Unsolvable Problems

Part One

Recap from Last Time

What problems can we solve with a computer?

What does it mean to solve a problem?

Very Important 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 enters neither an accept nor a reject state.

- *M **does not accept w*** if it either rejects *w* or loops infinitely on *w*.

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

- *M **halts on w*** if it accepts *w* or rejects *w*.

does not reject

does not accept

Accept

Loop

Reject

halts
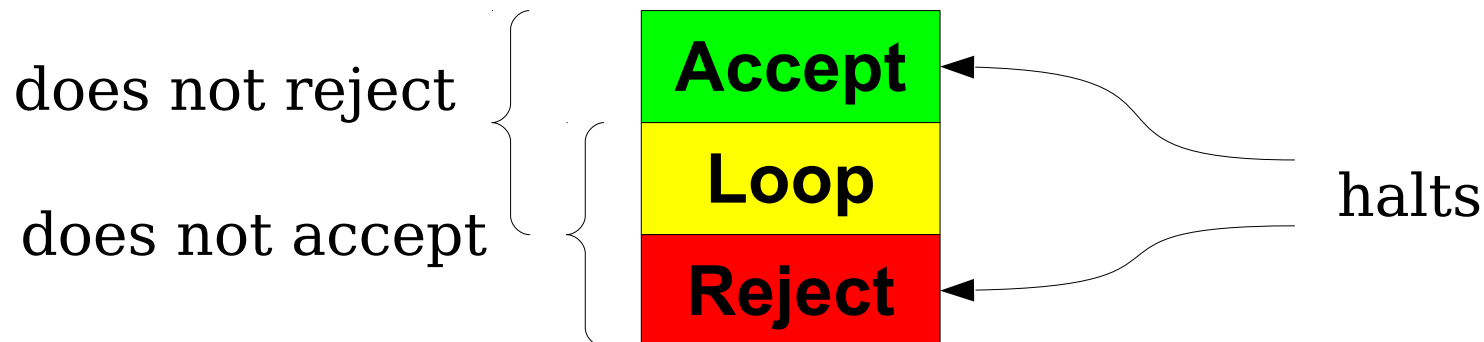
The Language of a TM

- The language of a Turing machine M, denoted $\mathscr{L}(M)$, is the set of all strings that M accepts:
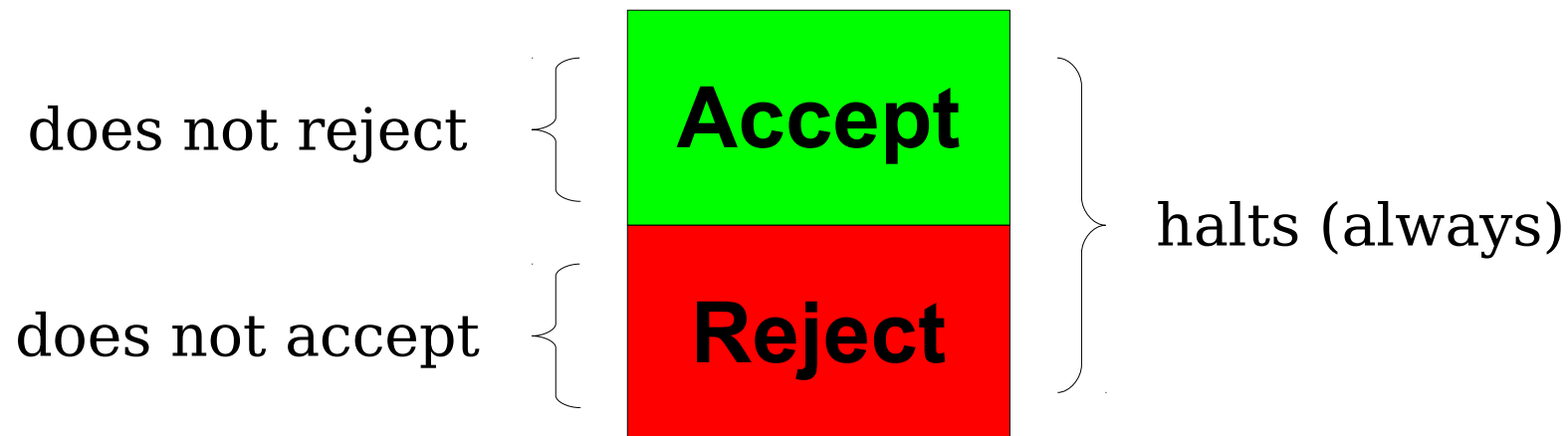
$$\mathscr{L}(M) = \{ \, w \in \Sigma^* \mid M \text{ accepts } w \, \}$$

- For any $w \in \mathscr{L}(M)$, M accepts w.

- For any $w \notin \mathscr{L}(M)$, M does not accept w.

  - It might loop forever, or it might explicitly reject.

- A language is called **recognizable** if it is the language of some TM. A TM for a language is sometimes called a **recognizer** for that language.

- Notation: the class **RE** is the set of all recognizable languages.

$$L \in \mathbf{RE} \quad \leftrightarrow \quad L \text{ is recognizable}$$

Deciders

- Some Turing machines always halt; they never go into an infinite loop.

- If M is a TM and M halts on every possible input, then we say that M is a **_decider_**.

- For deciders, accepting is the same as not rejecting and rejecting is the same as not accepting.

does not reject $\Big\{$ **Accept**

does not accept $\Big\{$ **Reject**

$\Big\}$ halts (always)

Decidable Languages

- A language L is called ***decidable*** if there is a decider M such that $\mathscr{L}(M) = L$.

- Equivalently, a language L is decidable if there is a TM M such that

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

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

- The class **R** is the set of all decidable languages.

$$L \in \mathbf{R} \quad \leftrightarrow \quad L \text{ is decidable}$$

- A language that is not in **R** is called ***undecidable***.

The Universal Turing Machine

- ***Theorem (Turing, 1936)****:* There is a Turing machine **U_{TM}** 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 (accepts, rejects, or loops).

- The observable behavior of U_{TM} is the following:

  - If M accepts w, then U_{TM} accepts $\langle M, w \rangle$.

  - If M rejects w, then U_{TM} rejects $\langle M, w \rangle$.

  - If M loops on w, then U_{TM} loops on $\langle M, w \rangle$.

- **U_{TM} accepts $\langle M, w \rangle$ if and only if M accepts w.**

The Language of U_{TM}

- U_{TM} accepts $\langle M, w \rangle$ iff M is a TM that accepts w.

- Therefore:

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

$$\mathscr{L}(U_{TM}) = \{ \langle M, w \rangle \mid M \text{ is a TM and } w \in \mathscr{L}(M) \}$$

- For simplicity, define $A_{TM} = \mathscr{L}(U_{TM})$.

Regular Languages

CFLs

RE

A_{TM}

All Languages

New Stuff!

Self-Referential Software

Quines

Quines

- A ***Quine*** is a program that, when run, prints its own source code.

- Quines aren't allowed to just read the file containing their source code and print it out; that's cheating.

- How would you write such a program?

Writing a Quine

Self-Referential Programs

- ***Claim:*** Going forward, assume that any program can be augmented to include a method called `mySource()` that returns a string representation of its source code.

- General idea:

  - Write the initial program with `mySource()` as a placeholder.

  - Use the Quine technique we just saw to convert the program into something self-referential.

  - Now, `mySource()` magically works as intended.

The Recursion Theorem

- There is a deep result in computability theory called ***Kleene's second recursion theorem*** that, informally, states the following:

  It is possible to construct TMs that perform arbitrary computations on their own descriptions.

- Intuitively, this generalizes our Quine constructions to work with arbitrary TMs.

- Want the formal statement of the theorem? Take CS154!

Time-Out for Announcements!

Midterm Logistics

- The midterm is tonight from 7PM – 10PM. *Good luck!*

- Locations are divvied up by last (family) name:

  - `Abd` – `Pre`: Go to Hewlett 200.

  - `Pri` – `Vil`: Go to Hewlett 201.

  - `Vo` – `Xie`: Go to Hewlett 101.

  - `Yan` – `Zhu`: Go to Hewlett 103.

Your Questions

"There is a lot of talk in the media about the computing power of quantum computers. Is the theory of quantum computers different from what we've learned in this class? How powerful do you think a quantum computer can be in theory?"

We must be looking at totally different news sources, since I haven't seen anything in the news about quantum computing in years. ☺

From a *computability* perspective, quantum computers are equal in power to Turing machines. There's nothing a TM can do that a quantum computer can't or vice-versa.

From a *complexity* perspective, the answer is "we're not fully sure." There's a ton of research into problems that quantum computers can solve fundamentally faster than regular computers. We have a few examples (look up Shor's algorithm and Grover's algorithm for details), but there are more questions than answers.

"Can we see you in a man bun?"

Next
question…

"I'm a female student in CS and I always feel like I'm behind, even though I know I'm not (in terms of grades, classes, internships, etc.) Do you have any tips on how I can get rid of that insecurity?"

"Going off a question below, I'm also a woman in CS. It seems like the default assumption is that women are bad at CS until proven otherwise. Do you have any advice on how to get past the insecurity and added pressures that come with this assumption?"

I don't have a short answer, so let me give you a long one.

Back to CS103!

Where We're Going

- We are about to use the existence of self-referential programs to find a concrete example of an undecidable problem.

- Before we go there, though, we need to set up some notation and terminology we'll use in a little while.

TMs and Programs

- Every TM

  - receives some input,

  - does some work, then

  - (optionally) accepts or rejects.

- We can model a TM as a computer program where

  - the input is provided by a special method `getInput()` that returns the input to the program,

  - the program's logic is written in a normal programming language, and the program

  - (optionally) calls the special method `accept()` to immediately accept the input and `reject()` to immediately reject the input.

TMs and Programs

- Here's a sample program we might use to model a Turing machine for { $w \in$ {**a**, **b**}* | w has the same number of **a**'s and **b**'s }:

```
int main() {
    string input = getInput();
    int difference = 0;

    for (char ch: input) {
        if (ch == 'a') difference++;
        else if (ch == 'b') difference--;
        else reject();
    }

    if (difference == 0) accept();
    else reject();
}
```

TMs and Programs

- As mentioned before, it's always possible to build a method `mySource()` into a program, which returns the source code of the program.

- For example, here's a narcissistic program:

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

    if (input == me) accept();
    else reject();
}
```

TMs and Programs

- Sometimes, TMs use other TMs as subroutines.

- We can think of a decider for a language as a method that takes in some number of arguments and returns a boolean.

- For example, a decider for $\{ a^n b^n \mid n \in \mathbb{N} \}$ might be represented in software as a method with this signature:

```
bool isAnBn(string w);
```

- Similarly, a decider for $\{ \langle m, n \rangle \mid m, n \in \mathbb{N}$ and m is a multiple of $n \}$ might be represented in software as a method with this signature:

```
bool isMultipleOf(int m, int n);
```

TMs and Programs

- On Wednesday, we built a TM for the language
  { $w \in \{0, 1\}^*$ | w has the same number of 0s and 1s } out
  of a decider for { 0^n1^n | $n \in \mathbb{N}$ } and a sorting subroutine.

- We could represent that in code like this:

```
bool is0n1n(string w);
string sort(string input);

int main() {
    string input = sort(getInput());
    if (is0n1n(input)) accept();
    else reject();
}
```

It's Showtime!

The Problem of Loops

- Suppose we have a TM M and a string w.

- If we run M on w, we may never find out whether $w \in \mathscr{L}(M)$ because M might loop.

- Is there some algorithm we can use to determine whether M is eventually going to accept w?

- If so, we *can* decide whether $w \in \mathscr{L}(M)$: we run the algorithm to see if M accepts w.

A Decider for A_{TM}

- ***Recall:*** A_{TM} is the language of the universal Turing machine.

- We know that $\langle M, w \rangle \in A_{TM}$ if and only if M accepts w.

- The universal Turing machine is a *recognizer* for A_{TM}. Could we build a *decider* for A_{TM}?

A Recipe for Disaster

- Suppose that $A_{TM} \in \mathbf{R}$.

- Formally, this means that there is a TM that decides A_{TM}.

- Intuitively, this means that there is a TM that takes as input a TM M and string w, then

  - accepts if M accepts w, and
  - rejects if M does not accept w.

A Recipe for Disaster

- To make the previous discussion more concrete, let's explore the analog for computer programs.

- If A_{TM} is decidable, we could construct a function

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

  that takes in as input a program and a string, then returns true if the program will accept the input and false otherwise.

- What could we do with this?

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();
    }
}
```

What does this program do?

```cpp
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?

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?

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?

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?

What does this program do?

```cpp
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?

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!

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!

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?

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?

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?

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?

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?

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?

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!

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!

Knowing the Future

- This TM is analogous to a classical philosophical/logical paradox:

  If you know what you are fated to do, can you avoid your fate?

- If A_{TM} is decidable, we can construct a TM that determines what it's going to do in the future (whether it will accept its input), then actively chooses to do the opposite.

- This leads to an impossible situation with only one resolution: A_{TM} ***must not be decidable!***

Theorem: $A_{TM} \notin \mathbf{R}$.

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}.

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 this machine is given any TM/string pair, it will then determine whether the TM accepts the string and report back the answer.

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 this machine is given any TM/string pair, it will then determine whether the TM accepts the string and report back the answer.

Given this, we could then construct the following TM:

$M =$ "On input w:

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

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 this machine is given any TM/string pair, it will then determine whether the TM accepts the string and report back the answer.

Given this, we could then construct the following TM:

M = "On input w:
      Have M obtain its own description, $\langle M \rangle$.
      Run D on $\langle M, w \rangle$ and see what it says.
      If D says that M will accept w, reject.
      If D says that M will not accept w, accept."

Choose any string w and trace through the execution of the machine, focusing on the answer given back by machine D.

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 this machine is given any TM/string pair, it will then determine whether the TM accepts the string and report back the answer.

Given this, we could then construct the following TM:

M = "On input w:
       Have M obtain its own description, $\langle M \rangle$.
       Run D on $\langle M, w \rangle$ and see what it says.
       If D says that M will accept w, reject.
       If D says that M will not accept w, accept."

Choose any string w and trace through the execution of the machine, focusing on the answer given back by machine D. If D says that M will accept w, notice that M then proceeds to reject w, contradicting what D says.

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 this machine is given any TM/string pair, it will then determine whether the TM accepts the string and report back the answer.

Given this, we could then construct the following TM:

M = "On input w:
      Have M obtain its own description, $\langle M \rangle$.
      Run D on $\langle M, w \rangle$ and see what it says.
      If D says that M will accept w, reject.
      If D says that M will not accept w, accept."

Choose any string w and trace through the execution of the machine, focusing on the answer given back by machine D. If D says that M will accept w, notice that M then proceeds to reject w, contradicting what D says. Otherwise, if D says that M will not accept w, notice that M then proceeds to accept w, contradicting what D says.

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 this machine is given any TM/string pair, it will then determine whether the TM accepts the string and report back the answer.

Given this, we could then construct the following TM:

M = "On input w:
      Have M obtain its own description, $\langle M \rangle$.
      Run D on $\langle M, w \rangle$ and see what it says.
      If D says that M will accept w, reject.
      If D says that M will not accept w, accept."

Choose any string w and trace through the execution of the machine, focusing on the answer given back by machine D. If D says that M will accept w, notice that M then proceeds to reject w, contradicting what D says. Otherwise, if D says that M will not accept w, notice that M then proceeds to accept w, contradicting what D says.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \notin \mathbf{R}$.

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 this machine is given any TM/string pair, it will then determine whether the TM accepts the string and report back the answer.

Given this, we could then construct the following TM:

M = "On input w:
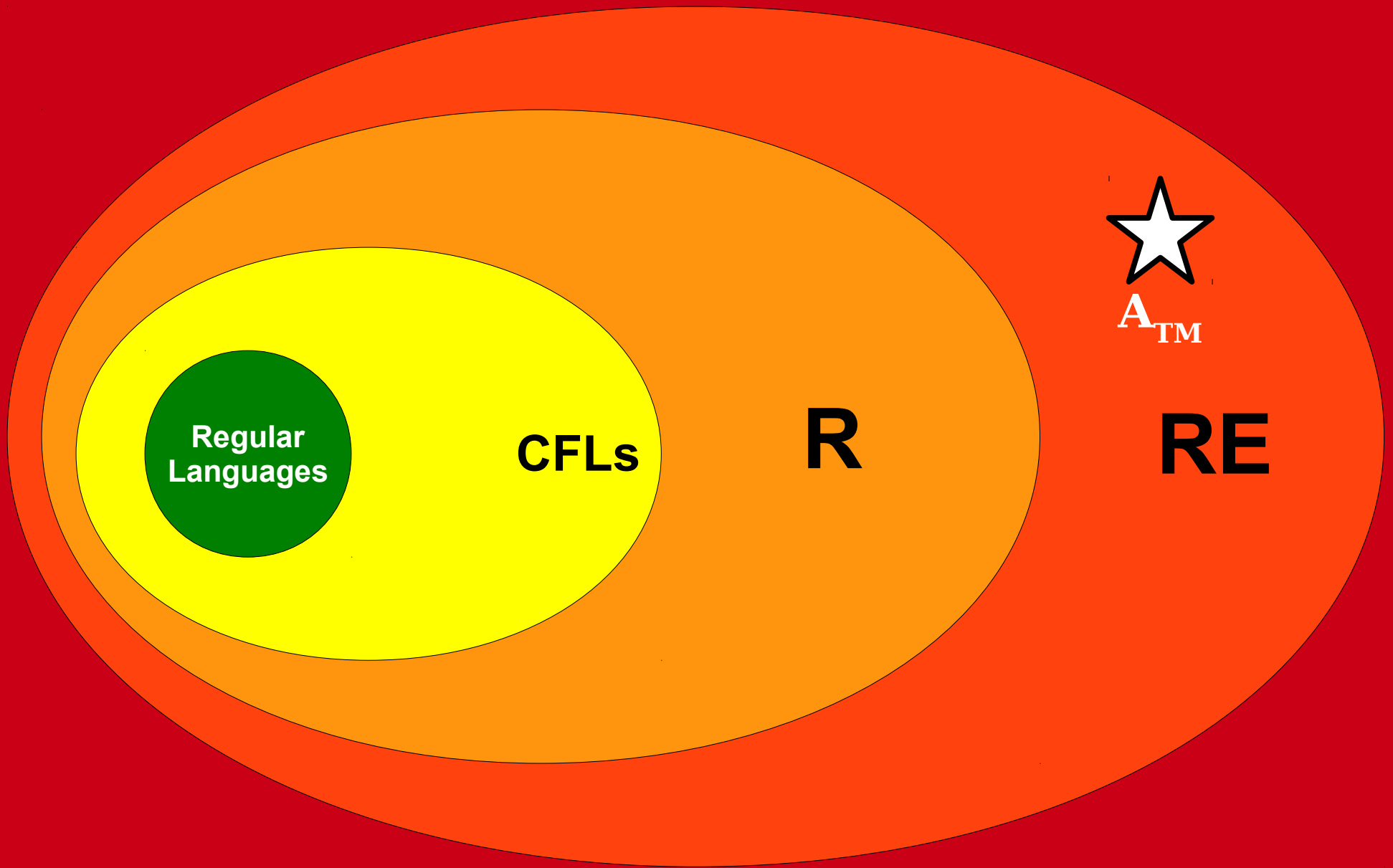   Have M obtain its own description, $\langle M \rangle$.
      Run D on $\langle M, w \rangle$ and see what it says.
      If D says that M will accept w, reject.
      If D says that M will not accept w, accept."

Choose any string w and trace through the execution of the machine, focusing on the answer given back by machine D. If D says that M will accept w, notice that M then proceeds to reject w, contradicting what D says. Otherwise, if D says that M will not accept w, notice that M then proceeds to accept w, contradicting what D says.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \notin \mathbf{R}$. ∎

Regular Languages

CFLs

R

A_{TM}

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 TM will accept a string.

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

- What do these two statements really mean? As in, why should you care?

$A_{TM} \notin R$

- The proof we've done says that

  There is no possible way to design an algorithm that will determine whether a program will accept an input.

- Notice that our proof just assumed there was some decider for A_{TM} and didn't assume anything about how that decider worked. In other words, no matter how you try to implement a decider for A_{TM}, you can never succeed!

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

- 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 an TM and any string w, either the TM accepts the string or it doesn't – *but there is no algorithm we can follow that will always tell us which it is!*

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

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

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

- As you'll see, this means that it's provably impossible for computers to be able to answer questions about what a program will do.

R ≠ RE

- The fact that **R ≠ RE** has enormous philosophical ramifications.

- A problem is in class **R** if there is an *algorithm* for solving it – there's some computational procedure that will give you the answer.

- A problem is in class **RE** if there is a *semialgorithm* for it. If the answer is "yes," the machine can tell this to you, but if the answer is "no," you may never learn this.

- Because **R ≠ RE**, there are some problems where "yes" answers can be checked, but there is no algorithm for deciding what the answer is.

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

More Impossibility Results

The Halting Problem

- The most famous undecidable problem is the *__halting problem__*, which asks:

    Given a TM *M* and a string *w*, will *M* halt when run on *w*?

- As a formal language, this problem would be expressed as

    ***HALT* = { ⟨*M, w*⟩ | *M* is a TM that halts on *w* }**

- How hard is this problem to solve?

- How do we know?

HALT ∉ **R**

- ***Claim:*** *HALT* ∉ **R**.

- If *HALT* is decidable, we could write some function

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

  that accepts as input a program and a string input, then reports whether the program will halt when run on the given input.

- Then, we could do this...

What does this program do?

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

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

    if (willHalt(me, input)) {
        while (true) {
            // loop infinitely
        }
    } else {
        accept();
    }
}
```

What does this program do?

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

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

    if (willHalt(me, input)) {
        while (true) {
            // loop infinitely
        }
    } else {
        accept();
    }
}
```

Imagine running this program on some input. What happens if…

… this program halts on that input?

What does this program do?

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

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

    if (willHalt(me, input)) {
        while (true) {
            // loop infinitely
        }
    } else {
        accept();
    }
}
```

Imagine running this program on some input. What happens if…

… this program halts on that input?

What does this program do?

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

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

    if (willHalt(me, input)) {
        while (true) {
            // loop infinitely
        }
    } else {
        accept();
    }
}
```

Imagine running this program on some input. What happens if…

… this program halts on that input?

What does this program do?

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

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

    if (willHalt(me, input)) {
        while (true) {
            // loop infinitely
        }
    } else {
        accept();
    }
}
```

Imagine running this program on some input. What happens if…

… this program halts on that input?

What does this program do?

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

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

    if (willHalt(me, input)) {
        while (true) {
            // loop infinitely
        }
    } else {
        accept();
    }
}
```

Imagine running this program on some input. What happens if...

... this program halts on that input?

What does this program do?

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

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

    if (willHalt(me, input)) {
        while (true) {
            // loop infinitely
        }
    } else {
        accept();
    }
}
```

Imagine running this program on some input. What happens if…

… this program halts on that input?
It loops on the input!

What does this program do?

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

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

    if (willHalt(me, input)) {
        while (true) {
            // loop infinitely
        }
    } else {
        accept();
    }
}
```

Imagine running this program on some input. What happens if…

… this program halts on that input?
It loops on the input!

What does this program do?

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

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

    if (willHalt(me, input)) {
        while (true) {
            // loop infinitely
        }
    } else {
        accept();
    }
}
```

Imagine running this program on some input. What happens if…

… this program halts on that input?
It loops on the input!

… this program loops on this input?

What does this program do?

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

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

    if (willHalt(me, input)) {
        while (true) {
            // loop infinitely
        }
    } else {
        accept();
    }
}
```

Imagine running this program on some input. What happens if…

… this program halts on that input?
It loops on the input!

… this program loops on this input?

What does this program do?

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

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

    if (willHalt(me, input)) {
        while (true) {
            // loop infinitely
        }
    } else {
        accept();
    }
}
```

Imagine running this program on some input. What happens if…

… this program halts on that input?
It loops on the input!

… this program loops on this input?

What does this program do?

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

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

    if (willHalt(me, input)) {
        while (true) {
            // loop infinitely
        }
    } else {
        accept();
    }
}
```

Imagine running this program on some input. What happens if…

… this program halts on that input?
It loops on the input!

… this program loops on this input?

What does this program do?

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

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

    if (willHalt(me, input)) {
        while (true) {
            // loop infinitely
        }
    } else {
        accept();
    }
}
```

Imagine running this program on some input. What happens if…

… this program halts on that input?
It loops on the input!

… this program loops on this input?

What does this program do?

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

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

    if (willHalt(me, input)) {
        while (true) {
            // loop infinitely
        }
    } else {
        accept();
    }
}
```

Imagine running this program on some input. What happens if...

... this program halts on that input?
It loops on the input!

... this program loops on this input?

What does this program do?

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

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

    if (willHalt(me, input)) {
        while (true) {
            // loop infinitely
        }
    } else {
        accept();
    }
}
```

Imagine running this program on some input. What happens if…

… this program halts on that input?
It loops on the input!

… this program loops on this input?
It halts on the input!

What does this program do?

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

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

    if (willHalt(me, input)) {
        while (true) {
            // loop infinitely
        }
    } else {
        accept();
    }
}
```

Imagine running this program on some input. What happens if…

… this program halts on that input?
It loops on the input!

… this program loops on this input?
It halts on the input!

Theorem: *HALT* ∉ **R**.

Proof: By contradiction; assume that *HALT* ∈ **R**. Then there is some decider *D* for *HALT*. 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 then construct the following TM:

M = "On input *w*:
      Have *M* obtain its own description, ⟨*M*⟩.
      Run *D* on ⟨*M*, *w*⟩ and see what it says.
      If *D* says that *M* halt on *w*, go into an infinite loop.
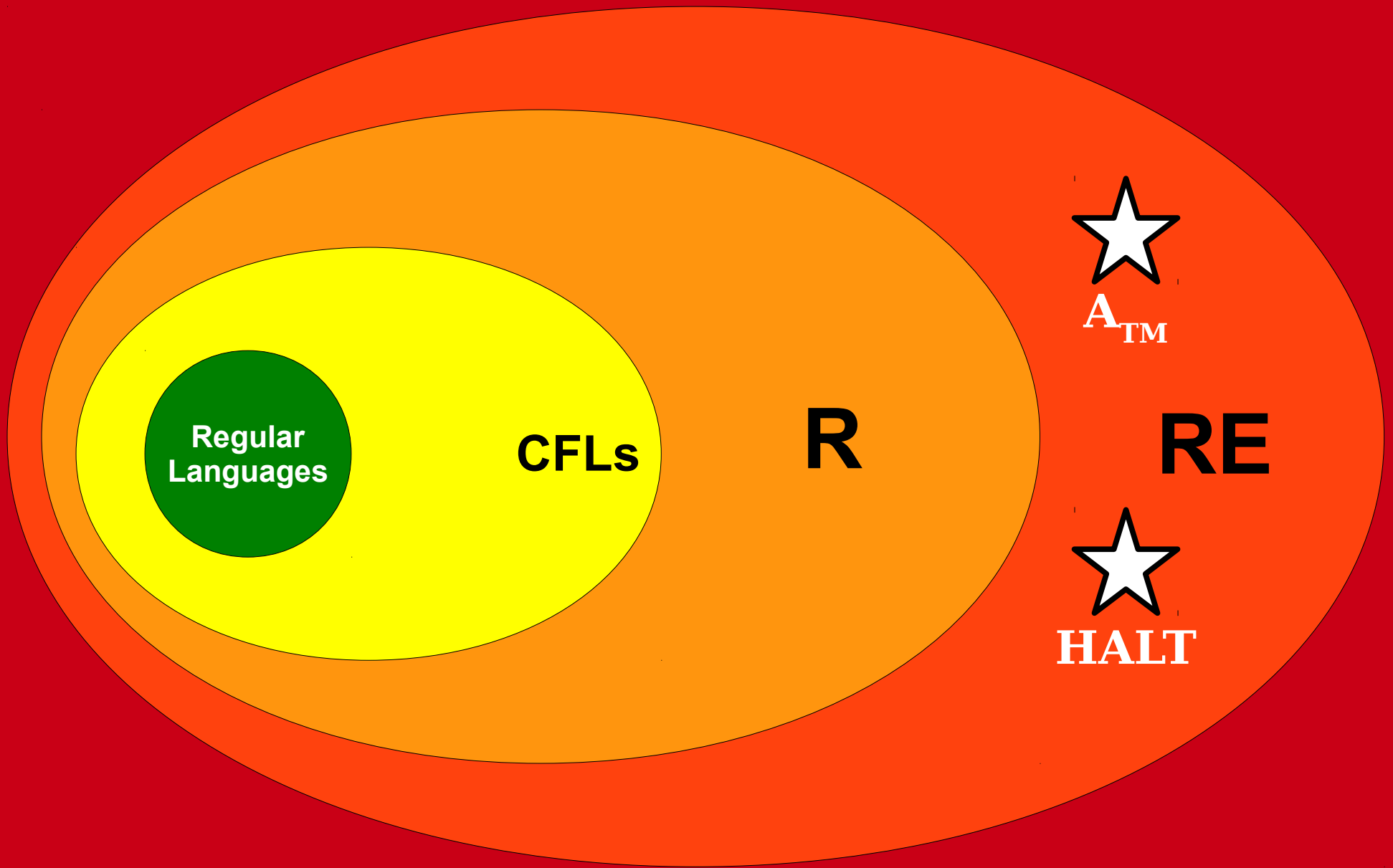   If *D* says that *M* loop on *w*, accept."

Choose any string *w* and trace through the execution of the machine, focusing on the answer given back by machine *D*. If *D* says that *M* will halt on *w*, notice that *M* then proceeds to loop on *w*, contradicting what *D* says. Otherwise, if *D* says that *M* will loop on *w*, notice that *M* then proceeds to accept *w*, so *M* halts on *w*, contradicting what *D* says.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, *HALT* ∉ **R**. ∎

$HALT \in \mathbf{RE}$

- ***Claim:*** $HALT \in \mathbf{RE}$.

- ***Idea:*** If you were certain that a TM *M* halted on a string *w*, could you convince me of that?

- Yes – just run *M* on *w* and see what happens!

```
int main() {
    TM M = getInputTM();
    string w = getInputString();

    feed w into M;
    while (true) {
        if (M is in an accepting state) accept();
        else if (M is in a rejecting state) accept();
        else simulate one more step of M running on w;
    }
}
```