At the Foot of the Mountains of Undecidability

18 November 2021
“What problems can we solve with a computer?”
“What problems can we solve with a computer?”
The *Church–Turing thesis* states:

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

Problems solvable by any feasible computing machine

CFLs

Regular languages
Problems solvable by Turing machines

All languages

CFLs

Regular languages
Because of the Church–Turing thesis, we can start to be less precise with our TM descriptions.
A *high-level description* of a Turing machine is a description of the form

\[ M = \text{“On input } x: \text{ Do something with } x.” \]

For example:

\[ M = \text{“On input } x:\]

Repeat the following:

\[ \text{If } |x| \leq 1, \text{ accept.} \]

\[ \text{If the first and last symbols of } x \text{ aren’t the same, reject.} \]

\[ \text{Remove the first and last characters of } x.” \]
A *high-level description* of a Turing machine is a description of the form

\[ M = \text{"On input } x: \]

Do something with \( x \)."

For example:

\[ M = \text{"On input } x: \]

Construct \( y \), the reverse of \( x \).

If \( x = y \), accept.

Otherwise, reject."
A *high-level description* of a Turing machine is a description of the form

\[ M = \text{“On input } x:\text{ do something with } x.” \]

For example:

\[ M = \text{“On input } x:\text{ if } x \text{ is a palindrome, accept. Otherwise, reject.”} \]
A *high-level description* of a Turing machine is a description of the form

\[ M = \text{“On input } x: \text{ Do something with } x.” \]

For example:

\[ M = \text{“On input } x: \text{ Check that } x \text{ has the form } 0^n1^m2^p. \]
\[ \text{If not, reject.} \]
\[ \text{If } nm = p, \text{ accept.} \]
\[ \text{Otherwise, reject.”} \]
A *high-level description* of a Turing machine is a description of the form

\[ M = \text{“On input } x: \]

\[ \text{Do something with } x.” \]

For example:

\[ M = \text{“On input } x: \]

\[ \text{Check that } x \text{ has the form } s?t, \text{ where } s, t \in \{0, 1\}^*. \]

\[ \text{If not, reject.} \]

\[ \text{If so, if } s \text{ is a substring of } t, \text{ accept.} \]

\[ \text{Otherwise, reject.”} \]
Many languages require the input to be in some particular form.

Think about the ADD and SEARCH languages from the assignments.

We can encode this directly into our TMs:

\[ M = \text{"On input } s?t, \text{ where } s, t \in \{0, 1\}^* , \]
\[ \text{If } s \text{ is a substring of } t, \text{ accept.} \]
\[ \text{Otherwise, reject."} \]

Machines of this form implicitly reject any inputs that don’t have the right format.
Many languages require the input to be in some particular form.

Think about the ADD and SEARCH languages from the assignments.

We can encode this directly into our TMs:

\[ M = \text{"On input } 0^n1^m2^p, \]
\[ \text{If } mn = p, \text{ accept.} \]
\[ \text{Otherwise, reject."} \]

Machines of this form implicitly reject any inputs that don't have the right format.
What’s allowed?

Rule of thumb:

You can include *anything* in a high-level description, as long as you could write a computer program for it.

A few exceptions: Can’t get input from the user, make purely random decisions, etc.

Unsure about what you can do? Try building the TM explicitly.
“What problems can we solve with a computer?”
A *decision problem* is a type of problem where the goal is to answer yes or no.
How can we represent inputs?

Think about files on your computer:

  Each file represents some data, but each file is encoded purely using 0s and 1s.

If Obj is an object, then ⟨Obj⟩ denotes some string representing Obj, like how it might be stored on disk.

We can encode multiple objects as a single string, e.g., ⟨x, y⟩.
Now we can ask “what’s 5 + 3?” by asking for each possible $x$ whether $\langle 5, 3, x \rangle$ is in the language of sums.
Our goal is to speak of *computers solving problems*.

We model this by looking at *Turing machines recognizing languages*.

By turning any problem into an equivalent *decision problem*, this precisely captures what we’re interested in.
“What problems can we solve with a computer?”

What does it mean to “solve” a problem?
Unlike finite automata, which automatically halt after reading the input, Turing machines keep running until they explicitly enter an accept or reject state.

As such, it's possible for a Turing machine to run forever without accepting or rejecting.
If a Turing machine might run forever, how do we formally define what it means to “build a Turing machine for a language”?

What implications does this have for problem-solving?
Terminology

Let $M$ be a Turing machine.

$M$ accepts a string $w$ if it enters an accept state when run on $w$.

$M$ rejects a string $w$ if it enters a reject state when run on $w$.

$M$ loops infinitely (or just loops) on a string $w$ if, when run on $w$, it never enters an accept or reject state.
M does not accept $w$ if it either rejects $w$ or loops infinitely on $w$.

M does not reject $w$ if it either accepts $w$ or loops on $w$.

M halts on $w$ if it accepts $w$ or rejects $w$. 
The *language of a Turing machine* $M$ is

$$L(M) = \{ w \in \Sigma^* \mid M \text{ accepts } w \}$$

If $w \in L(M)$, $M$ accepts $w$.

If $w \notin L(M)$, $M$ does not accept $w$.

That is, when $M$ is run on $w$, either it rejects or it loops forever.
Recognizable languages

A language is called Turing-recognizable (or just recognizable) if it is the language of some Turing machine.

A Turing machine $M$ where $L(M) = L$ is called a recognizer for $L$.

The set of all languages that are Turing-recognizable is called $RE$. 
Does this correspond to what you think it means to “solve a problem”?
If a Turing machine $M$ halts on every possible input – i.e., it never goes into an infinite loop – then we call $M$ a *decider*.

For deciders, accepting is the same as not rejecting and rejecting is the same as not accepting:

$$\begin{align*}
\text{Does not reject} & \quad \left\{ \begin{array}{l}
\text{Accept} \\
\text{Reject}
\end{array} \right. \\
\text{Does not accept} & \quad \left\{ \begin{array}{l}
\text{Does not accept} \\
\text{Reject}
\end{array} \right. \\
\{\} & \quad \text{Halts (always)}
\end{align*}$$
Decidable languages

A language is called *Turing-decidable* (or just *decidable*) if it is the language of *some* decider.

Equivalently, a language $L$ is Turing-decidable if there is a Turing machine $M$ such that

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

The set of all languages that are Turing-decidable is called $R$. 
Decidable problems – the languages in $\mathbb{R}$ – are problems that can truly be “solved” by a computer.

(Though that solution isn’t guaranteed to be acceptably fast.)
A feel for **R** and **RE**

You have a DFA. You want to see if the DFA accepts any strings of the form $a^n b^n$. 
A feel for \textbf{R} and \textbf{RE}

You have a DFA. You want to see if the DFA accepts any strings of the form \(a^n b^n\).

\textit{Not whether the language is} \(a^n b^n\); \textit{that’s impossible for a DFA!}
A feel for **R** and **RE**

You have a DFA. You want to see if the DFA accepts any strings of the form $a^n b^n$.

**An RE perspective:** Run the DFA on $a^0 b^0$, $a^1 b^1$, $a^2 b^2$, etc. If the DFA ever accepts, return true. But, if not, you may never learn this.

**An R perspective:** Look at the structure of the DFA and, somehow, determine whether it accepts any strings of this form, but without running the DFA on all of them.

*Can we do this?*
A feel for **R** and **RE**

Say you’re working on a computer science assignment. You wonder if there’s any input that will make your program crash.

**An RE perspective:** Try running the program on every possible input. If you see it crash, return true. If it never crashes, you will never learn this.

**An R perspective:** Look at the source code and somehow determine, with 100% certainty, whether the program will ever crash.

*Can we do this?*
A feel for R and RE

You have an X. You want to see if there’s a Y where X and Y go well together.

An RE perspective: List all the Ys in some order and check if X and Y go well together. If so, return true. If not, you might not learn anything.

An R perspective: Look at X and, somehow, determine whether such a Y exists without checking all Ys.

Can we do this?
Intuition 1: Problems in \textbf{RE} are ones that can be approached by doing some sort of exhaustive search over a potentially infinite list of options.

Intuition 2: Problems in \textbf{R} are ones that can be solved without having to exhaustively try infinitely many possibilities.
Every decider is a Turing machine, but not every Turing machine is a decider.

So, $R \subseteq RE$.

But is $R = RE$?

That is, if you can confirm “yes” answers to a problem, can you also solve that problem?
Is this right?

All languages

CFLs

Regular languages

R

RE

All languages
Or this?

- Regular languages
- CFLs
- R
- RE
- All languages
“What problems can we solve with a computer?”
We're getting closer!

However, to understand the answer, we’re going to need to step back for a moment.
Let’s think about \textit{emergent properties}.

An emergent property of a system is a property that arises out of smaller pieces but which doesn’t seem to exist in any of the individual pieces, e.g.,

Individual neurons fire in response to particular combinations of inputs and this gives rise to human consciousness.

Individual atoms obey the laws of quantum mechanics and just interact with other atoms, and this gives rise to literally everything.
All computing systems equal to Turing machines exhibit several surprising emergent properties. Because of the Church–Turing thesis, these must be inherent to computation; computation can’t exist without them. These emergent properties are what ultimately make computation so interesting and powerful. But they’re also computation’s Achilles heel – they’re how we find concrete examples of impossible problems.
The key emergent properties of computation that we’ll discuss are:

*Universality*: There is a single computing device capable of performing any computation.

*Self-reference*: Computing devices can ask questions about their own behavior.

The combination of these properties leads to simple examples of impossible problems and elegant proofs of impossibility.
Emergent property: \textit{Universality}
We’ve been designing Turing machines to solve specific problems.

Do you have a dedicated computer for each task you need to perform?

- Your email computer?
- Your word-processing computer?
- Your cute-cat-picture computer?
Can we make a “reprogrammable Turing machine”?
A Turing machine simulator

It’s possible to program a Turing machine simulator on an unbounded memory computer.

If we accept some limits on the “infinite” tape, we can even do this on a real computer.
This simple Turing machine takes a string of A's and B's and rearranges them so that all the A's come first.

State | Significance
-----|--------------
Start | No B's so far in sequence
0     | At least one B so far
1     | An A has been found after a B and has been changed to a B
2     | Leftmost B is changed to A
3     | Leftmost B is changed to A

TO RUN: enter a sequence of A's and B's, position readhead on leftmost symbol, and start.
A Turing machine simulator

While a simulator like this is an interactive tool to help us understand the theoretical model, we can also imagine it as a procedure

\[
\text{simulate\_tm}(M: \text{TM}, w: \text{str}) \rightarrow \text{bool}
\]

with the following behavior:

If \( M \) accepts \( w \), then \( \text{simulate\_tm}(M, w) \) returns True.
If \( M \) rejects \( w \), then \( \text{simulate\_tm}(M, w) \) returns False.
If \( M \) loops on \( w \), then \( \text{simulate\_tm}(M, w) \) loops infinitely.
simulate_tm

true!

false!

(loop)
Anything that can be done with an unbounded-memory computer can be done with a Turing machine.

So there must be a Turing machine that has the behavior of the simulate_tm method.
TM that runs other TMs

M

w...input...

accept!

(loop)

reject!
$M\xrightarrow{w} \text{...input...}$

$Universal\ TM$

- accept!
- (loop)
- reject!
THEOREM (Turing, 1936): There is a Turing machine $U$ called the *universal Turing machine* that, when run on an input of the form $\langle M, w \rangle$, where $M$ is a Turing machine and $w$ is a string, simulates $M$ running on $w$ and does whatever $M$ does on $w$.

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

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

If $M$ loops on $w$, then $U$ loops on $\langle M, w \rangle$. 
$U$ does to input $\langle M, w \rangle$

what

$M$ does on input $w$. 
The universal Turing machine $U$, schematically
Imagine you have some machine $M$ (like a program) that you want to run on input $w$. 

**Machine $M$**

Imagine you have some machine $M$ (like a program) that you want to run on input $w$. 

**Input $w$**

... a a a a a ...
Machine $M$

Take $M$ and write it down as a string.
(Remember how we can encode the finite-state control as table.)

Input $w$

\[
\begin{array}{cccccc}
\vdots & a & a & a & a & \vdots \\
\end{array}
\]
Take $M$ and write it down as a string. (Remember how we can encode the finite-state control as table.)
Machine $M$

Input $w$

Now take your input $w$ and write it down too.
Now take your input $w$ and write it down too.
Feed this into U.
Machine $M$

- Start state: $q_0$
- Transition: $a \rightarrow \square, R$
- Transition: $\square \rightarrow \square, R$
- Accepting state: $q_{acc}$
- Rejecting state: $q_{rej}$

Input $w$

- Input symbols: $\ldots \ a \ a \ a \ a \ a \ \ldots$

Universal TM $U$

- Start state: $q_0$
- Transition: $a \rightarrow \square, R$
- Transition: $\square \rightarrow \square, R$
- Accepting state: $q_{acc}$
- Rejecting state: $q_{rej}$

Input $\langle M, w \rangle$

- Input symbols: $\ldots q_0 a \square R \ldots q_1 a \ldots a a a a a \ldots$
- Tape symbols: $\ldots a a a a a \ldots$

- If $M$ is in accepting state, look up what $M$ should do upon reading $w$
- If $M$ is in rejecting state, update state and tape
- Look at next char of $w$
Machine $M$

Input $w$

Universal TM $U$

Input $\langle M, w \rangle$
Machine $M$

Universal TM $U$

Input $w$

Input $\langle M, w \rangle$

If $M$ is in accepting state

If $M$ is in rejecting state

Look at next char of $w$

Look up what $M$ should do upon reading $w$

Update state and tape

⋯ $a$ $a$ $a$ $a$ $a$ ⋯
Machine $M$

Universal TM $U$

Input $w$

Input $\langle M, w \rangle$
Machine $M$

Input $w$

Universal TM $U$

Input $\langle M, w \rangle$

If $M$ is in accepting state

Look at next char of $w$

If $M$ is in rejecting state

Look up what $M$ should do upon reading $w$

Update state and tape
Machine $M$

- **Start state**: $q_0$
- **Accepting state**: $q_{acc}$
- **Rejecting state**: $q_{rej}$

Input $w$

```
... a a a a ...```

Universal TM $U$

- **Start state**: $q_0$
- **Accepting state**: $q_{acc}$
- **Rejecting state**: $q_{rej}$

Input $\langle M, w \rangle$

```
... q_0 a □ R ... q_1 a ... a a a a a ...```

- **If $M$ is in accepting state**
  - Look at next char of $w$
  - Update state and tape

- **If $M$ is in rejecting state**
  - Look up what $M$ should do upon reading $w$
Machine $M$

Universal TM $U$

Input $w$

Input $\langle M, w \rangle$

If $M$ is in accepting state

If $M$ is in rejecting state

Look at next char of $w$

Look up what $M$ should do upon reading $w$

Update state and tape

\[ \cdots \quad a \quad a \quad a \quad a \quad \cdots \]
Machine $M$

Universal TM $U$

Input $w$

Input $\langle M, w \rangle$
Machine $M$

Input $w$

Universal TM $U$

Input $\langle M, w \rangle$

If $M$ is in accepting state
- Look at next char of $w$
- Update state and tape

If $M$ is in rejecting state
- Look up what $M$ should do upon reading $w$
- Update state and tape
Machine $M$

- **Start state**: $q_0$
- **Accepting state**: $q_{\text{acc}}$
- **Rejecting state**: $q_{\text{rej}}$

- Transition $a \rightarrow \square, R$:
  - From $q_0$ to $q_1$
  - From $q_1$ to $q_{\text{rej}}$

Input $w$

```
⋯ a a a ⋯
```

Universal TM $U$

- **Start state**: $q_0$
- **Accepting state**: $q_{\text{acc}}$
- **Rejecting state**: $q_{\text{rej}}$

- Transition $a \rightarrow \square, R$:
  - From $q_0$ to $q_1$
  - From $q_1$ to $q_{\text{rej}}$

Input $\langle M, w \rangle$

```
⋯ q_0 a □ R ⋯ q_1 a ⋯ a a a ⋯
```

- Look at next char of $w$
- If $M$ is in accepting state, look up what $M$ should do upon reading $w$
- If $M$ is in rejecting state, update state and tape

```
⋯ q_0 a □ R ⋯ q_1 a ⋯ a a a ⋯
```
Machine $M$

Input $w$

Universal TM $U$

Input $\langle M, w \rangle$

If $M$ is in accepting state:
- Look at next char of $w$
- If $M$ is in rejecting state:
  - Look up what $M$ should do upon reading $w$
- Update state and tape
As a high-level description:

\[ U = \text{“On input } \langle M, w \rangle, \text{ where } M \text{ is a TM and } w \in \Sigma^*, \]

Run \( M \) on \( w \).

If \( M \) accepts \( w \), \( U \) accepts \( \langle M, w \rangle \).

If \( M \) rejects \( w \), \( U \) rejects \( \langle M, w \rangle \).”

If \( M \) loops on \( w \), then \( U \) loops as well. This is implicit in the description.
The universal Turing machine $U$ is capable of performing any computation that could ever be computed by any feasible computing machine.

Why?

By the Church–Turing thesis, if there is a feasible computing machine, it can be converted into an equivalent TM $M$. $U$ can then simulate the behavior of $M$. 
Since $U$ is a Turing machine, it has a language, $L(U)$.

What is the language of the universal Turing machine?
The language of $U$

Recall that the language of a Turing machine is the set of all strings that the Turing machine accepts.

$U$, when run on a string $\langle M, w \rangle$, where $M$ is a TM and $w$ is a string, will

- accept $\langle M, w \rangle$ if $M$ accepts $w$,
- reject $\langle M, w \rangle$ if $M$ rejects $w$, and
- loop on $\langle M, w \rangle$ if $M$ loops on $w$. 
The language of $U$

Recall that the language of a Turing machine is the set of all strings that the Turing machine accepts.

$U$, when run on a string $\langle M, w \rangle$, where $M$ is a TM and $w$ is a string, will

- accept $\langle M, w \rangle$ if $M$ accepts $w$,
- reject $\langle M, w \rangle$ if $M$ rejects $w$, and
- loop on $\langle M, w \rangle$ if $M$ loops on $w$.

$L(U) = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \} = \{ \langle M, w \rangle \mid M \text{ is a TM and } w \in L(M) \}$
The acceptance language for Turing machines, denoted $A_{TM}$, is the language of the universal Turing machine:

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

Because there is a Turing machine, $U$, that recognizes $A_{TM}$, we know $A_{TM} \in RE$. 
Why do we care about universality?
Reason 1: It has practical consequences
What happens if we replace the Turing machine input with a normal computer program?
What happens if we replace the Turing machine input with a normal computer program?
Programs simulating programs

The fact that there’s a universal Turing machine, combined with the fact that computers can simulate TMs and *vice versa*, means that it’s possible to write a program that simulates other programs.

These programs go by many names:

- An *interpreter* like Java or (most implementations of) Python.
- A *virtual machine*, like VMWare or VirtualBox, that simulates an entire computer.
Party like it’s 1999 1990

The key idea behind the universal TM is that TMs can be fed as inputs into other TMs.

Similarly, an interpreter is a program that takes other programs as inputs.

Similarly, an emulator is a program that takes entire computers as inputs.

This hits at the core idea that computing devices can perform computations on other computing devices.
Reason 2: It’s philosophically interesting
Can computers think?

On 15 May 1951, Alan Turing delivered a radio lecture on the BBC, where he argued that “it is not altogether unreasonable to describe digital computers as brains”.

Why would he think this, given the very limited abilities of computers of the time?
“I should also say that ‘If any machine can be appropriately described as a brain, then any [every] digital computer can be so described.’

“This last statement needs some explanation. It may appear rather startling, but with some reservations it appears to be an inescapable fact.

“It can be shown to follow from a characteristic property of digital computers, which I will call their universality…”
“A digital computer is a universal machine in the sense that it can be made to replace any machine of a certain very wide class.

“It will not replace a bulldozer or a steam-engine or a telescope, but it will replace any rival design of calculating machine, that is to say any machine into which one can feed data and which will later print out results.
“In order to arrange for our computer to imitate a given machine it is only necessary to programme the computer to calculate *what the machine in question would do under given circumstances*, and in particular what answers it would print out. The computer can then be made to print out the same answers.
“If now some machine can be described as a brain, we have only to programme our digital computer to imitate it and it will also be a brain. If it is accepted that real brains, as found in animals, and in particular in men, are a sort of machine it will follow that our digital computer suitably programmed, will behave like a brain.”
“This argument involves several assumptions which can quite reasonably be challenged.”

Alan Turing, 1951
Emergent property: Self-referentiality
PINOCCHIO

MY NAME ISN'T PINOCCHIO.

BRANSON REESE

WOW!
This is your brain on self-reference.
We can write programs that act on themselves.

As a fun case, we can write quines – programs that print out their own source code.

The way to do this varies by language, but it tends to be confusing!

E.g., a Scheme/Racket quine:

```scheme
((lambda (x)
    (list x (list 'quote x)))
  '(lambda (x) (list x (list 'quote x))))
```
The fact that we can write quines isn’t a coincidence.

THEOREM (Kleene’s second recursion theorem). It is possible to construct TMs that perform arbitrary computations on their own “source code” (the string encoding of that TM).

In other words, any computing system that’s equal to a Turing machine possess some mechanism for self-reference.
If this is already making your brain hurt, it’s ok.

When we think about “real” computer programs (rather than TMs), we can instead think of the program reading its own source code from disk:

```
selfie.py:

def main(my_input):
    my_source = open("selfie.py").read()
    print(my_source)
```

A program that reads from disk doesn’t count as a true quine, but it lets us use the same idea.
**Teaser:** Self-reference lets machines compute on themselves. That will let them do *cruel and unusual things.*
A note on TM/program equivalence
Every Turing machine receives some input, does some work, then (optionally) accepts or rejects.

We can model a Turing machine as a computer program where the program’s logic is written in a normal programming language, and the program (optionally) calls the special function accept() to immediately accept the input and reject() to immediately reject the input.
Here’s a sample program we might use to model a Turing machine for \( \{ w \in \{a, b\}^* \mid w \text{ has the same number of } a \text{ and } b \} \):

```python
def main(input):
    difference = 0
    for c in input:
        if c == "a":
            difference += 1
        elif c == "b":
            difference -= 1
        else:
            reject()
    if difference == 0:
        accept()
    reject()
```
We can always build a function `my_source` into a program which returns the source code of the program (as in a quine or reading from disk).

For example, here’s a narcissistic program:

```python
def main(input):
    me = my_source()
    if input == my_source:
        accept()
    reject()
```
Self-defeating objects
A **self-defeating object** is an object whose essential properties ensure it doesn’t exist.

“**I don’t know. I may be man’s best friend, but I’m my own worst enemy.**”
**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 $n < n + 1$.

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

Contradiction!

We’re using $n$ to construct something that undermines $n$, hence the term “self-defeating”.
The general template for proving that $x$ is a self-defeating object is as follows:

Assume that $x$ exists.

Construct some object $f(x)$ from $x$.

Show that $f(x)$ has some impossible property.

Conclude that $x$ doesn’t exist.

The particulars of what $x$ and $f(x)$ are and why $f(x)$ has an impossible property depend on the specifics of the proof.
Beware

CLAIM: There is a largest integer.

PROOF SKETCH: Assume $x$ is the largest integer.

Notice that $x > x - 1$.

So there’s no contradiction.
Beware

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!
Beware

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!
You *cannot* show that a self-defeating object \( x \) exists by using this line of reasoning:

Suppose that \( x \) exists.

Construct some object \( g(x) \) from \( x \).

Show that \( g(x) \) has no undesirable properties.

Conclude that \( x \) exists.

The fact that \( g(x) \) has no bad properties doesn’t mean that \( x \) exists. It just means you didn’t look hard enough for a counterexample.
Teaser: Certain Turing machines can’t exist, as they’d be self-defeating objects.
We stand at the foot of the mountains of undecidability.

Next time we’ll see how the emergent properties of universality and self-reference bring us to the limits of computation.
Acknowledgments

This lecture incorporates material from:

W. Daniel Hillis, *The Pattern on the Stone*
Nancy Ide, Vassar College
Keith Schwarz, Stanford University
Michael Sipser, *Introduction to the Theory of Computation*