Unsolvable Problems

Part One

Outline for Today

- ***Recap from Last Time***
  - Where are we, again?
- ***Self-Reference***
  - Computational party tricks!
- ***TMs as Programs***
  - A little simplification.
- ***Self-Defeating Objects***
  - A new perspective on some proofs.
- ***Undecidable Problems***
  - Concrete problems that are too hard for algorithms!

Recap from Last Time

The ***Church-Turing Thesis*** claims that

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

> “This is not a theorem – it is a falsifiable scientific hypothesis. And it has been thoroughly tested!”
>
> - Ryan Williams

R and **RE**

- A language L is ***recognizable*** if there is a TM M with the following property:

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

- That is, for any string w:
  - If $w \in L$, then M accepts w.
  - If $w \notin L$, them M does not accept w.
    - It might reject w, or it might loop on w.
- This is a "weak" notion of solving a problem.
- The class **RE** consists of all the recognizable languages.

R and **RE**

- A language L is ***decidable*** if there is a TM M with the following properties:

  $\forall w \in \Sigma^*$. (M accepts $w \leftrightarrow w \in L$).

  M halts on all inputs.

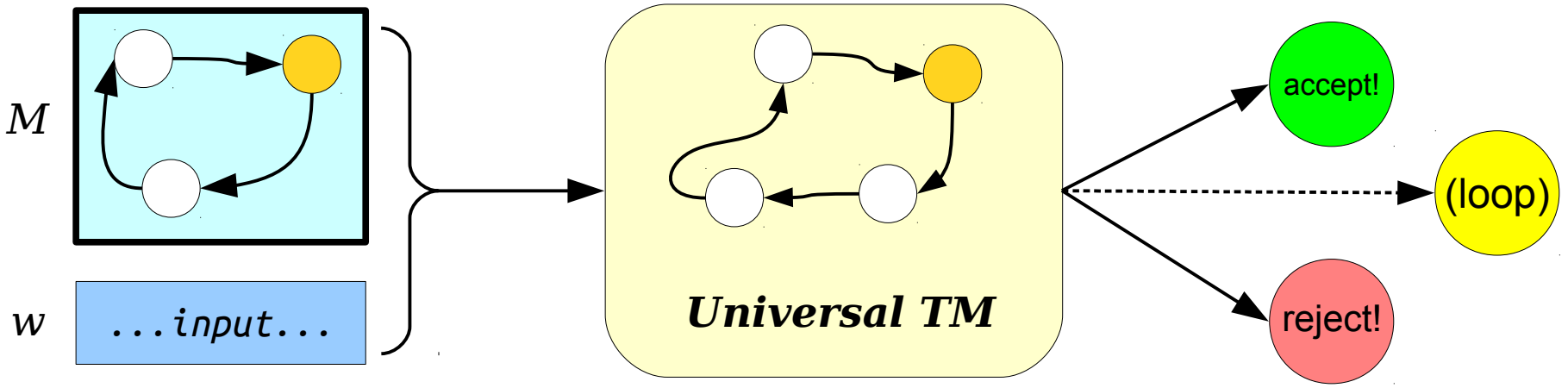
- That is, for any string w:
  - If $w \in L$, then M accepts w.
  - If $w \notin L$, then M rejects w.
- This is a "strong" notion of solving a problem.
- The class **R** consists of all the decidable languages.

Object Encodings

- 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*.
  - Think of it as how you'd store *Obj* on disk.
- We can encode multiple objects as a single string. For example, if M is a TM and w is a string, then ***⟨M, w⟩*** is a string representing the pair of M and w.

The Universal Turing Machine

- There is a TM named $\mathbf{U_{TM}}$ that is a ***universal Turing machine***.
- U_{TM} takes as input a pair $\langle M, w \rangle$, where M is a TM and w is a string.
- U_{TM} does to $\langle M, w \rangle$ whatever M does to w.



The Language A_{TM}

- The ***acceptance language for Turing machines***, denoted $\mathbf{A_{TM}}$, is the language of the universal Turing machine:

$$A_{TM} = \mathscr{L}(U_{TM})$$

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

- Useful fact:

$$\langle M, w \rangle \in A_{TM} \quad \leftrightarrow \quad M \text{ accepts } w.$$

- Because $A_{TM} = \mathscr{L}(U_{TM})$, we know that $A_{TM} \in \mathbf{RE}$.

Teaser #1:

This language A_{TM} has some interesting properties beyond what we've seen here.

New Stuff!

Self-Referential Software

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 (and technically incorrect if someone changes that file!)
- 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.

Self-Referential Programs

- The fact that we can write Quines is not 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 the TM).
- In other words, any computing system that's equal to a Turing machine possesses some mechanism for self-reference!
- Want to see how deep the rabbit hole goes? Take CS154!

Teaser #2:

Self-reference lets machines compute on themselves. That lets them do Cruel and Unusual Things.

A Note on TM/Program Equivalence

Equivalence of 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.

Equivalence of 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();
}
```

Equivalence of 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();
}
```

Equivalence of 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 $\{ \mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N} \}$ 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);
```

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 $n+1$?

Self-Defeating Objects

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! ■-*ish*

Question: Why doesn't the universal set exist?

(Refresher: the universal set contains everything.)

Answer: Because if $\mathscr{U}$ is the universal set, what happens when we look at $\wp(\mathscr{U})$?

Self-Defeating Objects

Theorem: The universal set doesn't exist.

Proof sketch: Suppose for the sake of contradiction that $\mathscr{U}$ exists.

Consider $\wp(\mathscr{U})$.

We know that $|\wp(\mathscr{U})| \leq |\mathscr{U}|$ because $\wp(\mathscr{U}) \subseteq \mathscr{U}$.

But Cantor's theorem tells us $|\mathscr{U}| < |\wp(\mathscr{U})|$.

Contradiction! ■-*ish*

Question: If S is a set, why can't there be a surjection $f : S \to \wp(S)$?

Answer: Because if f is a surjection from S to $\wp(S)$, what maps to $\{ x \in S \mid x \notin f(x) \}$?

Self-Defeating Objects

Theorem: No function $f : S \to \wp(S)$ is surjective.

Proof sketch: Suppose for the sake of contradiction that a surjective function $f : S \to \wp(S)$ exists.

Let $D = \{ x \in S \mid x \notin f(x) \}$.

Then if $f(x) = D$ for some $x \in S$, then $x \in f(x)$ if and only if $x \notin f(x)$.

Contradiction! ■-*ish*

Self-Defeating Objects

- 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.

An Important Point

Claim: There is a largest integer.

Proof: Assume x is the largest integer.

Notice that $x > x - 1$.

So there's no contradiction. ■-*ish*

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

How do we know there's no contradiction? We just checked one case.

Claim: The universal set exists.

Proof: Assume $\mathscr{U}$ exists.

Notice that $\varnothing \in \mathscr{U}$.

So there's no contradiction. ■-*ish*

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

How do we know there's no contradiction? We just checked one case.

Self-Defeating Objects

- 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 #3:

Certain Turing machines can’t exist, as they’d be self-defeating objects.

Time-Out for Announcements!

Don Knuth Lecture

- Don Knuth, Professor Emertius of the Art of Computer Programming, will be giving a public talk next Tuesday, December 4th from 6:30PM – 7:30PM here in NVIDIA Auditorium.
- The talk is on Dancing Links, an technique combining linked lists, recursive backtracking, and nondeterministic computation.
  - Whoa! It's like putting CS103 and CS106B into a particle accelerator and smashing them together!
- Highly recommended – this will probably be an amazing talk!

Problem Sets

- Problem Set Eight is due this Friday at 2:30PM.
  - Have questions? Stop by office hours or post online on Piazza! Feel free to post privately if you'd like.
  - This is the last problem set where you can use late days, but be strategic about doing so, since PS9 goes out on Friday.
- Problem Set Seven will be graded and returned later this evening.

~~Your Questions~~

Next time! Sorry - I was out sick yesterday and forgot to set this up.

Back to CS103!

Learning About a String

- Suppose M is a recognizer for some important language.
- We have a string w and we really, really want to know whether $w \in \mathscr{L}(M)$.
- How could we do this?

If you want to know whether this is true...

Observation:

$w \in \mathscr{L}(M)$

if and only if

M accepts w.

... you can try to determine whether this is true.

Learning About a String

- ***Option 1:*** Run M on w.
- What could happen?
  - M could accept w. Great! We know $w \in \mathscr{L}(M)$.
  - M could reject w. Great! We know $w \notin \mathscr{L}(M)$.
  - M could loop on w. Hmmm. We've learned nothing.
- This won't always tell us whether $w \in \mathscr{L}(M)$. We'll need a different strategy.

If you want to know whether this is true…

Observation:

$$w \in \mathscr{L}(M)$$

if and only if

$$M \text{ accepts } w$$

if and only if

$$\langle M, w \rangle \in A_{TM}.$$

… you can try to determine whether this is true.

Learning About a String

- ***Option 2:*** Use the universal Turing machine, which is a recognizer for A_{TM}!
- Specifically, run U_{TM} on $\langle M, w \rangle$.
- What could happen?
  - U_{TM} could accept $\langle M, w \rangle$. Great! Then $w \in \mathscr{L}(M)$.
  - U_{TM} could reject $\langle M, w \rangle$. Great! Then $w \notin \mathscr{L}(M)$.
  - U_{TM} could loop on $\langle M, w \rangle$. Hmmm. We've learned nothing.
- This won't always tell us whether $w \in \mathscr{L}(M)$. We'll need a different strategy.

Learning About a String

Option 2: Use the universal Turing machine, which is a recognizer for A_{TM}!

Specifically, run U_{TM} on $\langle M, w \rangle$.

What could happen?

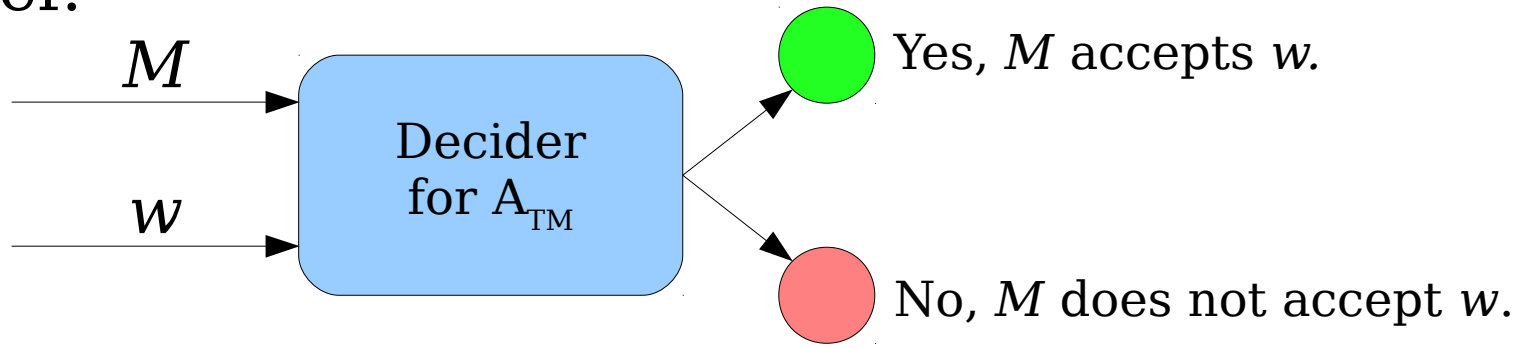
- U_{TM} could accept $\langle M, w \rangle$. Great! Then $w \in \mathscr{L}(M)$.
- U_{TM} could reject $\langle M, w \rangle$. Great! Then $w \notin \mathscr{L}(M)$.
- U_{TM} could loop on $\langle M, w \rangle$. Hmmm. We've learned nothing.

This won't always tell us whether $w \in \mathscr{L}(M)$. We'll need a different strategy.

What if we used a ***decider***, not a ***recognizer***?

Learning About a String

- ***Option 3:*** Build a *decider* for A_{TM}, rather than just a recognizer.

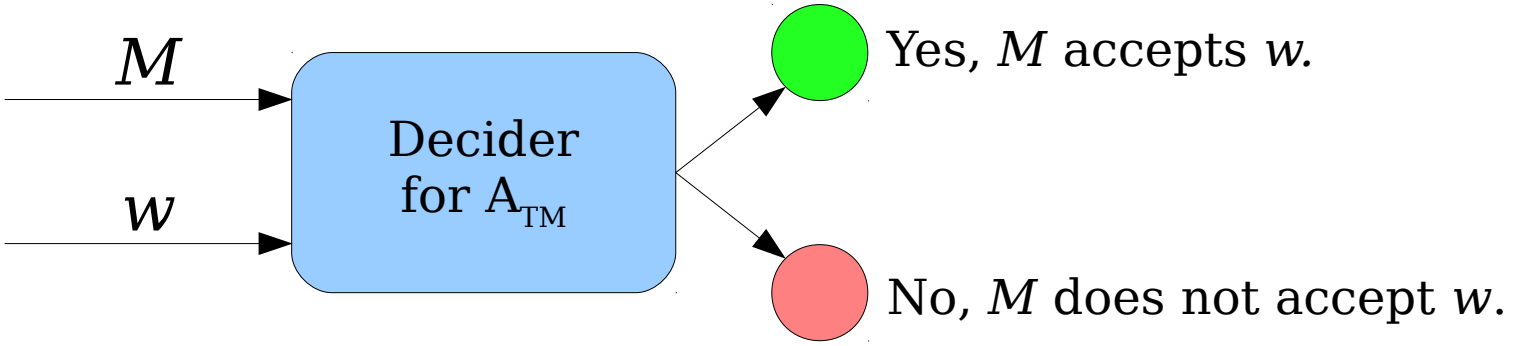


- Specifically, build a decider for A_{TM}, then run that decider on $\langle M, w \rangle$.
- What could happen?
  - The decider could accept $\langle M, w \rangle$. Then $w \in \mathscr{L}(M)$.
  - The decider could reject $\langle M, w \rangle$. Then $w \notin \mathscr{L}(M)$.
- ***Question:*** How do we build this decider?

Claim: A decider for A_{TM} is a self-defeating object. It therefore doesn't exist.

A Self-Defeating Object

- Let's suppose that, somehow, we managed to build a decider for A_{TM}.
- Schematically, that decider would look like this:



- We could represent this decider in software as a method

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

that takes as input a program and a string, then returns whether that program will accept that string.

What does this program do?

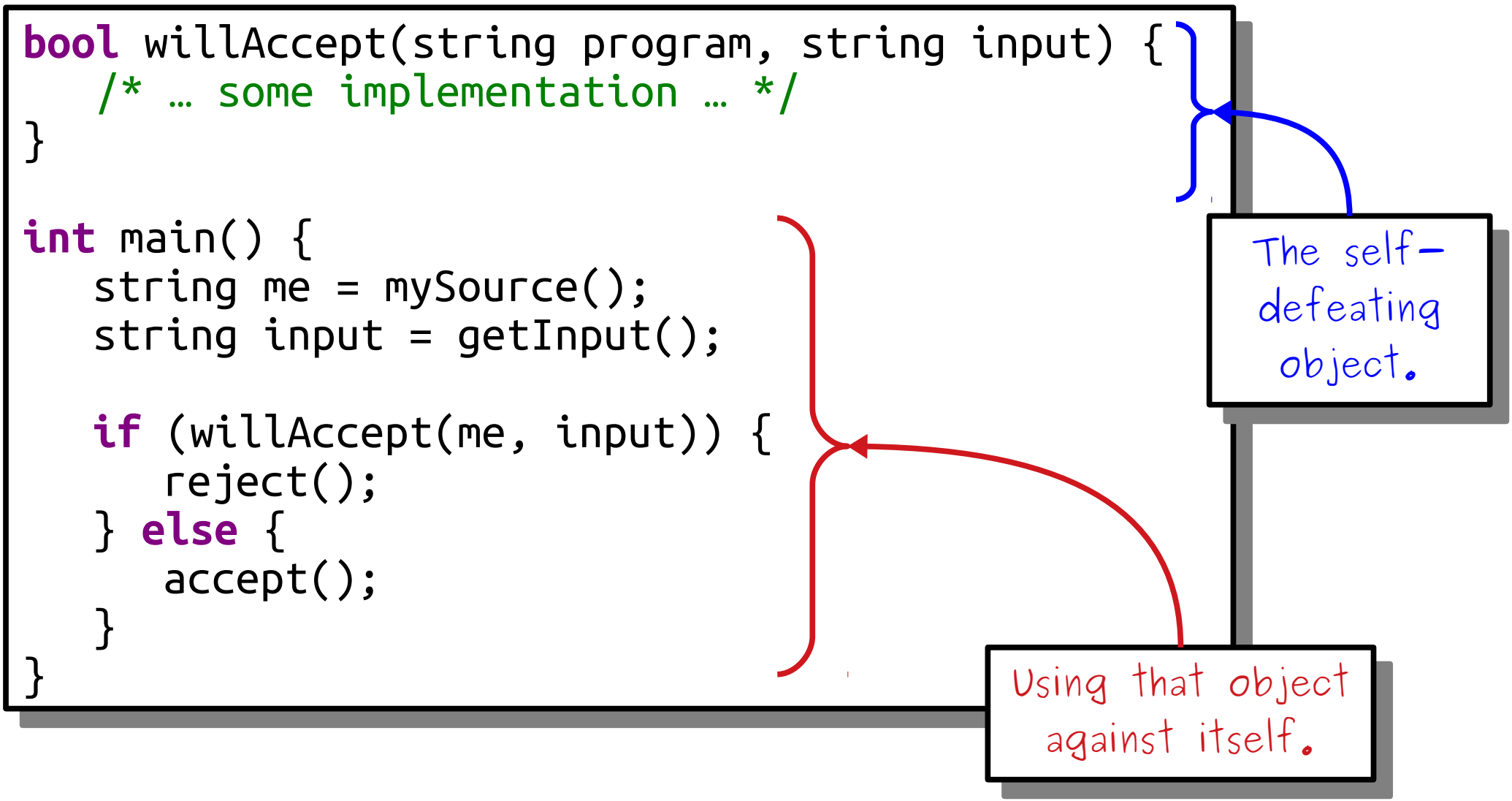
```
bool willAccept(string program, string input) {
   /* … some implementation … */
}

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?



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

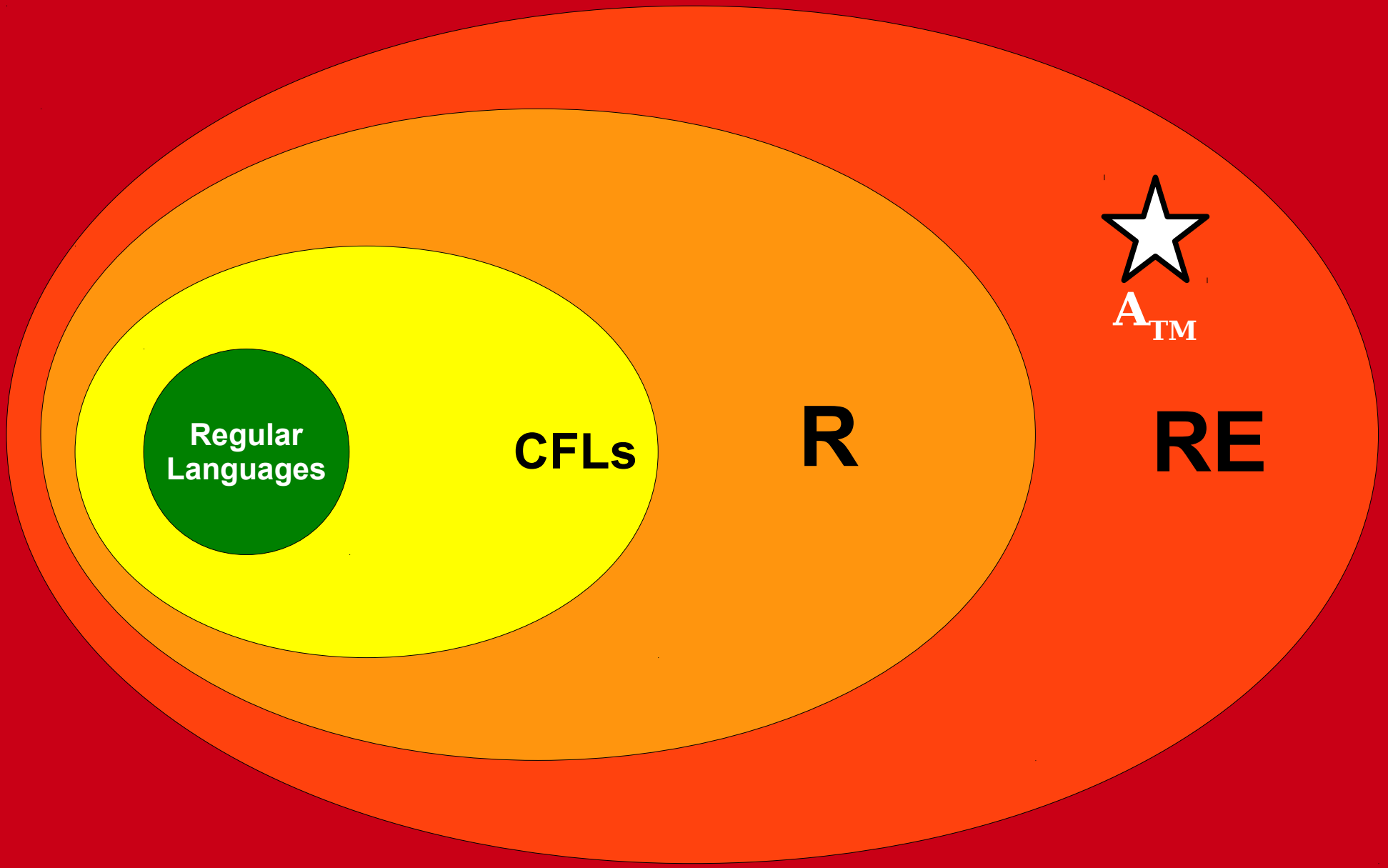
Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. Then there is some decider D for A_{TM}, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program P:

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

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

Choose any string w and trace through the execution of program P on input w, focusing on the answer given back by the `willAccept` method. If `willAccept(me, input)` returns true, then P must accept its input w. However, in this case P proceeds to reject its input w. Otherwise, if `willAccept(me, input)` returns false, then P must not accept its input w. However, in this case P proceeds to accept its input w.

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


Regular Languages
CFLs
R
RE
A_{TM}
All Languages

What Does This Mean?

- In one fell swoop, we've proven that
  - A decider for A_{TM} is a self-defeating object.
  - 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 three statements really mean? As in, why should you care?

Self-Defeating Objects

- The fact that a decider for A_{TM} is a self-defeating object is analogous to this classic philosophical question:

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

- If we have a decider for A_{TM}, we could use it to build a TM that determines what it's supposed to do next, then chooses to do the opposite!

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

- The proof we've done says that

There is no algorithm that can determine whether a program will accept an input.

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

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

- 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.

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

- 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*, one of these two statements is true:

  M* accepts *w ***M* does not accept *w***

  But since A_{TM} is undecidable, there is 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 you have the answer, you can confirm it (build a recognizer), but where if you don't have the answer, you can't come up with it in a mechanical way (build a decider).

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

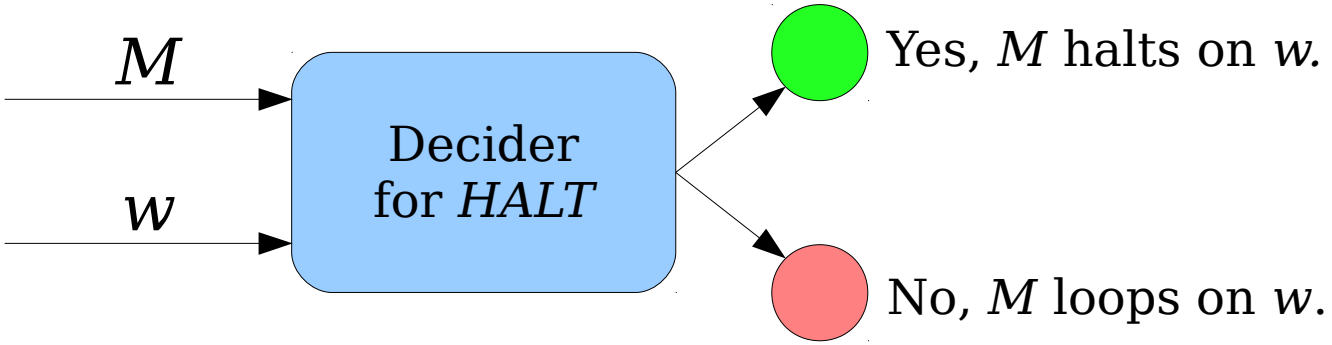
  ***HALT* = { ⟨*M*, *w*⟩ | *M* is a TM that halts on *w* }**

- This is an **RE** language. *(We'll see why later.)*
- How do we know that it's undecidable?

Claim: A decider for *HALT* is a self-defeating object. It therefore doesn't exist.

A Decider for *HALT*

- Let's suppose that, somehow, we managed to build a decider for $HALT = \{ \langle M, w \rangle \mid M \text{ is a TM that halts on } w \}$.
- Schematically, that decider would look like this:



- We could represent this decider in software as a method

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

that takes as input a program and a string, then returns whether that program will halt on that string.

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();
    }
}
```

The self-defeating object.

Using that object against itself.

Theorem: *HALT* $\notin$ **R**.

Proof: By contradiction; assume that *HALT* $\in$ **R**. Then there's a decider *D* for *HALT*, which we can represent in software as a method `willHalt` that takes as input the source code of a program and an input, then returns true if the program halts on the input and false otherwise.

Given this, we could then construct this program *P*:

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

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

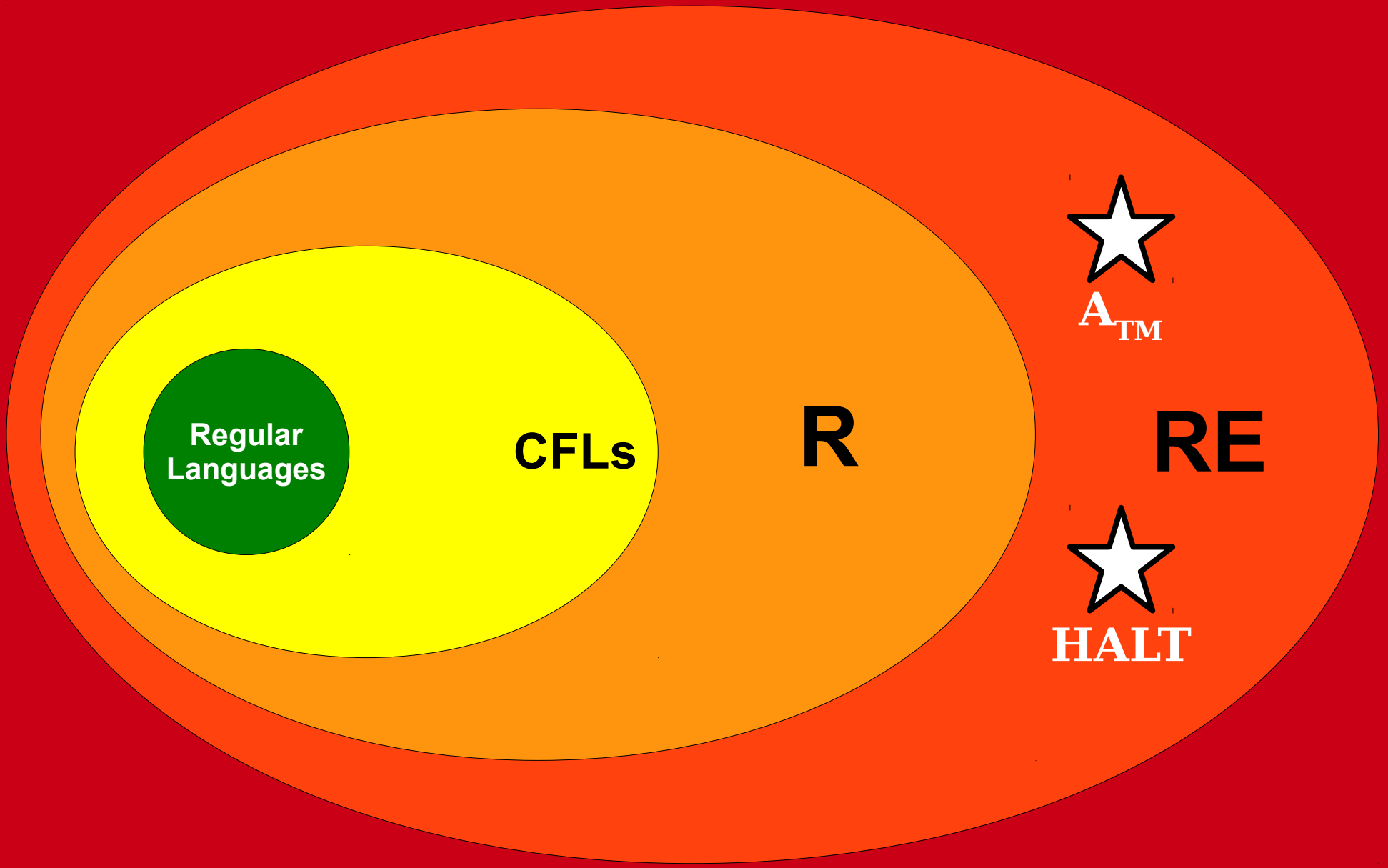
Choose any string *w* and trace through the execution of program *P* on input *w*, focusing on the answer given back by the `willHalt` method. If `willHalt(me, input)` returns true, then *P* must halt on its input *w*. However, in this case *P* proceeds to loop infinitely on *w*. Otherwise, if `willHalt(me, input)` returns false, then *P* must not halt its input *w*. However, in this case *P* proceeds to accept its input *w*.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, *HALT* $\notin$ **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;
    }
}
```


Regular Languages
CFLs
R
RE
A_{TM}
HALT
All Languages

Next Time

- ***Intuiting RE***
  - What exactly is the class **RE** all about?
- ***Verifiers***
  - A totally different perspective on problem solving.
- ***Beyond RE***
  - Finding an impossible problem using very familiar techniques.