CMPU 240 · Theory of Computation

The Limits of Computation

28 April 2026

The Limits of Computation

Exam 2 grades

Extra credit 🥺

Where are we?

We've introduced our final model of a computer, the Turing machine.

We designed Turing machines at the level of individual states and transitions.

We've also seen that you can also describe a Turing machine more abstractly, like writing pseudocode.

The *Church–Turing thesis* states:

> Every effective method of computation is either equivalent to or weaker than a Turing machine.

R and RE

A language L is *Turing-recognizable* if there is a TM M such that

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

This is a *weak* notion of solving a problem:

If $w \in L$, then M accepts w.

If $w \notin L$, then M does not accept w. It might reject – or it might loop forever!

The class **RE** consists of all Turing-recognizable languages.

R and RE

A language L is *Turing-decidable* if there is a TM M such that

$$\forall w \in \Sigma^* . \ (M \text{ accepts } w \Leftrightarrow w \in L) \text{ and}$$
$$(M \text{ halts on all inputs}).$$

This is a *strong* notion of solving a problem:

If $w \in L$, then M accepts w.

If $w \notin L$, then M rejects w.

The class **R** consists of all Turing-decidable languages.

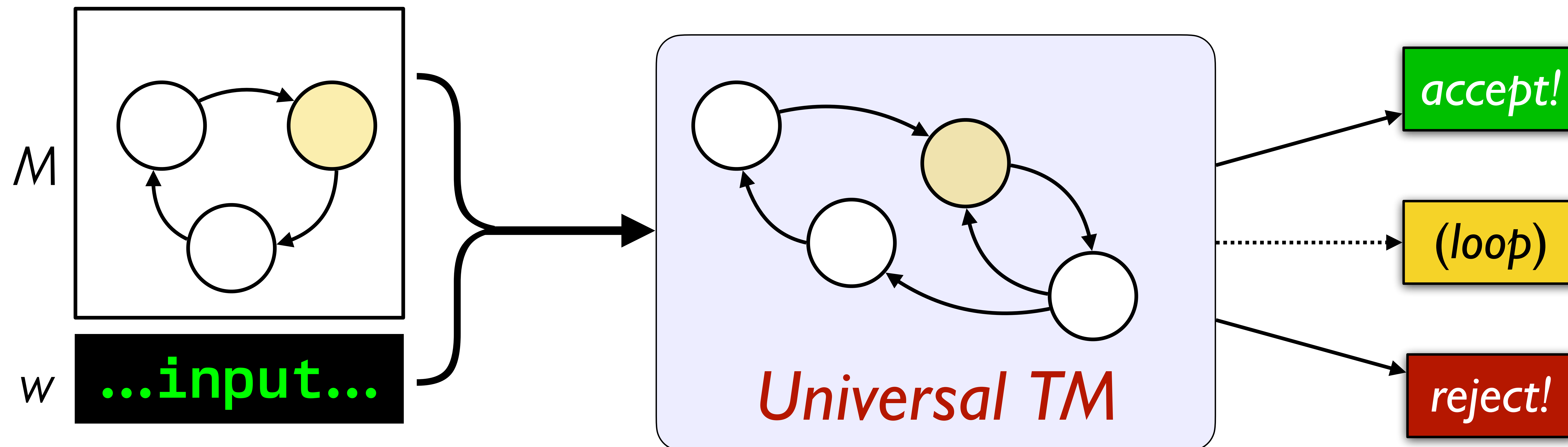
If *Obj* is an object, then $\langle Obj \rangle$ denotes some string representing *Obj*, similar to how it might be stored on disk or in memory on a real computer.

We can encode multiple objects as a single string. For example, if M is a TM and w is a string, then $\langle M, w \rangle$ is a string representing the pair of M and w.

There is a TM named U that is a *universal Turing machine*.

U takes as input a pair $\langle M, w\rangle$, where M is a TM and w is a string.

U does to $\langle M, w\rangle$ whatever M does to w.



The language of U is called A_{TM}:

$$A_{\mathrm{TM}} = L(U)$$
$$= \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

A_{TM} is the *acceptance language for Turing machines*.

Because there is a Turing machine, U, that recognizes A_{TM}, we know $A_{\mathrm{TM}} \in \mathbf{RE}$.

Self-referential programs

CLAIM Any program can be augmented to include a function called `my_source()` that returns a string representation of its source code

THEOREM It is possible to build Turing machines that get their own encodings and perform arbitrary computations on them.

```python
def zaira():
    me = my_source()
    print(me)
```

```python
def anastasia(input: str):
    me = my_source()
    return input == me
```

```python
def tamara():
    me = my_source()
    result = 0
    for c in me:
        if c == "a":
            result += 1
    return result
```

What do each of these programs do?

Part 1: *Self-defeating objects*

A *self-defeating object* is an object whose essential properties ensure it doesn’t exist.

Question: Why is there no largest integer?

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

THEOREM There is no largest integer.

PROOF SKETCH Suppose for the sake of contradiction that there is a largest integer. Call that integer n.

Consider the integer $n + 1$.

Notice that $n < n + 1$.

But then n isn't the largest integer.

Contradiction!

THEOREM There is no largest integer.

PROOF SKETCH Suppose for the sake of contradiction that there is a largest integer. Call that integer n.

Consider the integer $n + 1$.

Notice that $n < n + 1$.

*We're using **n** to construct something that undermines **n**, hence the term "self-defeating".*

But then n isn't the largest integer.

Contradiction!

A bad line of reasoning

CLAIM There is a largest integer.

PROOF SKETCH Assume x is the largest integer.

Notice that $x > x - 1$.

So there's no contradiction.

A bad line of reasoning

CLAIM There is a largest integer.

PROOF SKETCH Assume x is the largest integer.

Notice that $x > x - 1$.

So there's no contradiction.

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

A bad line of reasoning

CLAIM There is a largest integer.

PROOF SKETCH Assume x is the largest integer.

Notice that $x > x - 1$.

So there's no contradiction.

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

How do we know there's no contradiction? We only checked one case!

A bad line of reasoning

If you can show

$$x \textit{ exists} \Rightarrow \bot,$$

then you know that x doesn't exist. This is proof by contradiction.

$\bot$ is something you know to be false

But just because you can show

$$x \textit{ exists} \Rightarrow \top,$$

you cannot conclude from this that x exists. This is not a valid proof technique!

$\top$ is something you know to be true

Part 2: *The oracle*

An oracle appears who claims they can see into the future.

For a nominal fee, the oracle will tell you anything you want to know about the future.

One day a trickster arrives. The trickster wants to expose the oracle as a fraud.

The trickster says the following:

> *"I have a yes–no question about the future. But before I ask my question, let's talk payment.*
>
> *"If you answer 'yes', then I'll pay you $42.*
>
> *"If you answer 'no', then I'll pay you $240."*

The oracle thinks for a moment then agrees.

One day a trickster arrives. The trickster wants to expose the oracle as a fraud.

The trickster says the following:

> *"I have a yes–no question about the future. But before I ask my question, let's talk payment.*
>
> *"If you answer 'yes', then I'll pay you $42.*
>
> *"If you answer 'no', then I'll pay you $240."*

The oracle thinks for a moment then agrees.

Trickster pays $42
if the oracle answers "yes".

Trickster pays $240
if the oracle answers "no".

The trickster then asks this question:

“Am I going to pay you $240?”

Trickster pays $42
if the oracle answers “yes”.

Trickster pays $240
if the oracle answers “no”.

The payment scheme the oracle agreed to means

Oracle says yes ⇔ *Trickster pays $42*.

The trickster's question to the oracle means

Oracle says yes ⇔ *Trickster pays $240*.

Putting this together, we get

Trickster pays $240 ⇔ *Trickster pays $42*.

This is impossible!

The oracle is a self-defeating object.

The trickster’s strategy is to couple the oracle’s behavior to what the future holds.

The trickster’s behavior is chosen in advance to make the oracle’s answer wrong.

Therefore, the oracle can’t answer all questions about all people in the future.

"…a crocodile grabs a little boy playing on the banks of the Nile. The mother begs the crocodile to return her child. 'Certainly,' says the crocodile, 'if you can tell me in advance exactly what I'll do, I'll give you back your son; but if you guess wrong, I'll eat him for lunch." The mother, weeping desperately, calls out her prediction: 'you'll devour my baby.'

"Now the crocodile is in a bind: 'I can't give you back the child, because if I do, it means you've guessed wrong, and as I told you, if you guess wrong, I devour him.' But the mother shrewdly objects, 'It's not like that at all! Quite the opposite, you *can't* eat my baby because if you devour him, that will mean I guessed correctly. You promised that if I did that, you would return the child, and I know that as an honorable crocodile you will keep your word."

Umberto Eco, *On the Shoulders of Giants*, recounting a dilemma from Diogenes Laertius

Part 3: *Why do programs loop?*

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

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

Question: Why are infinite loops possible?

Or, rather: Are infinite loops an inherent part of computation or are they some weird sort of "accident" in how we program computers?

THEOREM The language A_{TM} is recognizable but undecidable.

There's a ***recognizer*** for A_{TM}, namely the universal Turing machine U.

But it's impossible to build a ***decider*** for this language.

Stated differently, there's a program we can write – a universal TM – that ***must*** loop infinitely on some inputs.

THEOREM The language A_{TM} is recognizable but undecidable.

There's a *recognizer* for A_{TM}, namely the universal Turing machine U.

But it's impossible to build a *decider* for this language.

Goal: Prove this theorem, and explore its theoretical and philosophical implications.

Stated differently, there's a program we can write – a universal TM – that *must* loop infinitely on some inputs.

Recall the language A_{TM} is

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

The universal TM U has the following behavior when given as input a TM M and a string w:

If M accepts w → U accepts $\langle M, w\rangle$.

If M rejects w → U rejects $\langle M, w\rangle$.

If M loops on w → U loops on $\langle M, w\rangle$.

U is a recognizer for A_{TM}, but, because of that last case, it's not a decider for A_{TM}.

Recall the language A_{TM} is

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

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

If M accepts w → D ? $\langle M, w\rangle$.

If M rejects w → D ? $\langle M, w\rangle$.

If M loops on w → D ? $\langle M, w\rangle$.

Recall the language A_{TM} is

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

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

If M accepts w → D accepts $\langle M, w\rangle$.

If M rejects w → D rejects $\langle M, w\rangle$.

If M loops on w → D rejects $\langle M, w\rangle$.

***Goal:** Prove that there is no way to build a program that meets these requirements.*

We can envision a decider for A_{TM} as a function

```
will_accept(fn: str, input: str) -> bool
```

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

It then does the following:

If **`fn(input)`** returns true, **`will_accept(fn, input)`** returns true.

If **`fn(input)`** returns false, **`will_accept(fn, input)`** returns false.

If **`fn(input)`** loops, then **`will_accept(fn, input)`** returns false.

We're going to show it's impossible to write a function that actually does this. But, for now, let's just explore what such a decider would do.

```
fn = """
def f(input: str) -> bool:
    if input == "":
        return False
    return input[0] == "a"
"""

input = "abbababba"

will_accept(fn, input) → ?
```

```
fn = """
def g(input: str) -> bool:
    while True:
        input += input
"""

input = "yay! "

will_accept(fn, input) → ?
```

```
fn = """
def h(input: str) -> bool:
    for c in input:
        if c != input[0]:
            return True
    return False
"""

input = "aaaaaa"

will_accept(fn, input) → ?
```

```
fn = """
def j(input: str) -> bool:
   n = len(input)
   while n > 1:
       if n % 2 == 0: n /= 2
       else: n = 3 * n + 1
   return True
"""

input = ... # 10^137 "a"s

will_accept(fn, input) → ?
```

Why is A_{TM} hard?

A decider for A_{TM} would need to be able to determine not just what *simple* programs will do but also *resolve open problems in mathematics*, like determining whether the hailstone sequence terminates for any input – and much, much more.

In other words, this seemingly simple problem of "is this program going to accept?" accidentally scoops up other, seemingly harder, problems.

Part 4: *Putting it all together*

To recap,

We’re assuming that, somehow, someone wrote a function

```
will_accept(fn: str, input: str) -> bool
```

that takes the code of a function and an input to that function, then

returns true if `fn(input)` returns true, and

returns false if `fn(input)` doesn’t return true.

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

Idea: Convert the oracle story into a program.

Am I going to pay you $240?

Trickster pays $42 if the oracle answers "yes".

Trickster pays $240 if the oracle answers "no".

```python
def will_accept(fn: str, w: str) -> bool:
    """Returns True if fn(w) returns true.
    Otherwise, returns False."""
    ...


def trickster(input: str) -> bool:
    me = my_source()
    if will_accept(me, input):
        return False
    else:
        return True
```

```python
def will_accept(fn: str, w: str) -> bool:
    """Returns True if fn(w) returns true.
    Otherwise, returns False."""
    ...


def trickster(input: str) -> bool:
    me = my_source()
    if will_accept(me, input):
        return False
    else:
        return True
```

What happens if `will_accept` *says the* `trickster` *program* ***accepts*** *its input?*

```
def will_accept(fn: str, w: str) -> bool:
    """Returns True if fn(w) returns true.
    Otherwise, returns False."""
    ...


def trickster(input: str) -> bool:
    me = my_source()
    if will_accept(me, input):
        return False
    else:
        return True
```

What happens if `will_accept` *says the* `trickster` *program **accepts** its input?*

```
def will_accept(fn: str, w: str) -> bool:
    """Returns True if fn(w) returns true.
    Otherwise, returns False."""
    ...


def trickster(input: str) -> bool:
    me = my_source()
    if will_accept(me, input):
        return False
    else:
        return True
```

What happens if `will_accept` *says the* `trickster` *program* ***accepts*** *its input?*

```
def will_accept(fn: str, w: str) -> bool:
    """Returns True if fn(w) returns true.
    Otherwise, returns False."""
    ...


def trickster(input: str) -> bool:
    me = my_source()
    if will_accept(me, input):
        return False
    else:
        return True
```

*What happens if `will_accept` says the `trickster` program **accepts** its input?*

```
def will_accept(fn: str, w: str) -> bool:
    """Returns True if fn(w) returns true.
    Otherwise, returns False."""
    ...


def trickster(input: str) -> bool:
    me = my_source()
    if will_accept(me, input):
        return False
    else:
        return True
```

What happens if `will_accept` *says the* `trickster` *program* ***accepts*** *its input?*

`trickster` ***rejects*** *the input!*

```
def will_accept(fn: str, w: str) -> bool:
    """Returns True if fn(w) returns true.
    Otherwise, returns False."""
    ...


def trickster(input: str) -> bool:
    me = my_source()
    if will_accept(me, input):
        return False
    else:
        return True
```

What happens if `will_accept` *says the* `trickster` *program* ***rejects*** *its input?*

```python
def will_accept(fn: str, w: str) -> bool:
    """Returns True if fn(w) returns true.
    Otherwise, returns False."""
    ...


def trickster(input: str) -> bool:
    me = my_source()
    if will_accept(me, input):
        return False
    else:
        return True
```

*What happens if `will_accept` says the `trickster` program **rejects** its input?*

```python
def will_accept(fn: str, w: str) -> bool:
    """Returns True if fn(w) returns true.
    Otherwise, returns False."""
    ...


def trickster(input: str) -> bool:
    me = my_source()
    if will_accept(me, input):
        return False
    else:
        return True
```

What happens if `will_accept` *says the* `trickster` *program* ***rejects*** *its input?*

```python
def will_accept(fn: str, w: str) -> bool:
    """Returns True if fn(w) returns true.
    Otherwise, returns False."""
    ...


def trickster(input: str) -> bool:
    me = my_source()
    if will_accept(me, input):
        return False
    else:
        return True
```

What happens if `will_accept` *says the* `trickster` *program* ***rejects*** *its input?*

```
def will_accept(fn: str, w: str) -> bool:
    """Returns True if fn(w) returns true.
    Otherwise, returns False."""
    ...


def trickster(input: str) -> bool:
    me = my_source()
    if will_accept(me, input):
        return False
    else:
        return True
```

What happens if `will_accept` *says the* `trickster` *program* ***rejects*** *its input?*

```
def will_accept(fn: str, w: str) -> bool:
    """Returns True if fn(w) returns true.
    Otherwise, returns False."""
    ...


def trickster(input: str) -> bool:
    me = my_source()
    if will_accept(me, input):
        return False
    else:
        return True
```

What happens if **`will_accept`** *says the* **`trickster`** *program* ***rejects*** *its input?*

`trickster` ***accepts*** *the input!*

```python
def will_accept(fn: str, w: str) -> bool:
    """Returns True if fn(w) returns true.
    Otherwise, returns False."""
    ...


def trickster(input: str) -> bool:
    me = my_source()
    if will_accept(me, input):
        return False
    else:
        return True
```

A self-defeating object

Using that object against itself

```python
def will_accept(fn: str, w: str) -> bool:
    """Returns True if fn(w) returns true.
    Otherwise, returns False."""
    ...


def trickster(input: str) -> bool:
    me = my_source()
    if will_accept(me, input):
        return False
    else:
        return True
```

"The largest integer n"

"The integer n + 1"

THEOREM There is no largest integer.

PROOF SKETCH Suppose for the sake of contradiction that there is a largest integer. Call that integer n. Consider the integer $n + 1$. Notice that $n < n + 1$. But then n isn't the largest integer. Contradiction!

THEOREM $A_{\text{TM}} \notin \mathbf{R}$.

THEOREM $A_{\mathrm{TM}} \notin \mathbf{R}$.

PROOF By contradiction; assume that $A_{\mathrm{TM}} \in \mathbf{R}$.

THEOREM $A_{\text{TM}} \notin \mathbf{R}$.

PROOF By contradiction; assume that $A_{\text{TM}} \in \mathbf{R}$.

Then there is some decider D for A_{TM}.

THEOREM $A_{\text{TM}} \notin \mathbf{R}$.

PROOF By contradiction; assume that $A_{\text{TM}} \in \mathbf{R}$. Then there is some decider D for A_{TM}. We can represent D as a function

```
will_accept(fn: str, w: str) -> bool
```

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

THEOREM $A_{TM} \notin \mathbf{R}$.

PROOF By contradiction; assume that $A_{TM} \in \mathbf{R}$.

Then there is some decider D for A_{TM}. We can represent D as a function

```
will_accept(fn: str, w: str) -> bool
```

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

Given this, consider this function:

```
def trickster(input: str) -> bool:
    me = my_source()
    if will_accept(me, input):
        return False
    else:
        return True
```

THEOREM $A_{TM} \notin \mathbf{R}$.

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

```
will_accept(fn: str, w: str) -> bool
```

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

Given this, consider this function:

```
def trickster(input: str) -> bool:
    me = my_source()
    if will_accept(me, input):
        return False
    else:
        return True
```

Since `will_accept` decides A_{TM} and `me` holds the source of `trickster`, we know that

`will_accept(me, input)` returns true
if and only if
`trickster(input)` returns true.

THEOREM $A_{TM} \notin \mathbf{R}$.

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

```
will_accept(fn: str, w: str) -> bool
```

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

Given this, consider this function:

```
def trickster(input: str) -> bool:
    me = my_source()
    if will_accept(me, input):
        return False
    else:
        return True
```

Since `will_accept` decides A_{TM} and `me` holds the source of `trickster`, we know that

`will_accept(me, input)` returns true
if and only if
`trickster(input)` returns true.

Given how `trickster` is written, we see that

`will_accept(me, input)` returns true
if and only if
`trickster(input)` doesn't return true.

THEOREM $A_{TM} \notin \mathbf{R}$.

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

```
will_accept(fn: str, w: str) -> bool
```

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

Given this, consider this function:

```
def trickster(input: str) -> bool:
    me = my_source()
    if will_accept(me, input):
        return False
    else:
        return True
```

Since `will_accept` decides A_{TM} and `me` holds the source of `trickster`, we know that

`will_accept(me, input)` returns true
if and only if
`trickster(input)` returns true.

Given how `trickster` is written, we see that

`will_accept(me, input)` returns true
if and only if
`trickster(input)` doesn't return true.

This means that

`trickster(input)` returns true
if and only if
`trickster(input)` doesn't return true.

THEOREM $A_{TM} \notin \mathbf{R}$.

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

```
will_accept(fn: str, w: str) -> bool
```

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

Given this, consider this function:

```
def trickster(input: str) -> bool:
    me = my_source()
    if will_accept(me, input):
        return False
    else:
        return True
```

Since `will_accept` decides A_{TM} and `me` holds the source of `trickster`, we know that

`will_accept(me, input)` returns true
if and only if
`trickster(input)` returns true.

Given how `trickster` is written, we see that

`will_accept(me, input)` returns true
if and only if
`trickster(input)` doesn't return true.

This means that

`trickster(input)` returns true
if and only if
`trickster(input)` doesn't return true.

We've reached a contradiction, so our assumption was wrong and A_{TM} is undecidable. ■

THEOREM $A_{TM} \notin \mathbf{R}$.

PROOF By contradiction; assume that $A_{TM} \in \mathbf{R}$. Then there is some decider D for A_{TM}. If D is given a TM–string pair, it will determine whether the TM accepts the string and report back the answer.

Given this, consider this TM:

M = "On input w:

1. Have M obtain its own description $\langle M \rangle$.
2. Run D on $\langle M, w \rangle$ and see what it says.
3. If D says that M will accept w, reject.
4. If D says that M will not accept w, accept."

The same proof, without using the equivalence of TMs and code

Since D decides A_{TM}, we know that

D accepts $\langle M, w \rangle$
if and only if
M accepts w.

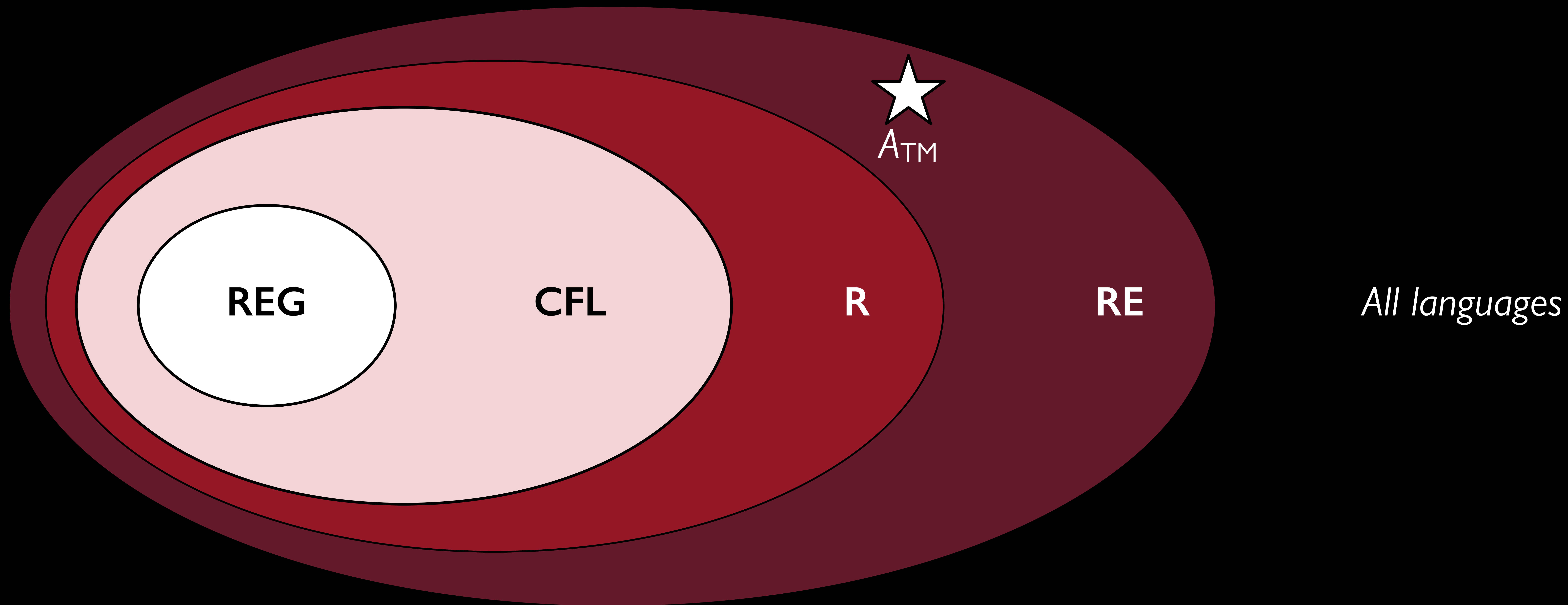
Given how M is written, we see that

D accepts $\langle M, w \rangle$
if and only if
M doesn't accept w.

This means that

M accepts w
if and only if
M doesn't accept w.

We've reached a contradiction, so our assumption was wrong and A_{TM} is undecidable. ■


A_TM
REG
CFL
R
RE
All languages

What does this mean?

In one fell swoop, we've proven that

A_{TM} is ***undecidable***; there is no general algorithm that can determine whether a Turing machine will accept a string.

R ≠ **RE**, because $A_{TM} \notin$ **R** but $A_{TM} \in$ **RE**.

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

What exactly does it mean for A_{TM} to be undecidable?

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

This means that it's provably impossible for computers to be able to answer most questions about what a program will do.

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

At a more fundamental level, the existence of undecidable problems tells us the following:

There is a difference between what is true and what we can discover is true.

Given a TM M and a string w, either

M accepts w or *M does not accept w*

But since A_{TM} is undecidable, there's no algorithm that can always determine *which* of these statements is true!

R ≠ RE

Because **R ≠ RE**, there is a difference between decidability and recognizability:

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

There are problems where, when the answer is "yes", you can confirm it, but where if you don't have the answer, you can't come up with it in a mechanical way.

Part 5: *More impossibility results*

The Halting Problem

Given a Turing machine M and a string w, will M halt when run on w?

As a formal language, this problem would be

$HALT_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM that halts on } w\}$

THEOREM $HALT_{TM}$ is recognizable, but undecidable.

There's a recognizer for $HALT_{TM}$.

There is no decider for $HALT_{TM}$.

CLAIM $HALT_{TM} \in \mathbf{RE}$.

IDEA If you were certain that a Turing machine M halted on a string w, could you convince me of that?

Yes – just run M on w and see what happens:

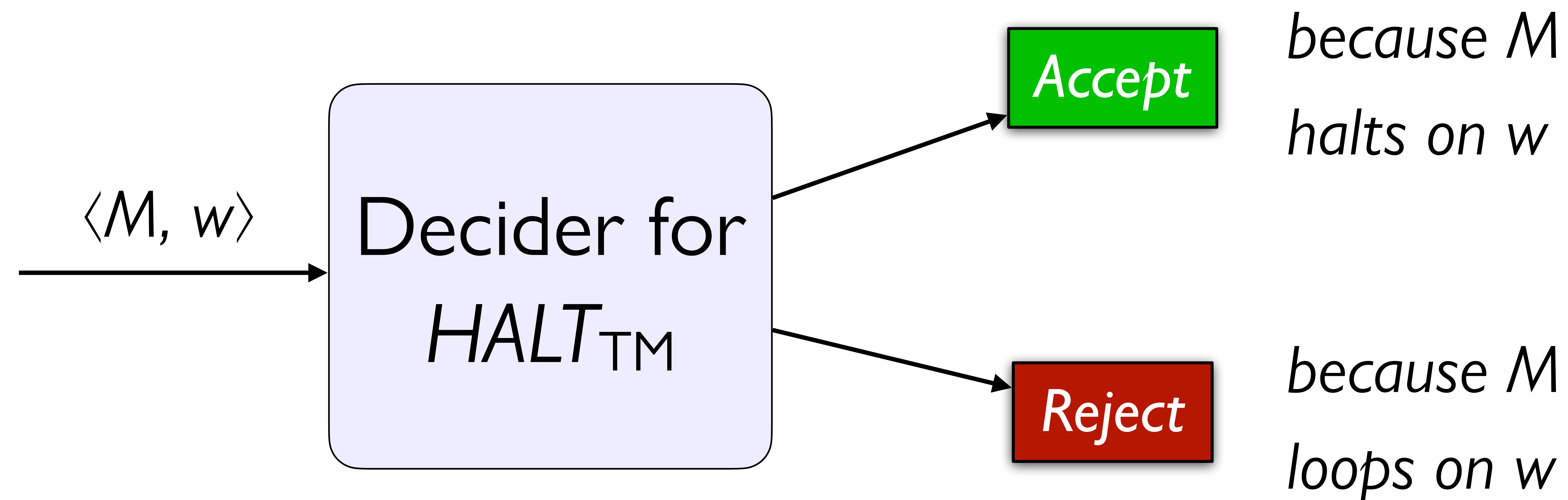
```
def will_halt(M: str, w: str) -> bool:
    set up a simulation of M running on w
    while True:
        if M returned True:
            return True
        elif M returned False:
            return True
        else:
            simulate one more step of M running on w
```

THEOREM The Halting Problem is undecidable.

A decider for $HALT_{TM}$

Suppose we managed to build a decider for

$HALT_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM that halts on } w\}$:



We could represent this in software as a function

```
will_halt(fn: str, input: str) -> bool
```

```python
def will_halt(fn: str, w: str) -> bool:
    """Returns True if fn(w) halts.
    Otherwise, returns False."""
    ...


def trickster(input: str) -> bool:
    me = my_source()
    if will_halt(me, input):
        # Infinite loop
        while True:
            pass
    else:
        return True
```

```
def will_halt(fn: str, w: str) -> bool:
    """Returns True if fn(w) halts.
    Otherwise, returns False."""
    ...


def trickster(input: str) -> bool:
    me = my_source()
    if will_halt(me, input):
        # Infinite loop
        while True:
            pass
    else:
        return True
```

`trickster(input)` *halts*

⇔

`will_halt(me, input)` *returns True*

⇔

`trickster(input)` *loops*

THEOREM $HALT_{TM} \notin \mathbf{R}$.

PROOF By contradiction; assume that $HALT_{TM} \in \mathbf{R}$. Then there is some decider D for $HALT_{TM}$. We can represent D as a function

```
will_halt(fn: str, w: str) -> bool
```

that takes as input the source code of a function `fn` and a string `w`, and returns true if `fn(w)` halts and returns false otherwise.

Given this, consider this function:

```
def trickster(input: str) -> bool:
    me = my_source()
    if will_halt(me, input):
        while True:
            pass
    else:
        return True
```

Since `will_halt` decides $HALT_{TM}$ and `me` holds the source of `trickster`, we know that

`will_halt(me, input)` returns true
if and only if
`trickster(input)` halts.

Given how `trickster` is written, we see that

`will_halt(me, input)` returns true
if and only if
`trickster(input)` loops.

This means that

`trickster(input)` halts
if and only if
`trickster(input)` loops.

This is impossible. We've reached a contradiction, so our assumption was wrong and $HALT_{TM} \notin \mathbf{R}$. ■

THEOREM $HALT_{TM} \notin \mathbf{R}$.

PROOF By contradiction; assume that $HALT_{TM} \in \mathbf{R}$. Then there is some decider D for $HALT_{TM}$. If this machine is given any TM–string pair, it will then determine whether the TM halts on the string and report back the answer.

Given this, we could construct the following TM:

M = "On input w:

1. Have M obtain its own description $\langle M \rangle$.
2. Run D on $\langle M, w \rangle$ and see what it says.
3. If D says that M halts on w, go into an infinite loop.
4. If D says that M loops on w, accept."

Since D decides $HALT_{TM}$,

$D(\langle M, w \rangle)$ accepts if and only if $M(w)$ halts.

Given how M is written, we see that

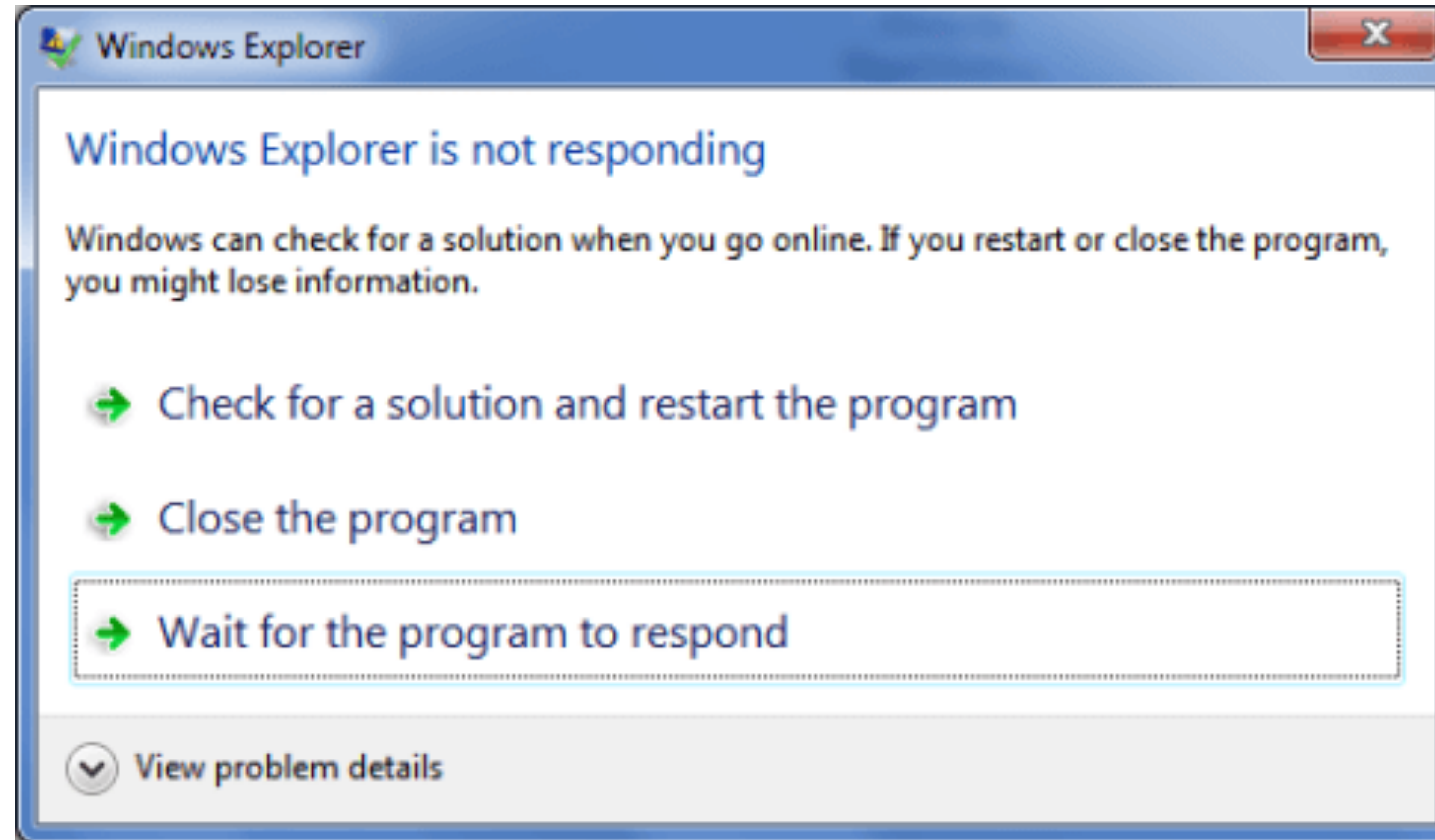
$D(\langle M, w \rangle)$ accepts if and only if $M(w)$ loops.

This means that

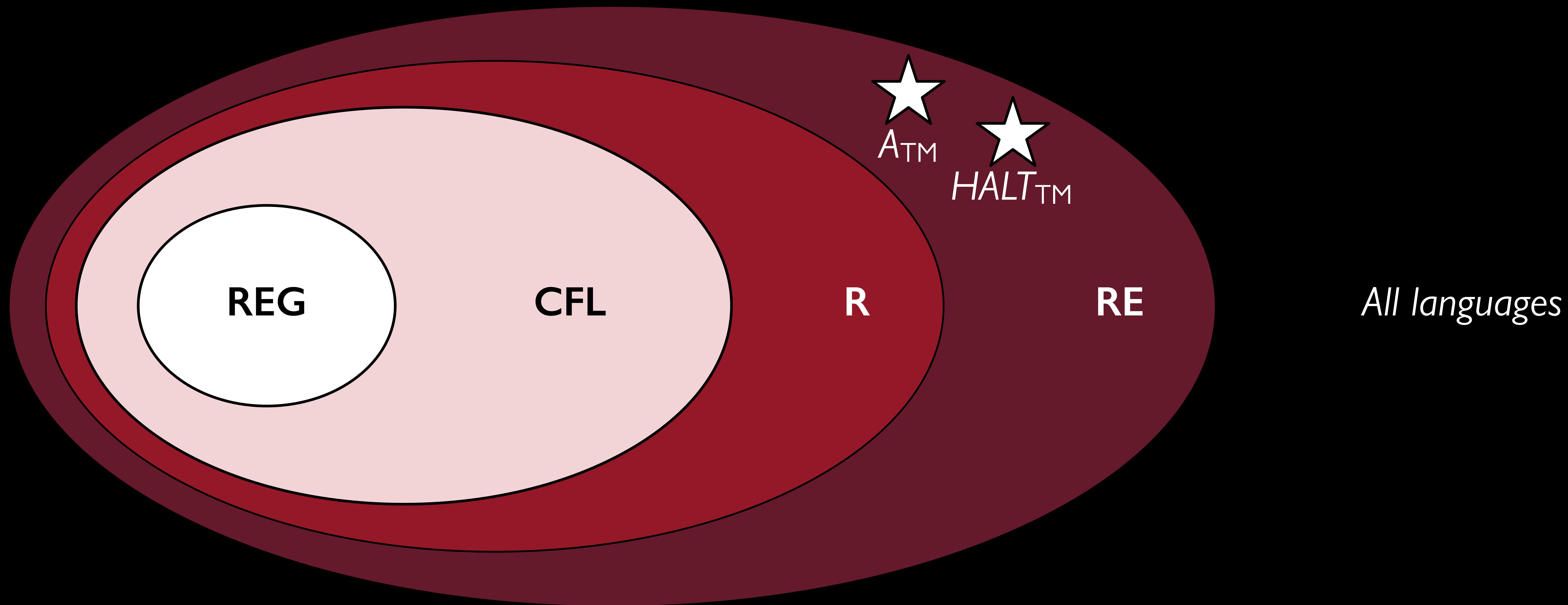
$M(w)$ halts if and only if $M(w)$ loops.

This is impossible. We've reached a contradiction, so our assumption was wrong and $HALT_{TM} \notin \mathbf{R}$. ■

The same proof, without using the equivalence of TMs and code



*Moral: This isn't **necessarily** Microsoft's fault.*


REG
CFL
R
RE
All languages
A_{TM}
$HALT_{TM}$

Ramifications – or, so what?

These problems might not seem all that exciting, so who cares if we can't solve them?

It turns out this same line of reasoning can be used to show many important, practical problems are impossible to solve.

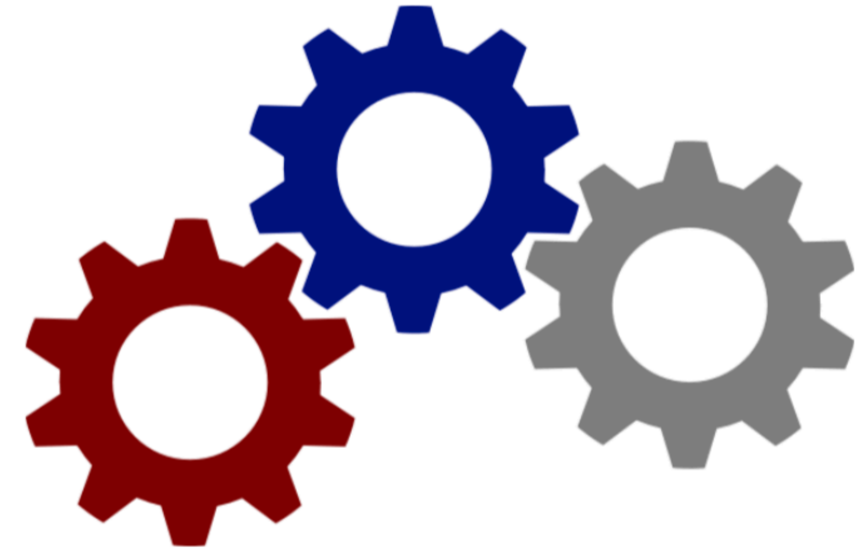
Analogy time!

Engineering problem

Design a diesel engine that doesn't emit lots of NO_x pollutants.

Engineering problem

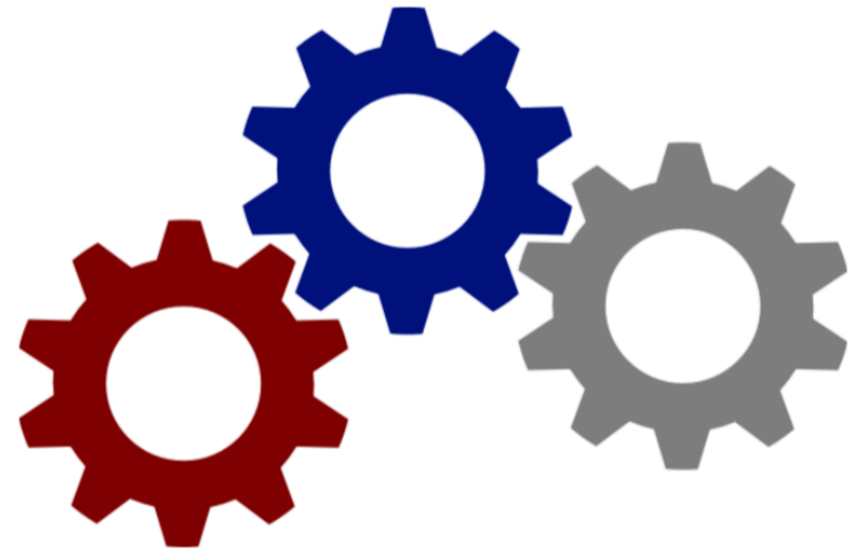
Design a diesel engine that doesn't emit lots of NO_x pollutants.



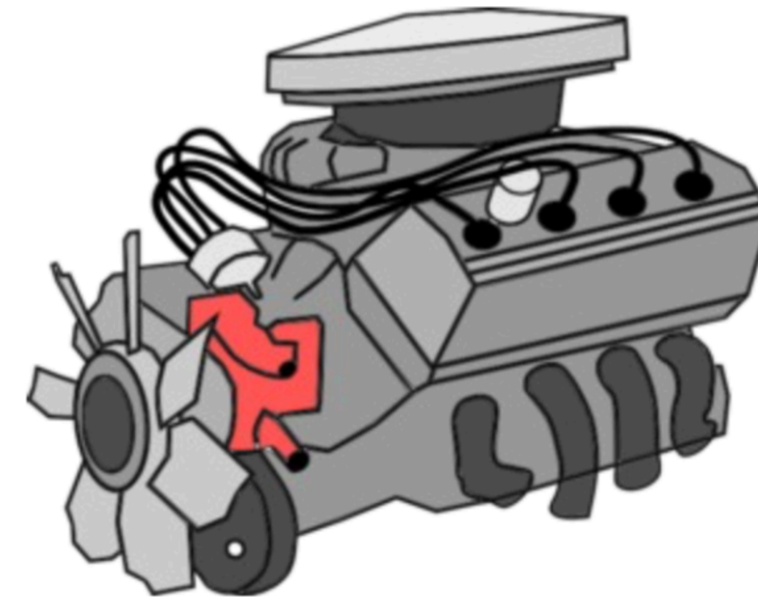
Engineering prowess!

Engineering problem

Design a diesel engine that doesn't emit lots of NO_x pollutants.



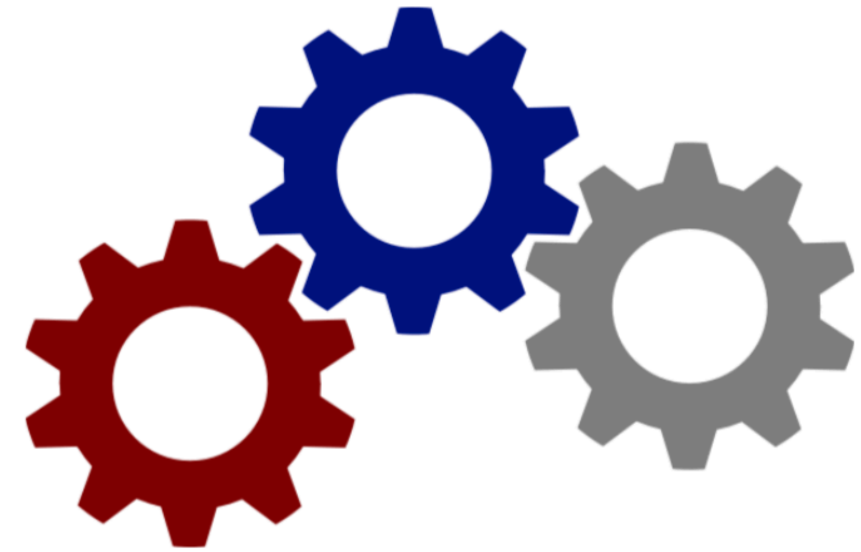
Engineering prowess!



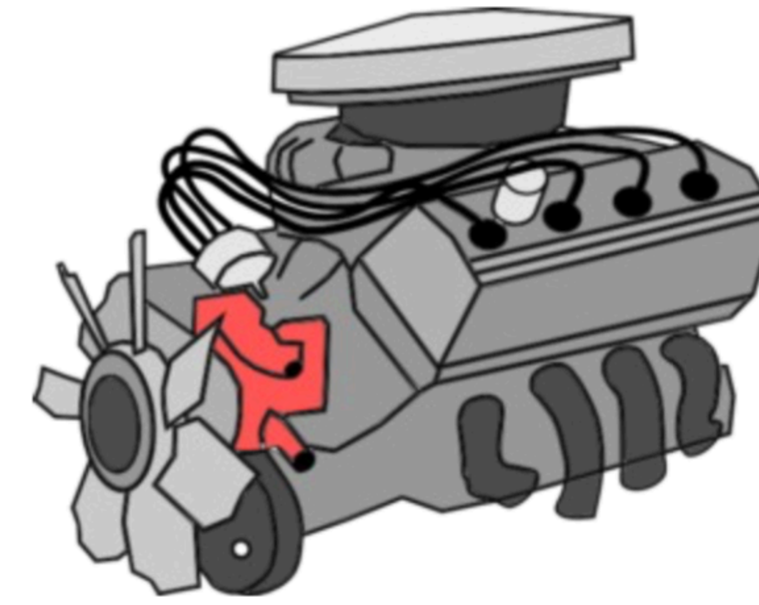
Awesome engine!

Engineering problem

Design a diesel engine that doesn't emit lots of NO_x pollutants.



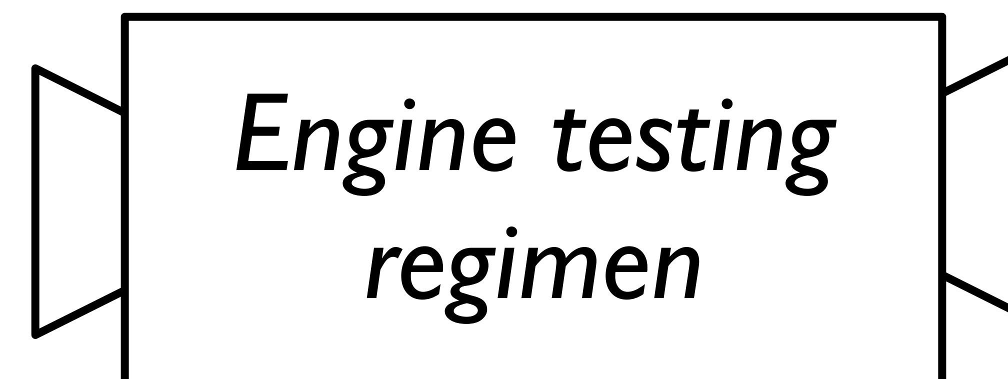
Engineering prowess!



Awesome engine!

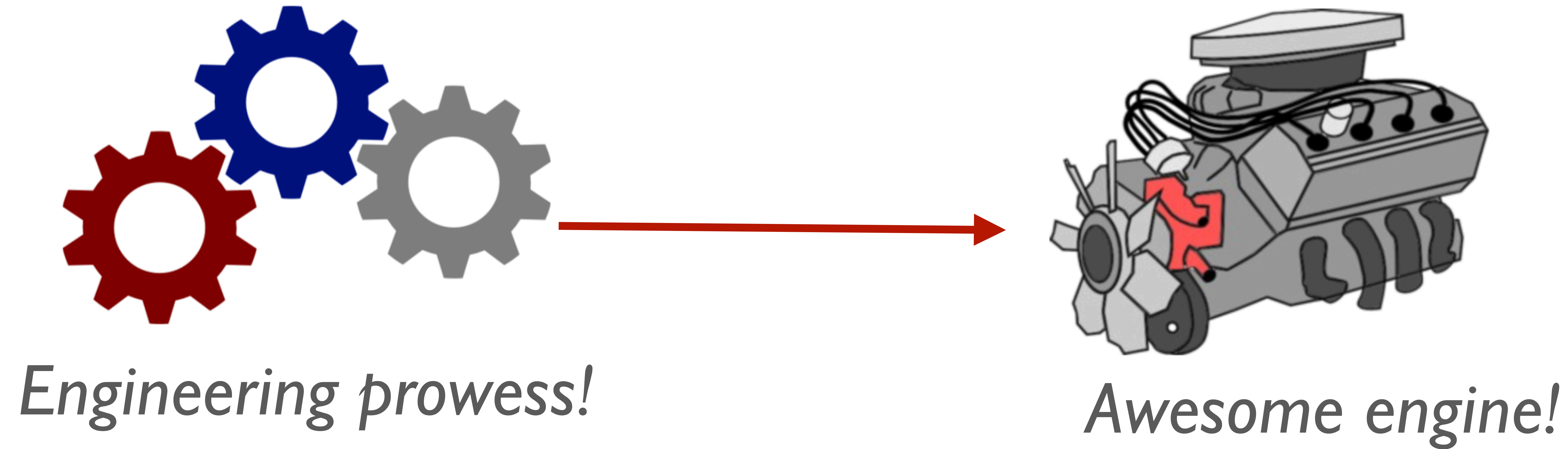
Regulatory problem

Design a testing procedure that, given a diesel engine, determines whether it emits lots of NO_x pollutants.



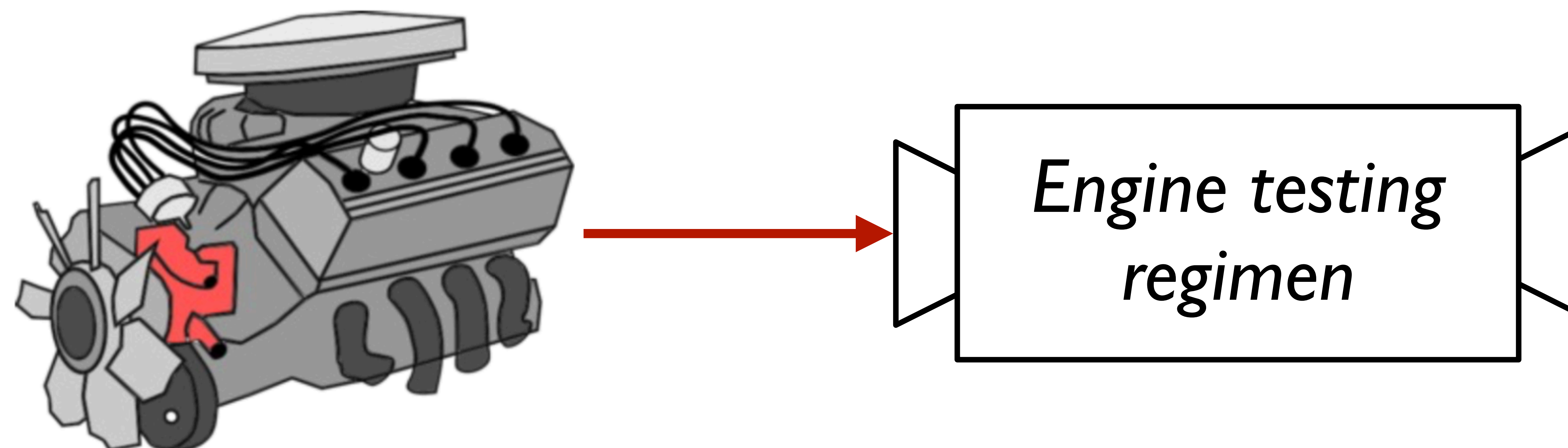
Engineering problem

Design a diesel engine that doesn't emit lots of NO_x pollutants.



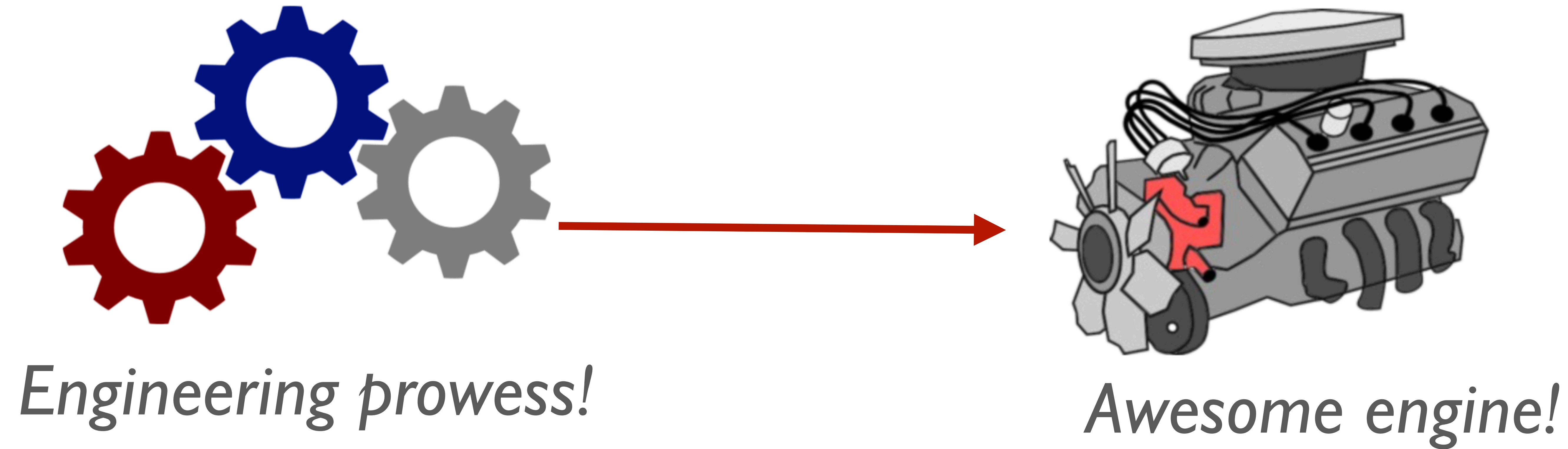
Regulatory problem

Design a testing procedure that, given a diesel engine, determines whether it emits lots of NO_x pollutants.



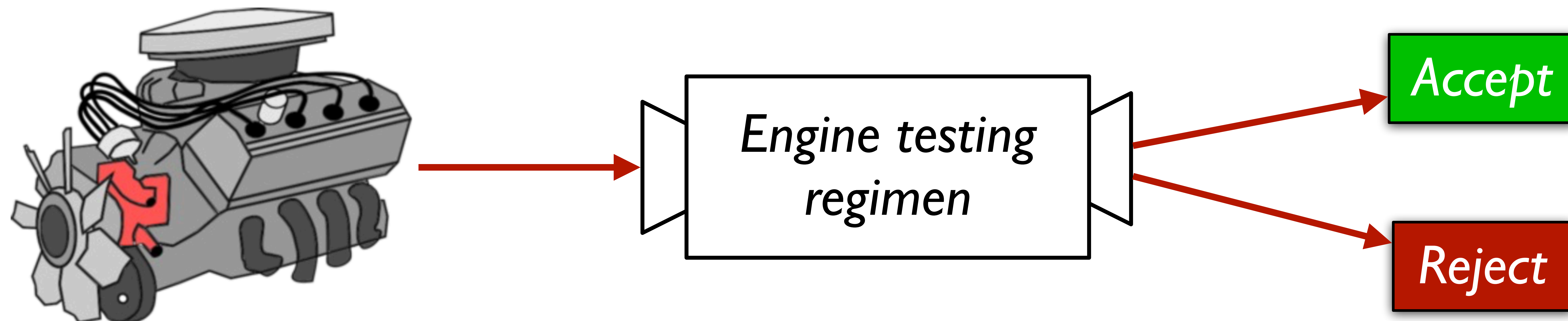
Engineering problem

Design a diesel engine that doesn't emit lots of NO_x pollutants.



Regulatory problem

Design a testing procedure that, given a diesel engine, determines whether it emits lots of NO_x pollutants.



Almost all "regulatory problems" about computer programs are undecidable.

That is, almost all problems of the form "does this program have [behavioral property *X*]" are undecidable.

This can be formalized as a result called Rice's Theorem.

Secure voting

Suppose you want to make a voting machine for use in an imaginary election between two parties.

Let Σ = {`r`, `d`}, for no particular reason.

A string consists of a series of votes for the candidates.

For example, `rrdddrd` means “two people voted for `r`, then three people voted for `d`, then one more person voted for `r`, then one more person voted for `d`”.

A voting machine is a program that takes as input a string of `r`s and `d`s and then reports whether person `r` won the election.

It would be equivalent to ask "did `d` win"?

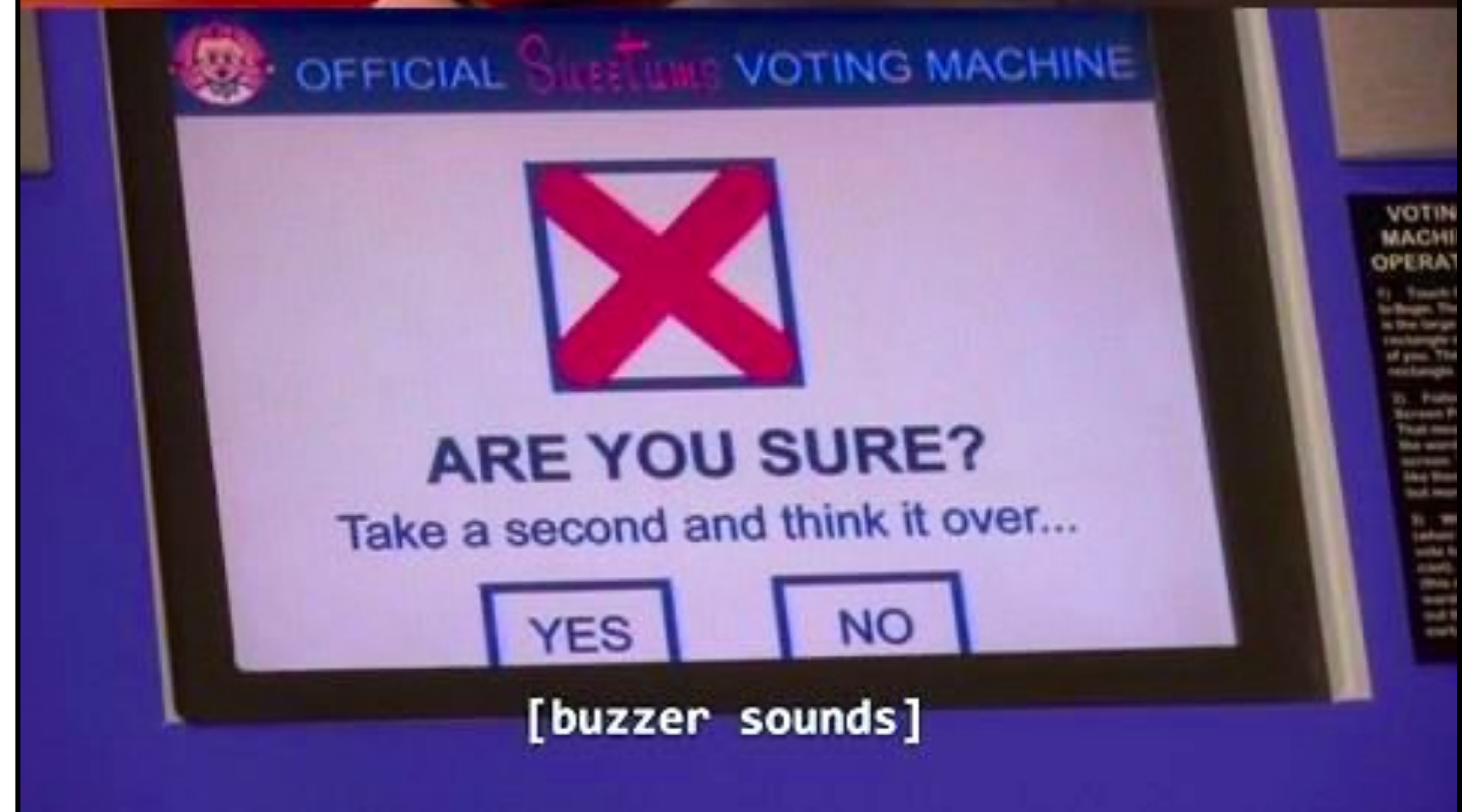
For simplicity, this model assumes centralized voting, e.g., done online.

Question: Given a Turing machine that someone claims is a secure voting machine, could we automatically check whether it's really a secure voting machine?

WATCH WHAT HAPPENS
WHEN I VOTE FOR BOBBY NEWPORT.

OFFICIAL Sweetums VOTING MACHINE
GOOD CHOICE!
Enjoy a voucher for a complimentary Sweetums candy bar.
VOTING MACHINE OPERATION

BUT WATCH WHAT HAPPENS
WHEN YOU VOTE FOR ME.


OFFICIAL Sweetums VOTING MACHINE
ARE YOU SURE?
Take a second and think it over...
YES
NO
[buzzer sounds]

```python
def accuracy(input: str) -> bool:
    r_votes = count_rs(input)
    d_votes = count_ds(input)
    if r_votes > d_votes:
        return True  # R won
    else:
        return False  # R lost
```

A (simple) secure voting machine

```python
def burn(input: str) -> bool:
    if input[0] == "r":
        return True  # R won
    else:
        return False  # R lost
```

A (simple) insecure voting machine

```python
def disintegration(input: str) -> bool:
    r_votes = count_rs(input)
    d_votes = count_ds(input)
    if r_votes = d_votes:
        # Tied; R lost.
        return False
    if r_votes < d_votes:
        # D won; R lost.
        return False
    else:
        # R won
        return True
```

An (evil) insecure voting machine

```python
def disintegration(input: str) -> bool:
    r_votes = count_rs(input)
    d_votes = count_ds(input)
    if r_votes = d_votes:
        # Tied; R lost.
        return False
    if r_votes < d_votes:
        # D won; R lost.
        return False
    else:
        # R won
        return True
```

This is assignment; == is equality testing. Python won't actually let you shoot yourself in the foot like this, but other languages (like C) will, and it's a common bug!

An (evil) insecure voting machine

```python
def faith(input: str) -> bool:
    n = len(input)
    while n > 1:
        if n % 2 == 0:
            n /= 2
        else:
            n = 3 * n + 1
    r_votes = count_rs(input)
    d_votes = count_ds(input)
    if r_votes > d_votes:
        return True  # R won
    else:
        return False  # R lost
```

```python
def faith(input: str) -> bool:
    n = len(input)
    while n > 1:
        if n % 2 == 0:
            n /= 2
        else:
            n = 3 * n + 1
    r_votes = count_rs(input)
    d_votes = count_ds(input)
    if r_votes > d_votes:
        return True  # R won
    else:
        return False  # R lost
```

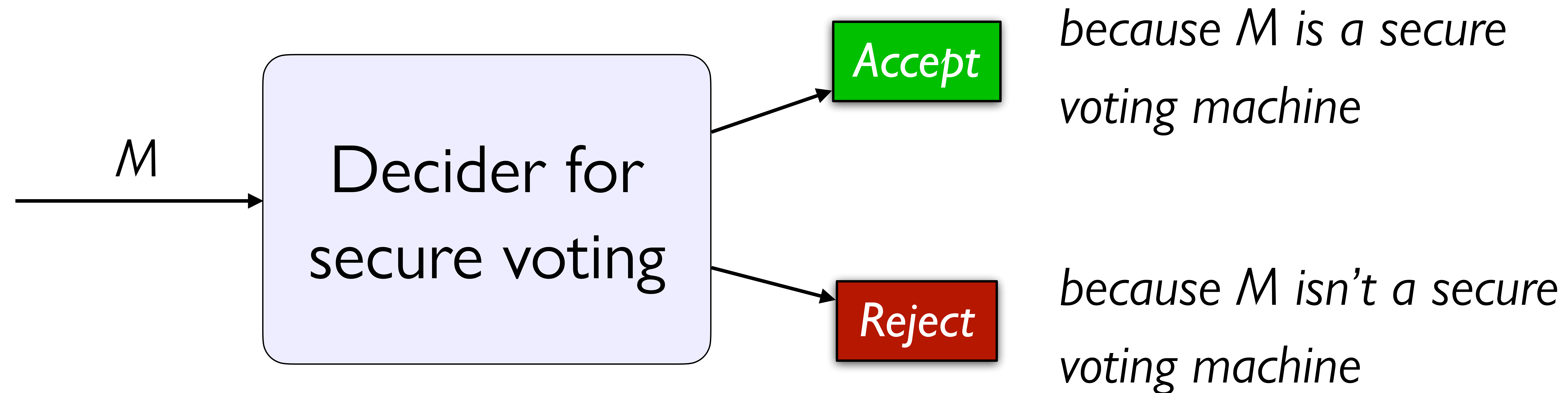
No one knows!

A voting machine is a program that takes as input a string of `r`s and `d`s and then reports whether person `r` won the election.

- It would be equivalent to ask "did `d` win"?
- For simplicity, this model assumes centralized voting, e.g., done online.

Question: Given a Turing machine that someone claims is a secure voting machine, could we automatically check whether it's really a secure voting machine?

Suppose that, somehow, we managed to build a decider for the secure voting problem:



We could represent this in software as a function

```
is_secure_vm(fn: str) -> bool
```

```python
def is_secure_vm(fn: str) -> bool:
    """Return True if fn accepts only
    strings with more "r"s than "d"s."""
    ...


def trickster(input: str) -> bool:
    me = my_source()
    answer = count_rs(input) > count_ds(input)
    if is_secure_vm(me):
        return not answer
    else:
        return answer
```

```python
def is_secure_vm(fn: str) -> bool:
    """Return True if fn accepts only
    strings with more "r"s than "d"s."""
    ...


def trickster(input: str) -> bool:
    me = my_source()
    answer = count_rs(input) > count_ds(input)
    if is_secure_vm(me):
        return not answer
    else:
        return answer
```

`trickster` *is a secure voting machine*

⇔

`is_secure_vm(me)` *returns True*

⇔

`trickster` *isn't a secure voting machine*

THEOREM The secure voting problem is undecidable.

PROOF By contradiction; assume the secure voting problem is decidable. Then there is some decider *D* for the secure voting problem, which we can represent in software as a function

```
is_secure_vm(fn: str) -> bool
```

that takes as input the source code of a function `fn` and returns true if the program is a secure voting machine (that is, accepts precisely the strings with more `r`s than `d`s) and false otherwise.

Given this, we could construct this program:

```
def trickster(input: str) -> bool:
    me = my_source()
    ans = count_rs(input) > count_ds(input)
    if is_secure_vm(me):
        return not ans
    else:
        return ans
```

Since `is_secure_vm` decides the secure voting problem and `me` holds the source of `trickster`, we know that

`is_secure_vm(me)` returns true
if and only if
`trickster` is a secure voting machine.

Given how `trickster` is written, we see that

`is_secure_vm(me)` returns true
if and only if
`trickster` isn't a secure voting machine.

This means that

`trickster` is a secure voting machine
if and only if
`trickster` isn't a secure voting machine.

This is impossible. We've reached a contradiction, so our assumption was incorrect and the secure voting problem is undecidable. ■

Interpreting this result

This tells us that there is *no* general algorithm that we can follow to determine whether a program is a secure voting machine.

In other words, any general algorithm to check voting machines will always be wrong on *at least one input*.

The previous example might seem contrived, but it's not. This is a problem we really would like to be able to solve – but it's provably impossible!

What can we do?

Design algorithms that work in *many* but not *all* cases.

This is often done in practice!

Fall back on human verification of voting machines.

We do this too!

Carry a healthy degree of skepticism about electronic voting machines.

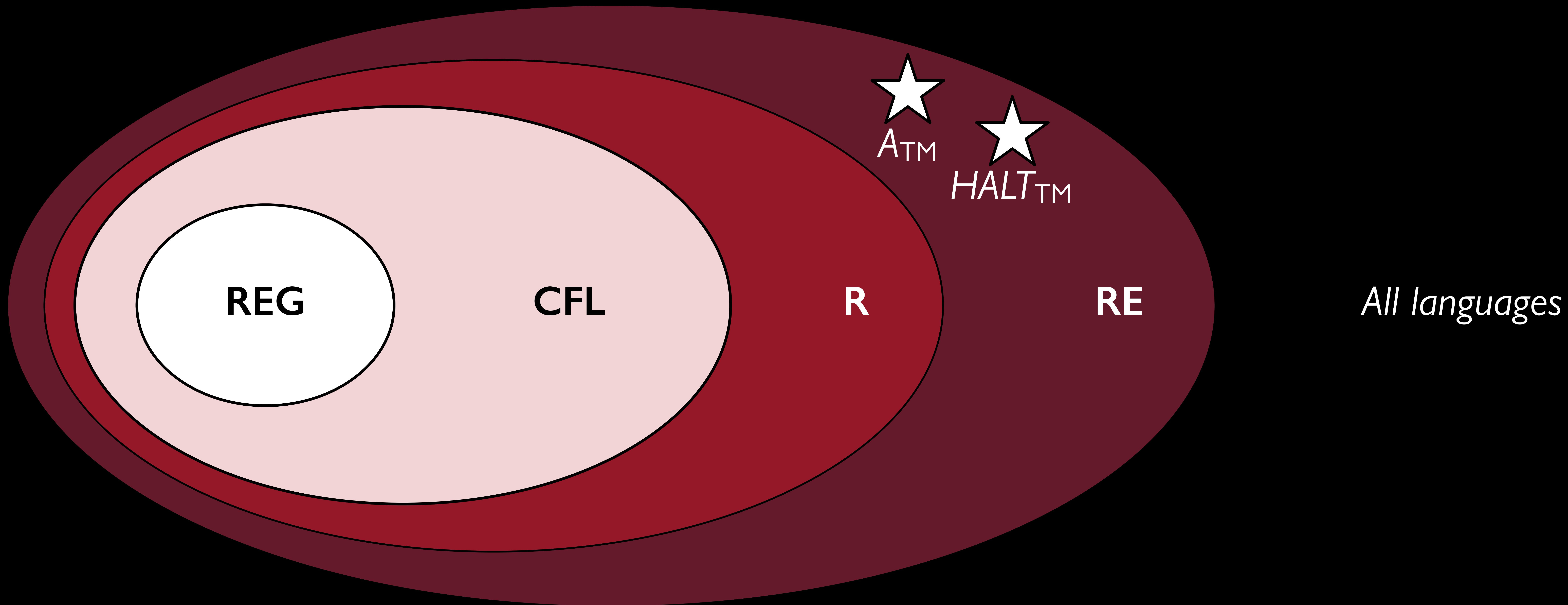
We were *born* skeptical.

Beyond **R** and **RE**

We've now seen how to use self-reference as a tool for showing undecidability – finding languages not in **R**.

We still haven't broken out of **RE** yet, though.

Anything out here?



Acknowledgments

This lecture incorporates material from:

- DALL-E via Bing Chat
- Umberto Eco, *On the Shoulders of Giants*
- Nancy Ide, Vassar College
- Keith Schwarz, Stanford University
- Michael Sipser, *Introduction to the Theory of Computation*