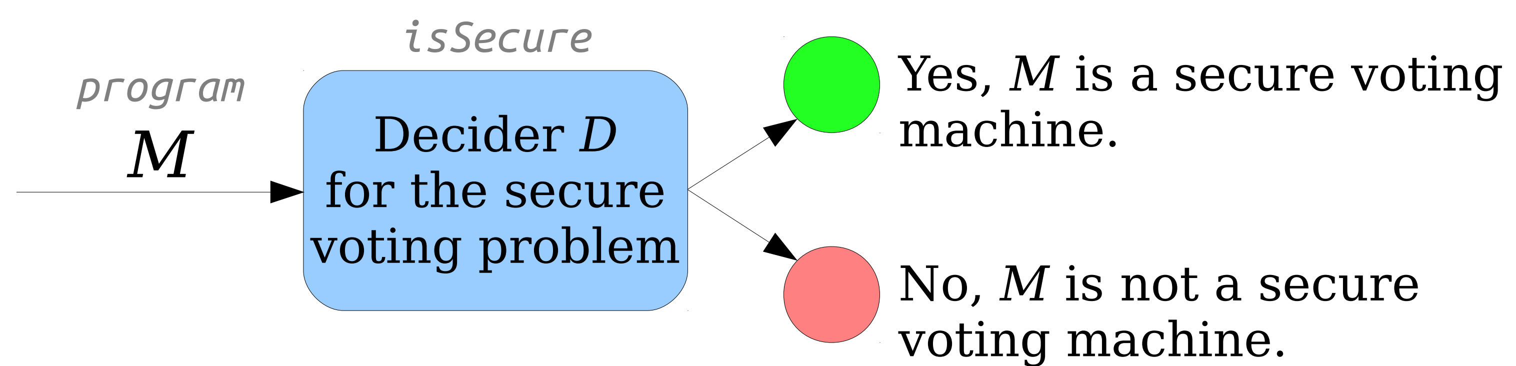
The secure voting problem is decidable.

There is a decider D for the secure voting problem

We can write programs that use D as a helper method

Program P is secure if and only if program P is not secure.

Contradiction!



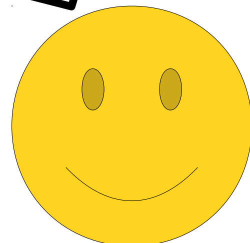

```
bool isSecure(string program)
```

Program P design specification:

- ✓ If P is a secure voting machine, then P is not a secure voting machine.
- ✓ If P is not a secure voting machine, then P is a secure voting machine.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (isSecure(me)) {
    accept();
  } else {
    if (countRs(input) > countDs(input)) accept();
    else reject();
  }
}
```

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



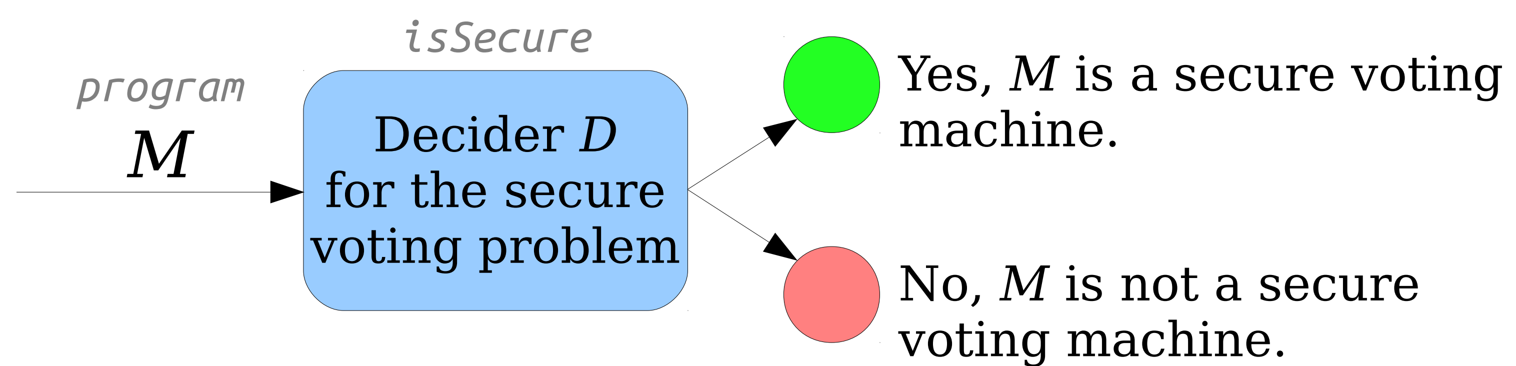
The secure voting problem is decidable.

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

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

Program *P* is secure if and only if program *P* is not secure.

Contradiction!




```
bool isSecure(string program)
```

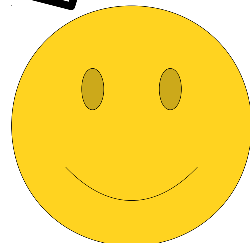
Program *P* design specification:

✓ If *P* is a secure voting machine, then *P* is not a secure voting machine.

✓ If *P* is not a secure voting machine, then *P* is a secure voting machine.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (isSecure(me)) {
    accept();
  } else {
    if (countRs(input) > countDs(input)) accept();
    else reject();
  }
}
```

That gives us the contradiction that we needed to get.



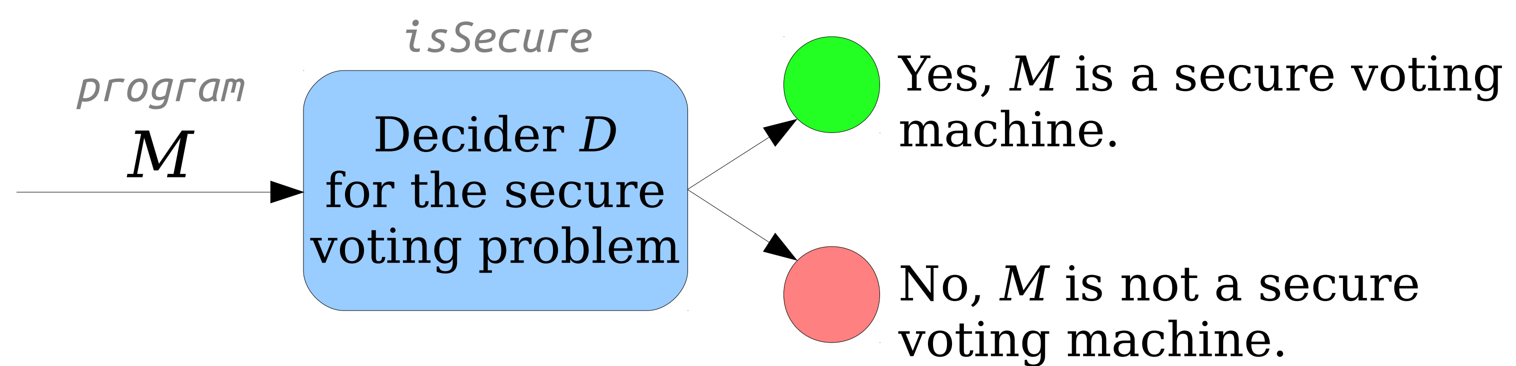
The secure voting problem is decidable.

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

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

Program *P* is secure if and only if program *P* is not secure.

Contradiction!




```
bool isSecure(string program)
```

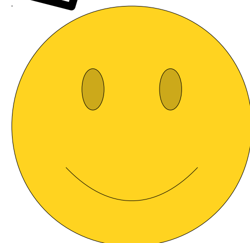
Program *P* design specification:

✓ If *P* is a secure voting machine, then
  P is not a secure voting machine.

✓ If *P* is not a secure voting machine, then
  P is a secure voting machine.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (isSecure(me)) {
    accept();
  } else {
    if (countRs(input) > countDs(input)) accept();
    else reject();
  }
}
```

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



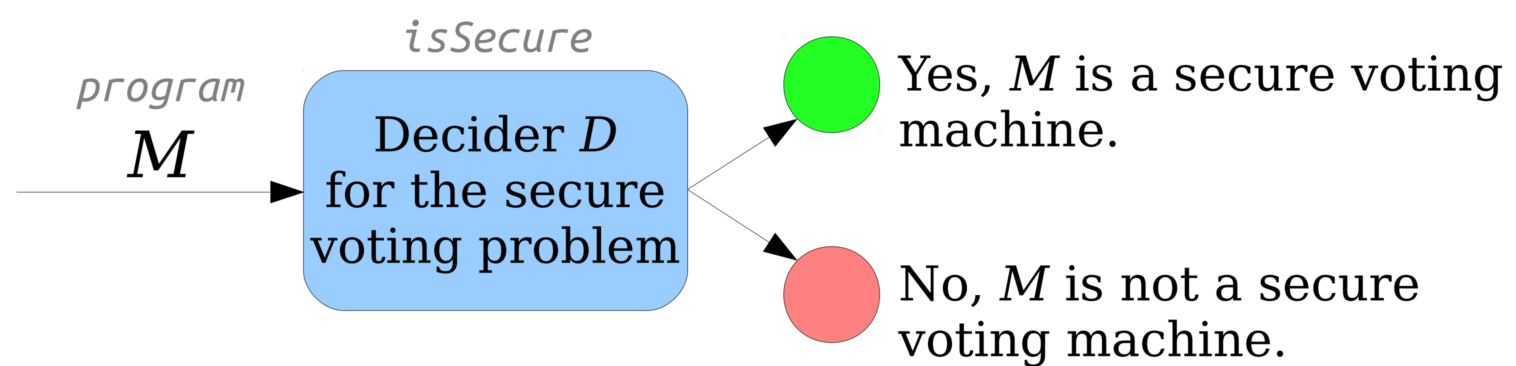
The secure voting problem is decidable.

There is a decider D for the secure voting problem

We can write programs that use D as a helper method

Program P is secure if and only if program P is not secure.

Contradiction!



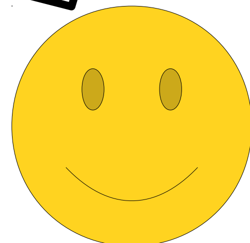

```
bool isSecure(string program)
```

Program P design specification:

- ✓ If P is a secure voting machine, then
  P is not a secure voting machine.
- ✓ If P is not a secure voting machine, then
  P is a secure voting machine.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (isSecure(me)) {
    accept();
  } else {
    if (countRs(input) > countDs(input)) accept();
    else reject();
  }
}
```

You might have noticed that this program isn't the one we used in lecture. But that's okay!



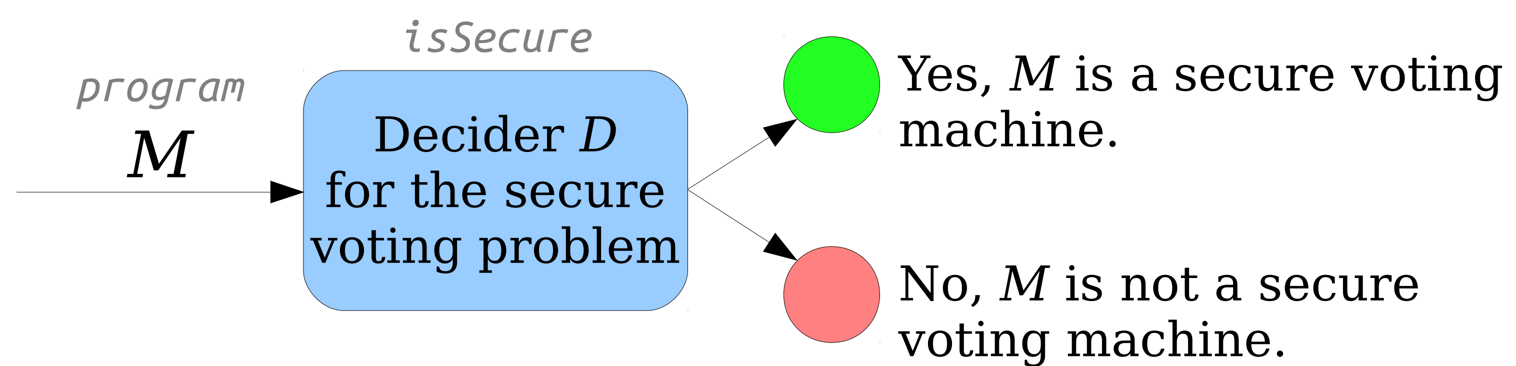
The secure voting problem is decidable.

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

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

Program *P* is secure if and only if program *P* is not secure.

Contradiction!



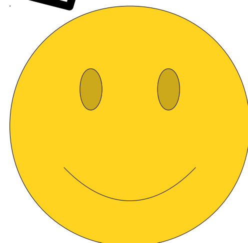

```
bool isSecure(string program)
```

Program *P* design specification:

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (isSecure(me)) {
    accept();
  } else {
    if (countRs(input) > countDs(input)) accept();
    else reject();
  }
}
```

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



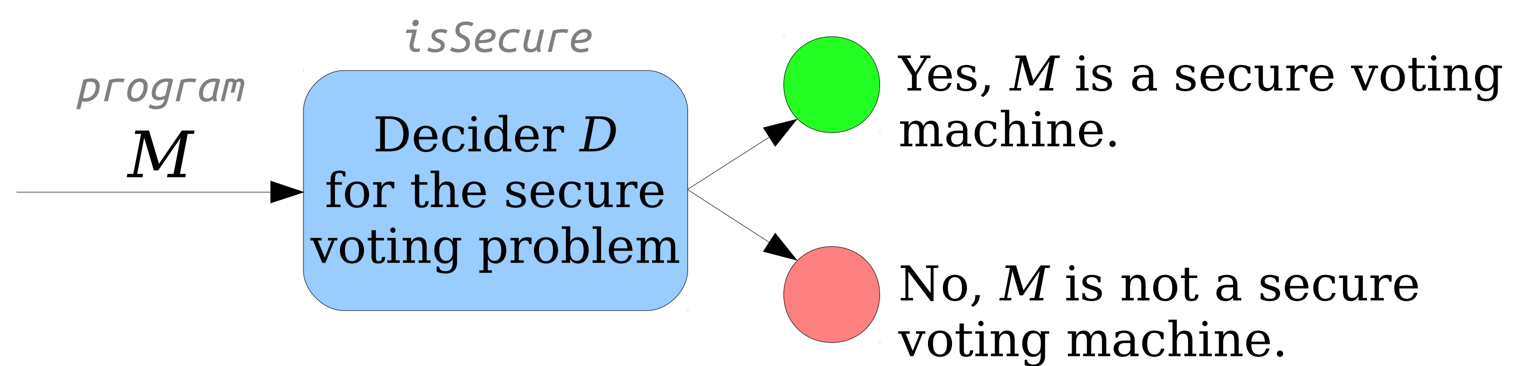
The secure voting problem is decidable.

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

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

Program *P* is secure if and only if program *P* is not secure.

Contradiction!




```
bool isSecure(string program)
```

Program *P* design specification:

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (isSecure(me)) {
    accept();
  } else {
    if (countRs(input) > countDs(input)) accept();
    else reject();
  }
}
```

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

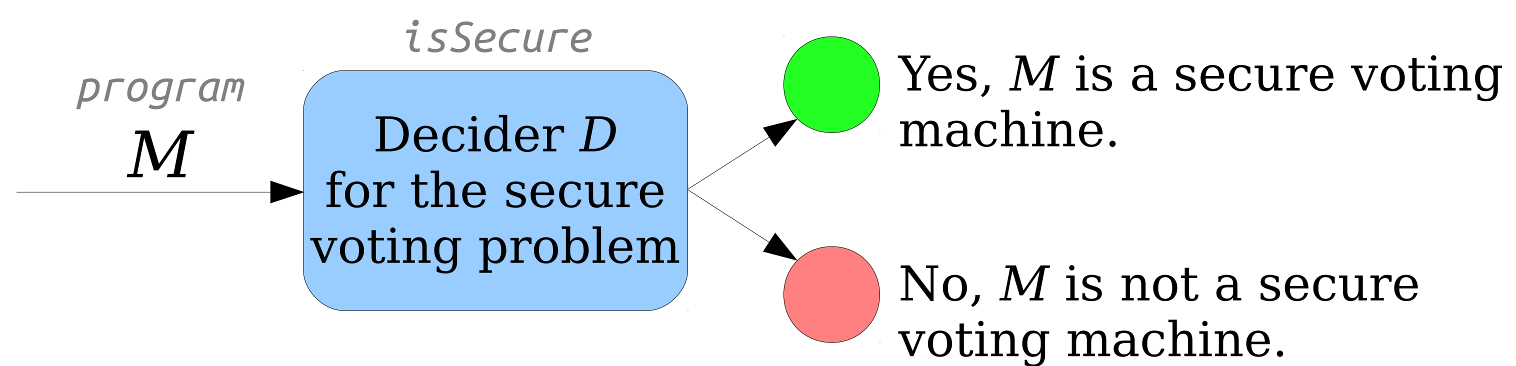
The secure voting problem is decidable.

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

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

Program *P* is secure if and only if program *P* is not secure.

Contradiction!



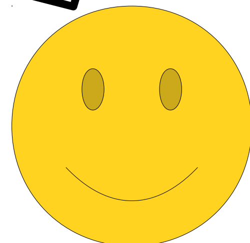

```
bool isSecure(string program)
```

Program *P* design specification:

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

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (isSecure(me)) {
    accept();
  } else {
    if (countRs(input) > countDs(input)) accept();
    else reject();
  }
}
```

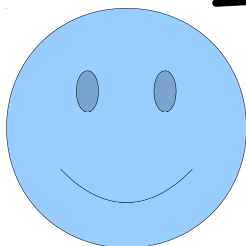
As long as you meet the design criteria, you should be good to go!



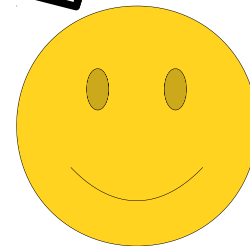
Let's take a minute to review the general process that we followed to get these results to work.

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.

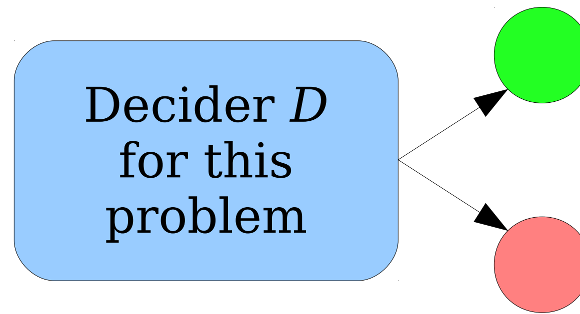
Contradiction!

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

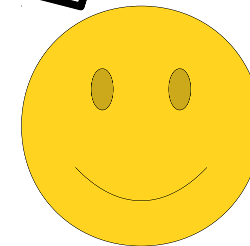
The problem in question is decidable

There is a decider D for that problem.

Contradiction!

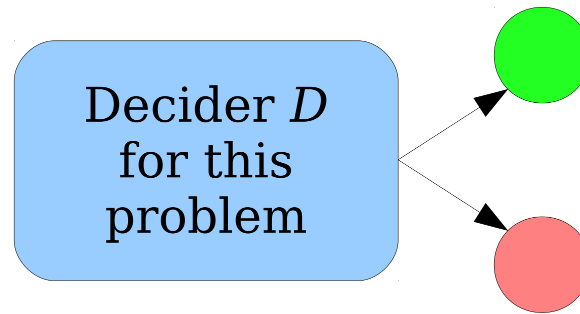


It's often a good idea to draw a picture showing what that decider looks like.



The problem in question is decidable

There is a decider *D* for that problem.

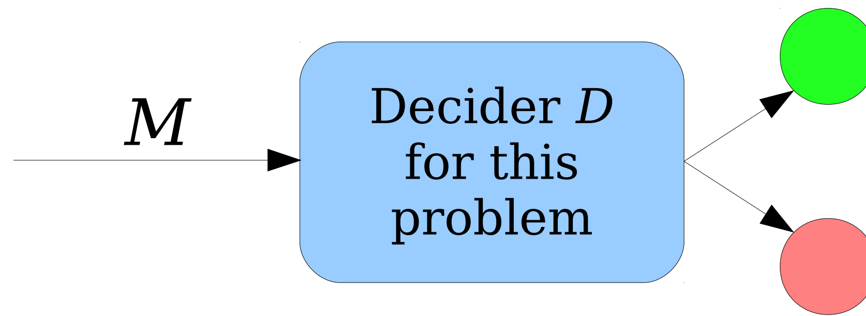


Think about what the inputs to the decider are going to look like. That depends on the language.

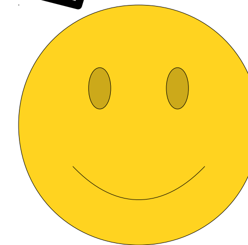
Contradiction!

The problem in question is decidable

There is a decider *D* for that problem.



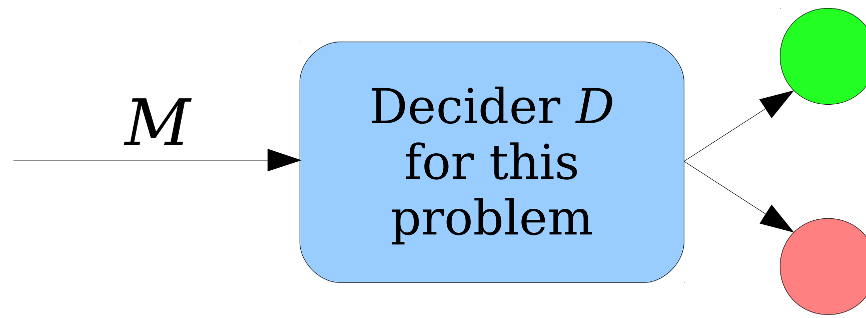
In the cases we're exploring in this class, there will always be at least one input that's a TM of some sort.



Contradiction!

The problem in question is decidable

There is a decider D for that problem.



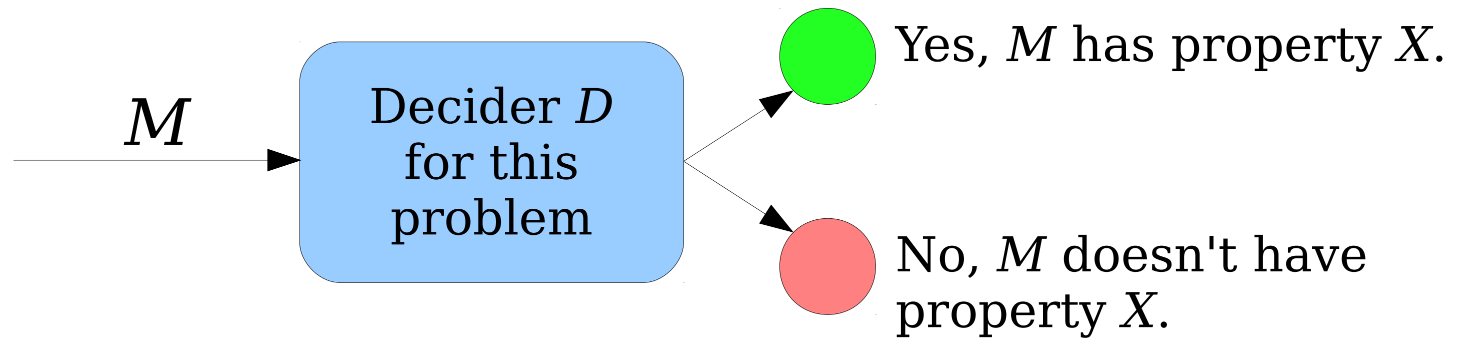
Next, think about what the decider is going to tell you about those inputs. That depends on the problem at hand.

Contradiction!

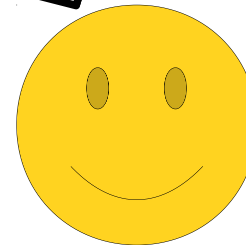
The problem in question is decidable

There is a decider D for that problem.

Contradiction!



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

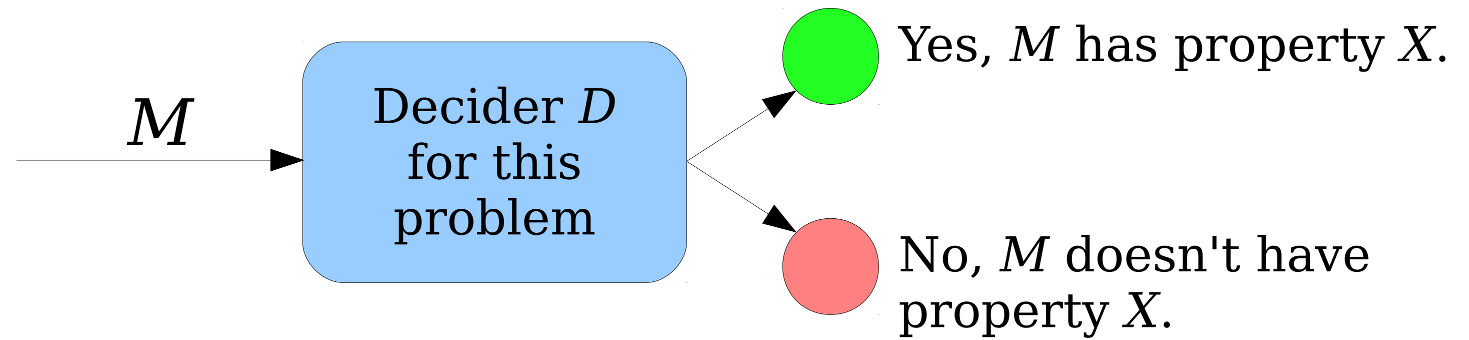


The problem in question is decidable

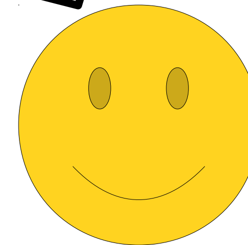
There is a decider *D* for that problem.

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

Contradiction!



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

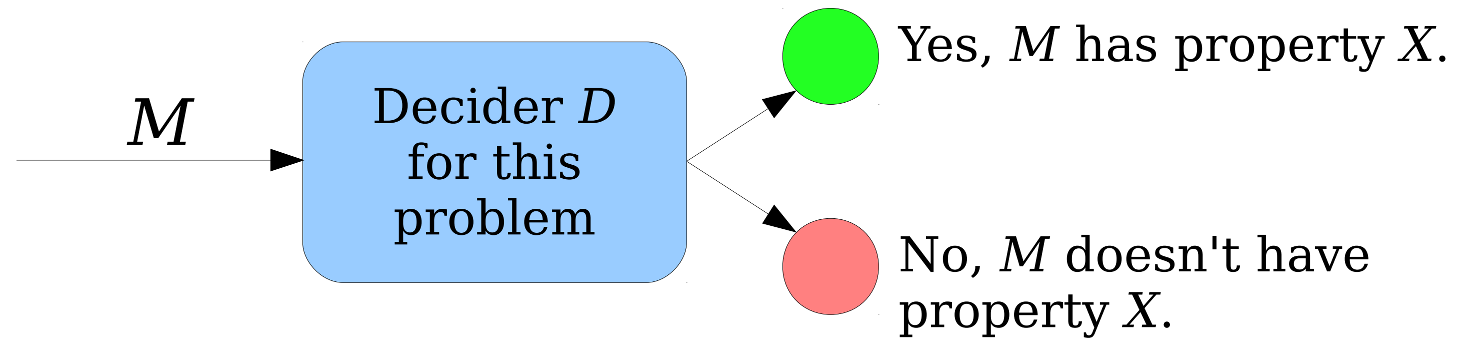


The problem in question is decidable

There is a decider D for that problem.

We can write programs that use D as a helper method

Contradiction!



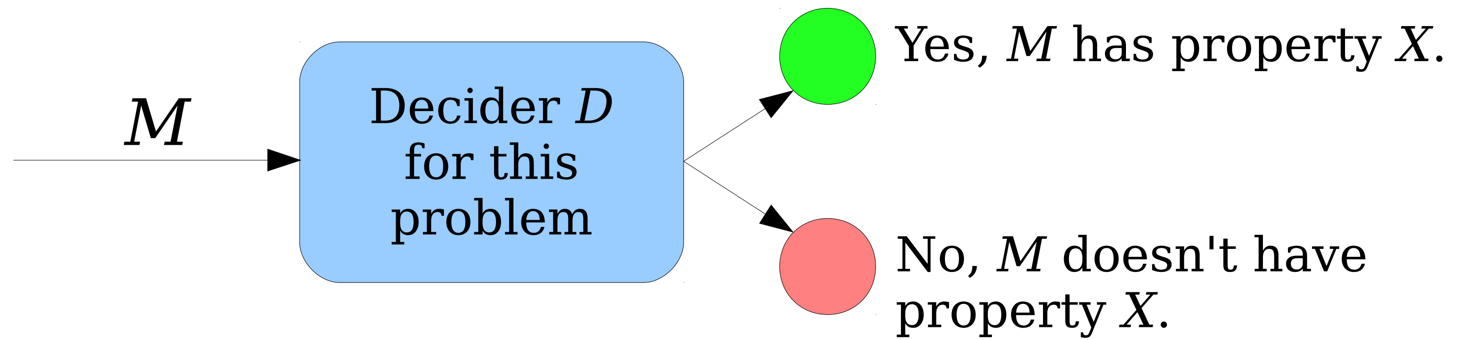
Think about what the decider would look like as a method in some high-level programming language.

The problem in question is decidable

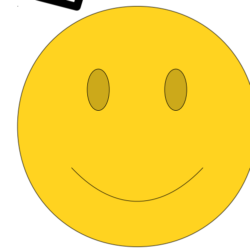
There is a decider *D* for that problem.

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

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 method.

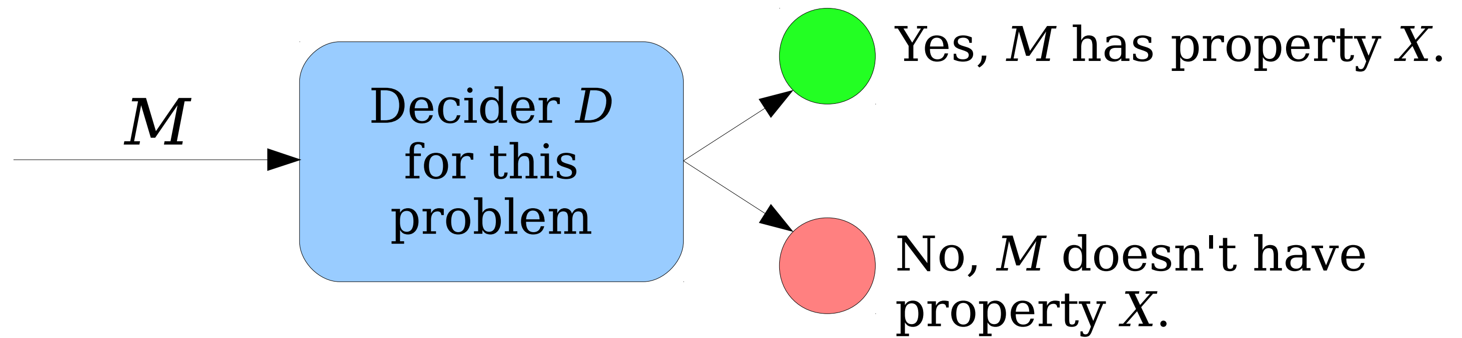


The problem in question is decidable

There is a decider D for that problem.

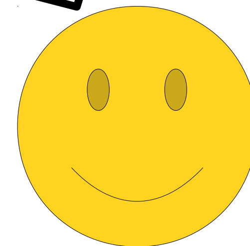
We can write programs that use D as a helper method

Contradiction!




```
bool hasPropertyX(string program)
```

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

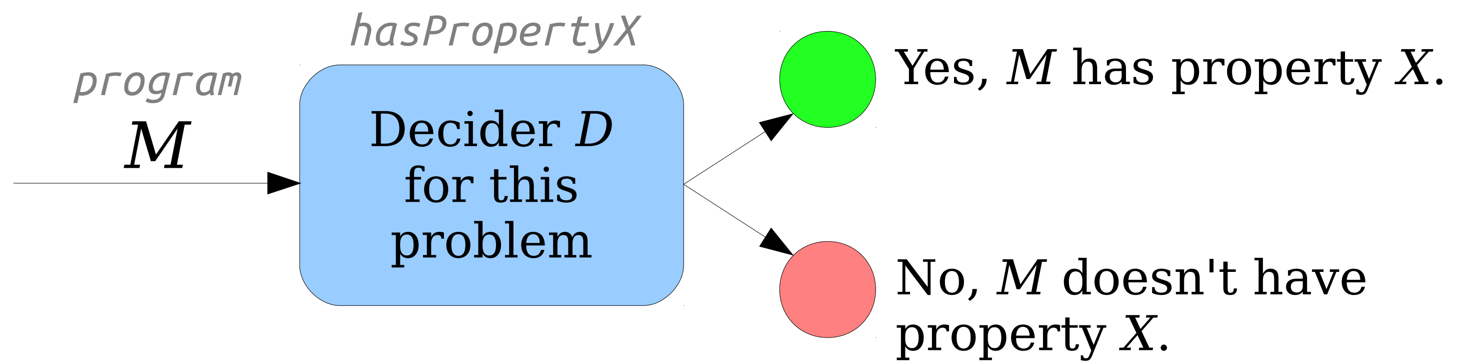


The problem in question is decidable

There is a decider D for that problem.

We can write programs that use D as a helper method

Contradiction!




```
bool hasPropertyX(string program)
```

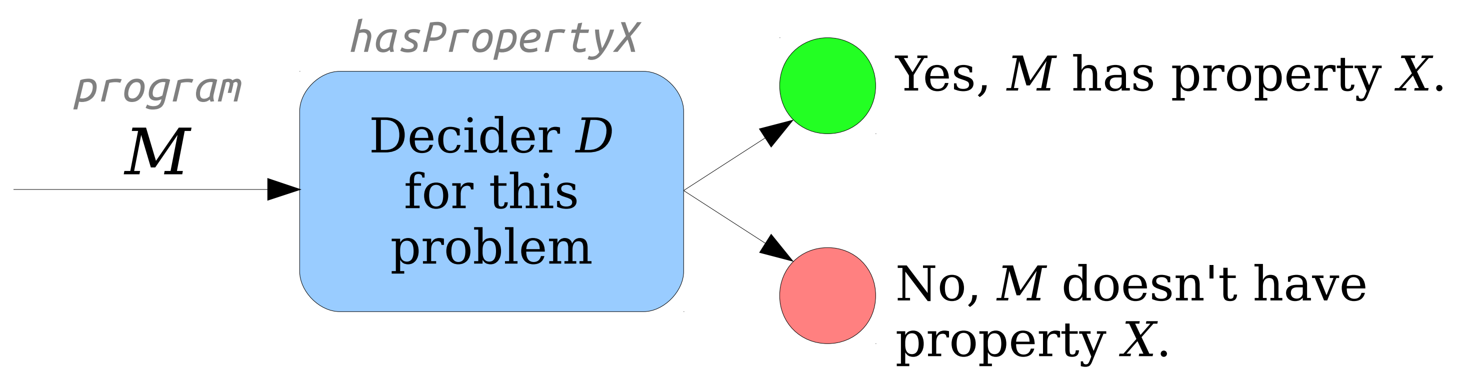
It doesn't hurt to label the decider D to show what parts of the decider correspond with the method.

The problem in question is decidable

There is a decider D for that problem.

We can write programs that use D as a helper method

Contradiction!




```
bool hasPropertyX(string program)
```

The next step is to build a self-referential program that gives you some sort of contradiction.

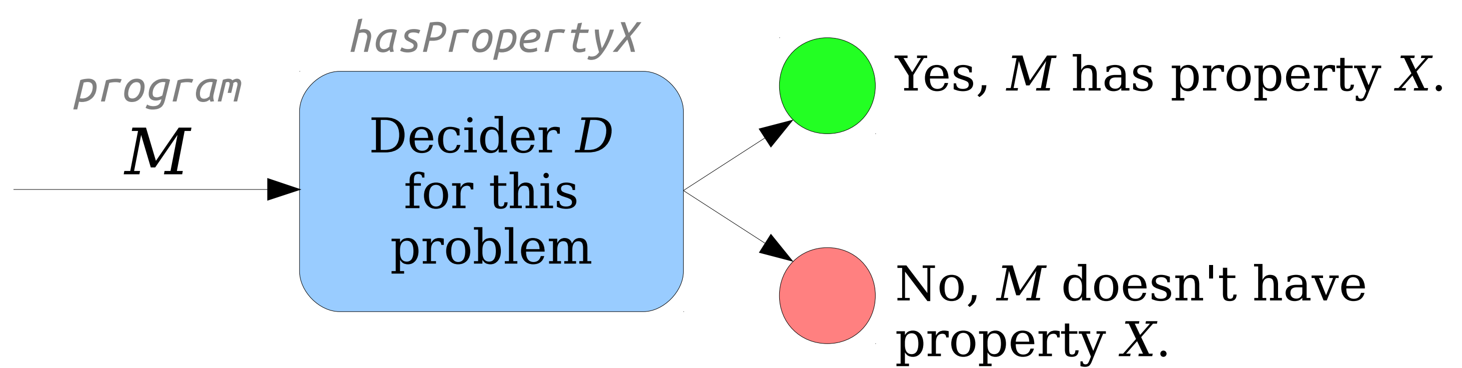
The problem in question is decidable

There is a decider D for that problem.

We can write programs that use D as a helper method

Program P has property X if and only if P doesn't have property X

Contradiction!




```
bool hasPropertyX(string program)
```

You're going to want to get a contradiction by building a program that has some property X if and only if it doesn't have some property X.

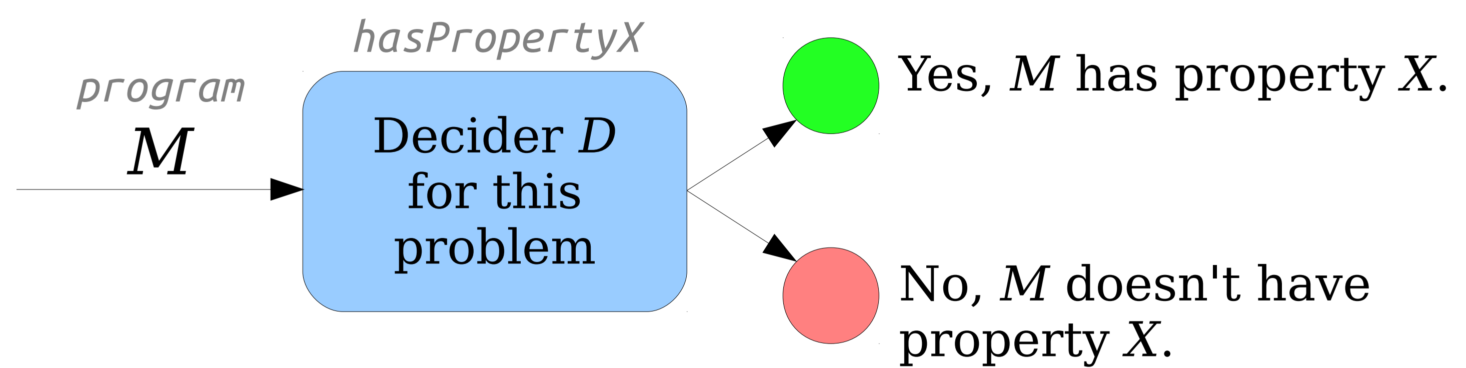
The problem in question is decidable

There is a decider *D* for that problem.

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

Program *P* has property *X* if and only if *P* doesn't have property *X*

Contradiction!




```
bool hasPropertyX(string program)
```

Now, you have to figure out how to write program P.

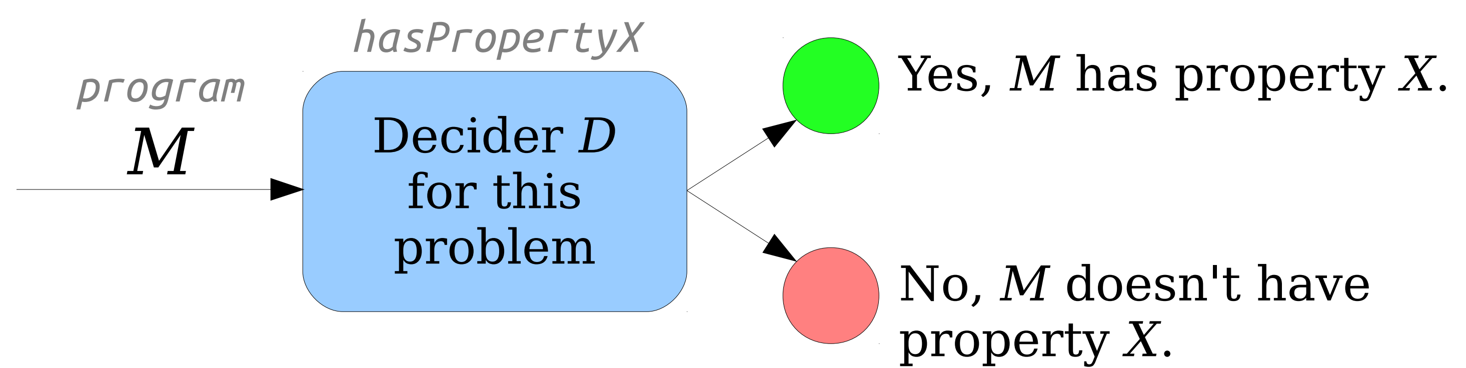
The problem in question is decidable

There is a decider D for that problem.

We can write programs that use D as a helper method

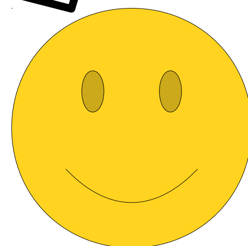
Program P has property X if and only if P doesn't have property X

Contradiction!




```
bool hasPropertyX(string program)
```

Program P design specification:



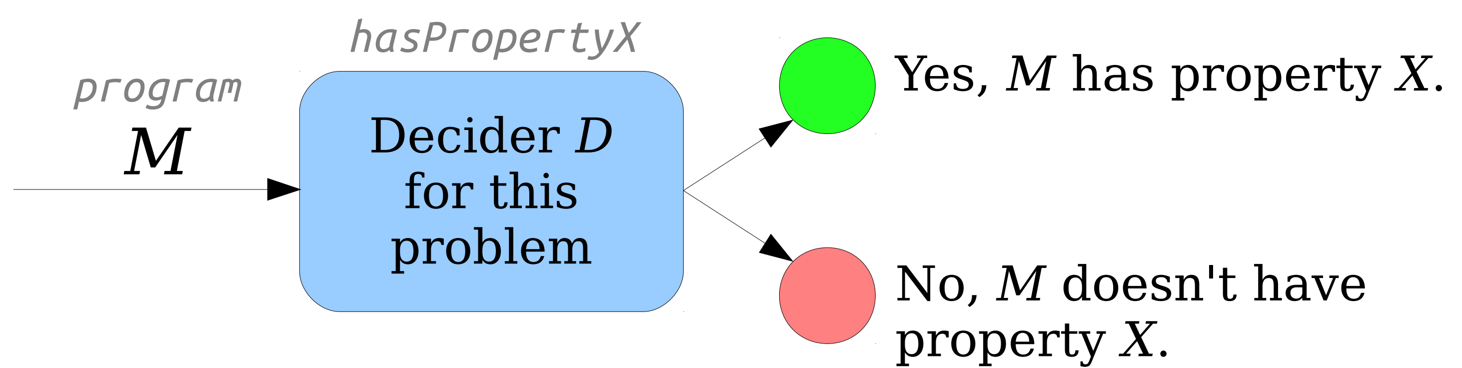
The problem in question is decidable

There is a decider *D* for that problem.

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

Program *P* has property *X* if and only if *P* doesn't have property *X*

Contradiction!




```
bool hasPropertyX(string program)
```

Program *P* design specification:

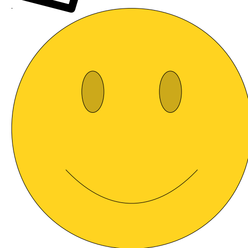
If *P* has property *X*, then

P does not have property *X*.

If *P* does not have property *X*, then

P has property *X*.

You can fill out that spec by reasoning about both directions of the implication.



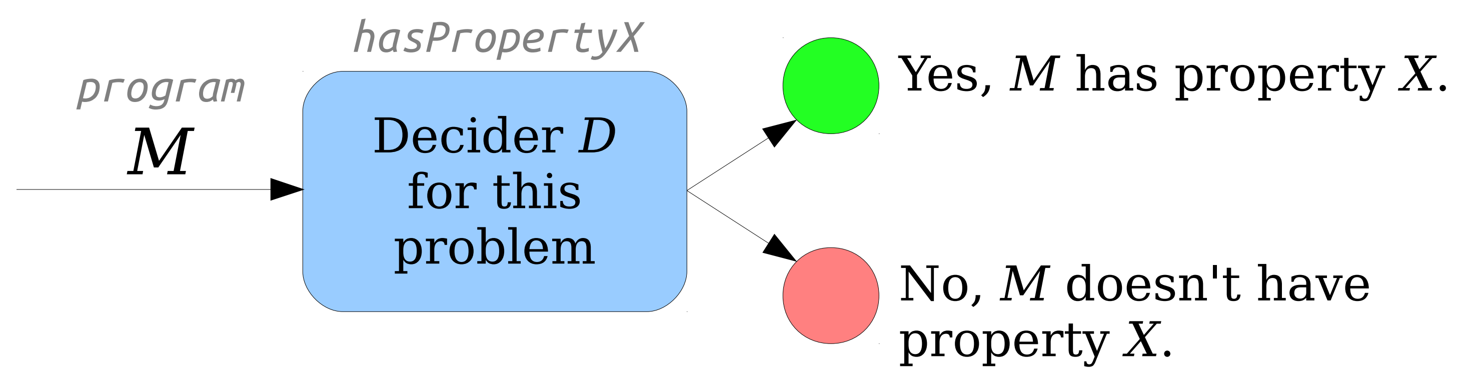
The problem in question is decidable

There is a decider *D* for that problem.

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

Program *P* has property *X* if and only if *P* doesn't have property *X*

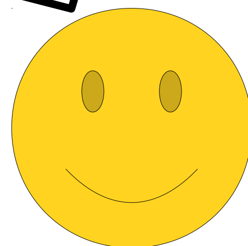
Contradiction!




```
bool hasPropertyX(string program)
```

Program *P* design specification:

If *P* has property *X*, then
P does not have property *X*.
If *P* does not have property *X*, then
P has property *X*.



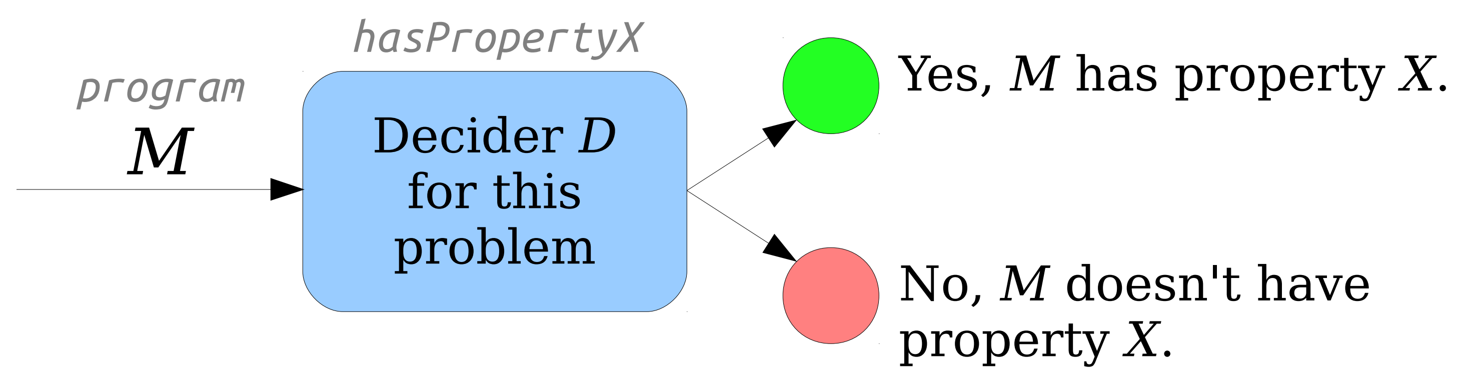
The problem in question is decidable

There is a decider D for that problem.

We can write programs that use D as a helper method

Program P has property X if and only if P doesn't have property X

Contradiction!




```
bool hasPropertyX(string program)
```

Program P design specification:

If P has property X, then
P does not have property X.
If P does not have property X, then
P has property X.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (hasPropertyX(me)) {
    // do something so you don't
    // have property X.
  } else {
    // Do something so you do
    // have property X.
  }
}
```

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

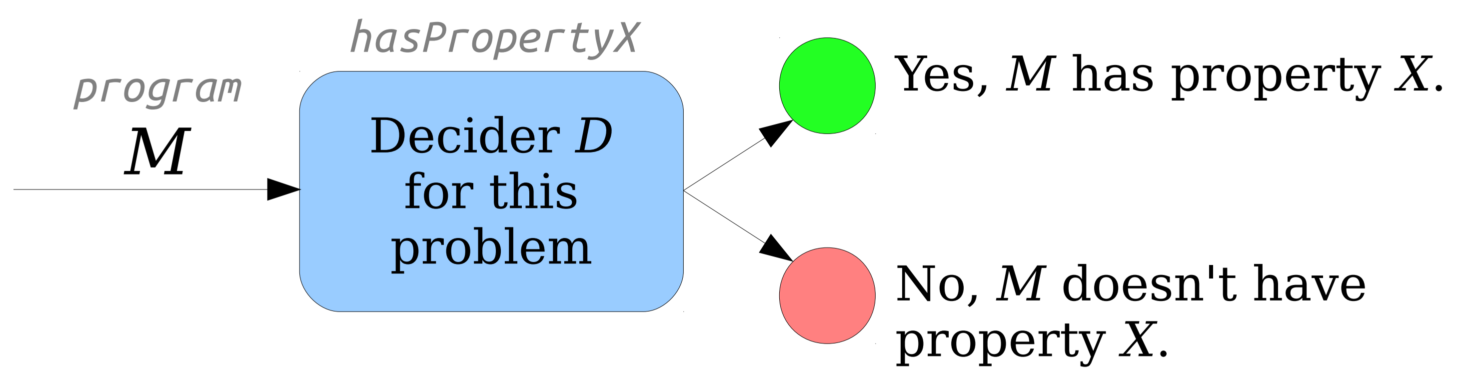
The problem in question is decidable

There is a decider D for that problem.

We can write programs that use D as a helper method

Program P has property X if and only if P doesn't have property X

Contradiction!

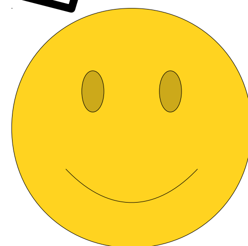



```
bool hasPropertyX(string program)
```

Program P design specification:

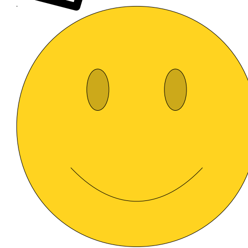
If P has property X, then
P does not have property X.
If P does not have property X, then
P has property X.

```
// Program P

int main() {
  string input = getInput();
  string me = mySource();

  if (hasPropertyX(me)) {
    // do something so you don't
    // have property X.
  } else {
    // Do something so you do
    // have property X.
  }
}
```



Hope this helps!

Please feel free to ask questions if you have them.



Did you find this useful? If so, let us know! We can go and make more guides like these.