Unsolvable Problems

Part One

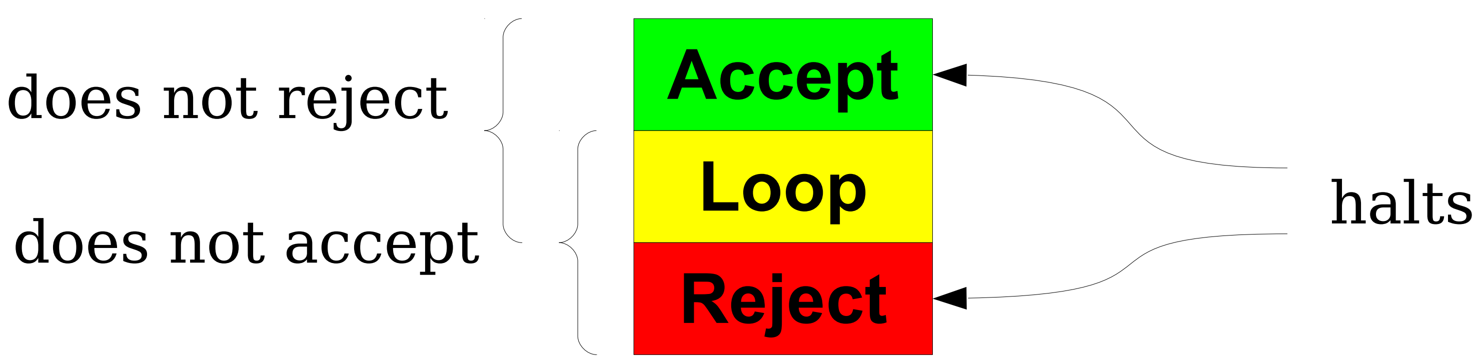
Recap from Last Time

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.



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:

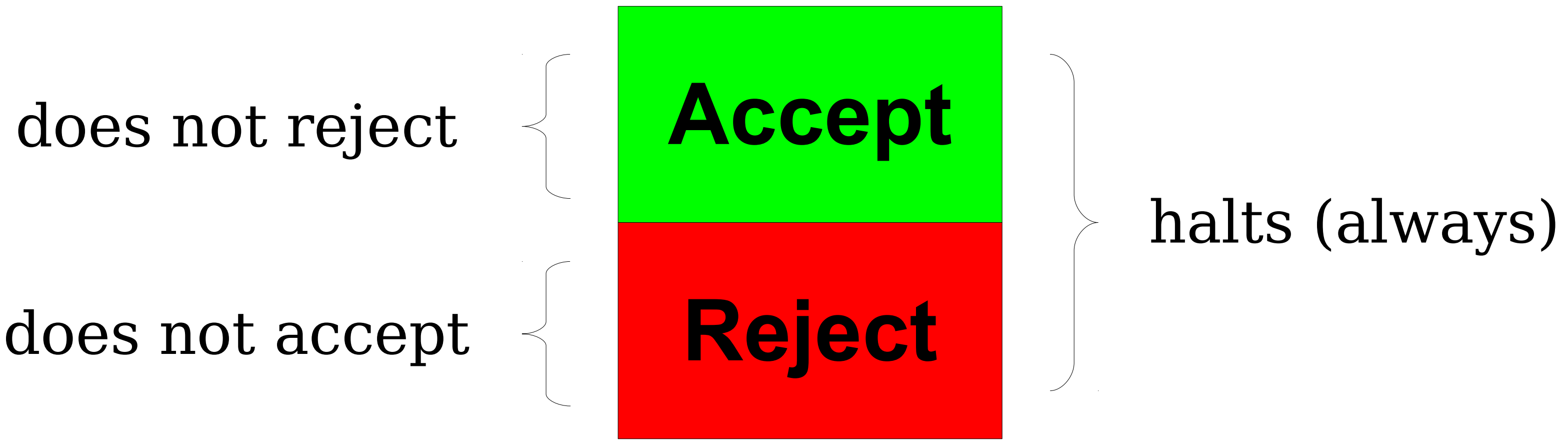
$$\boldsymbol{\mathscr{L}(M) = \{\ w \in \Sigma^* \mid M \text{ accepts } w\ \}}$$

- 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.



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 $\mathbf{R}$ is the set of all decidable languages.

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

- A language that is not in $\mathbf{R}$ is called ***undecidable***.

The Universal Turing Machine

- ***Theorem (Turing, 1936)****:* There is a Turing machine $\mathbf{U_{TM}}$ 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.**

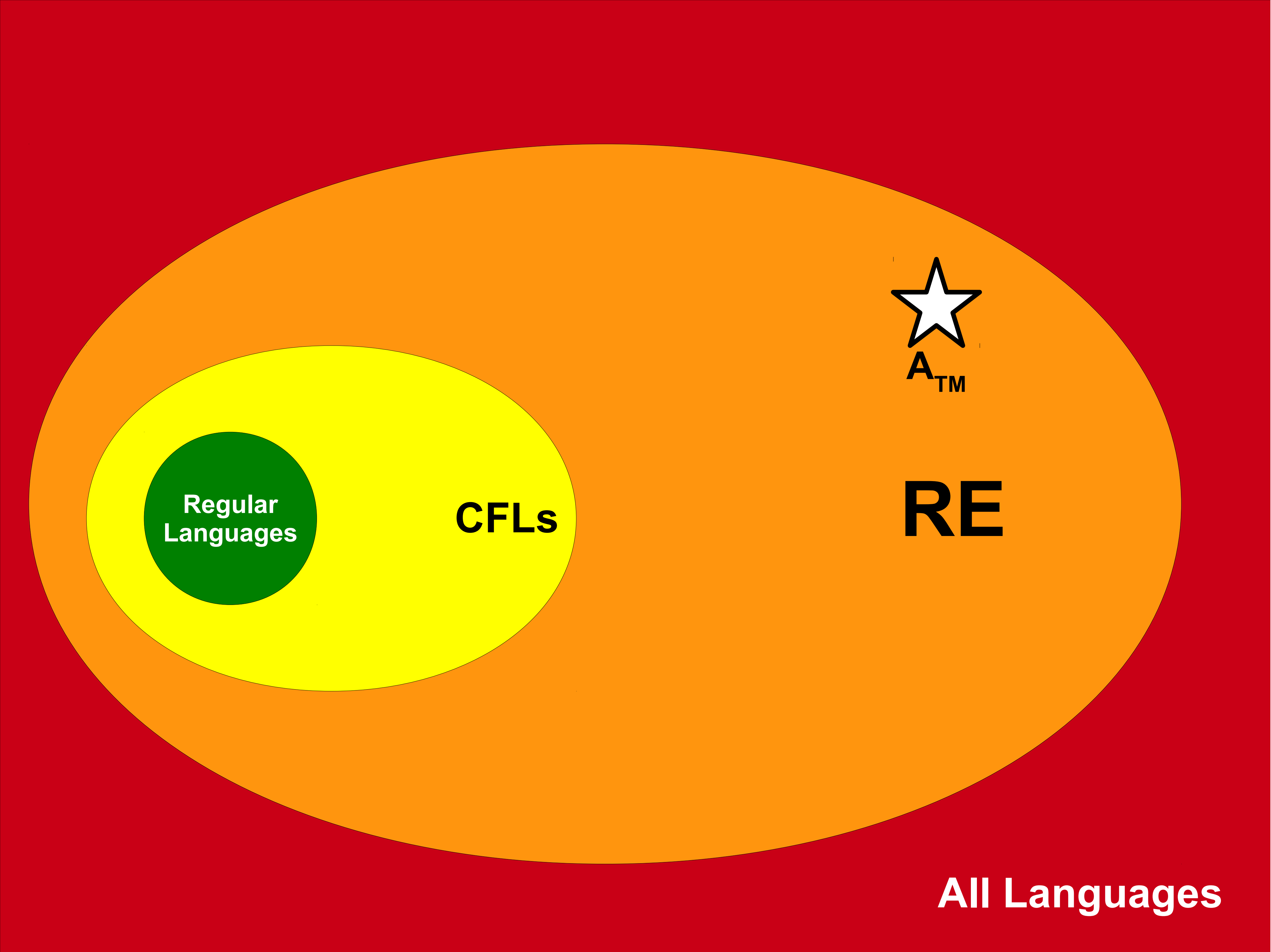
The Language of U_{TM}

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



Regular Languages
CFLs
A_{TM}
RE
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 (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.

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!

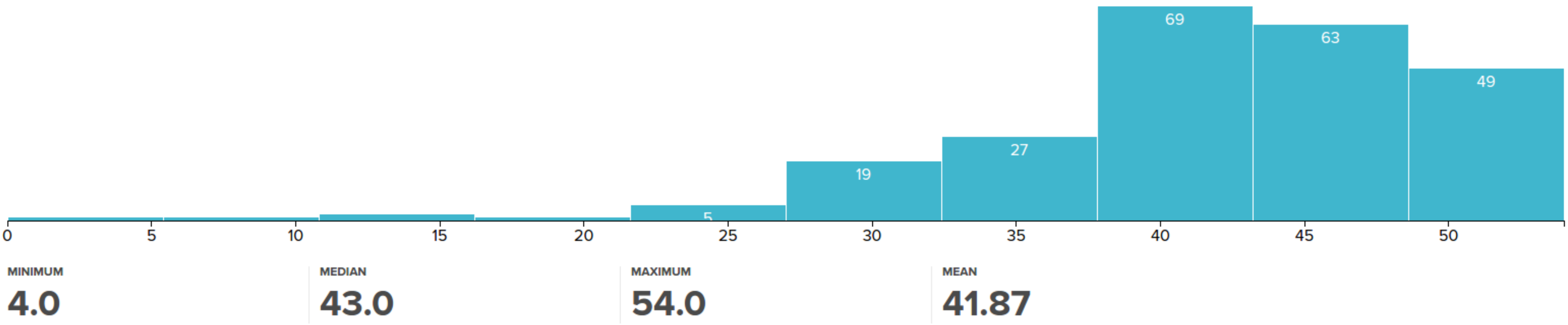
Time-Out for Announcements!

Problem Sets

- Problem Set Seven was due at the start of class today.
  - Want to use late days? You can extend the deadline to Monday.
- Problem Set Eight goes out today. It’s due next Friday.
  - Explore CFGs, Turing machines, **R** and **RE** languages, and the limits of computation!

Problem Set Six

- Here's the score distribution for PS6:



- The most common errors were trying to prove (rather than disprove) the results in Q6 and errors in the automaton for finding cocoa.

Extra Practice Problems

- We’ve released three more sets of extra practice problems (EPP9 – EPP11) to the course website.
- These are cumulative review problems that span all the topics from the quarter. We hope that this gives you a chance to study up on the topics you’re interested in!
- We’ll release solutions on Wednesday, along with some practice final exams.

Midterm Regrades

- We’ve finished regrading midterm exams. They’re now available for pickup in the filing cabinet with all the other midterms.

Your Questions

“How often do CS/EE majors enter non-tech careers, such as medicine, law, or government?”

Whatever the frequency is, it's not enough. We desperately need more people with technical skills in decision-making/leadership roles; at least at the US level, federal policy surrounding tech is either lacking, poorly thought out, or both.

I've written rec letters for folks going to medical school with a CS background, and that's a really exciting direction that I wish more people would explore. I also know some former section leaders who are now lawyers and some law students who are interested in studying CS, and I'm excited to see what they do going forward!

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
  - 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`$\}^* \mid w$ 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 $\{ \mathbf{a}^n\mathbf{b}^n \mid n \in \mathbb{N} \}$ might be represented in software as a method with this signature:

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

- Similarly, a decider for $\{ \langle m, n \rangle \mid m, n \in \mathbb{N} \text{ and } m \text{ is a multiple of } n \}$ 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 $\{ w \in \{\mathbf{0}, \mathbf{1}\}^* \mid w$ has the same number of $\mathbf{0}$s and $\mathbf{1}$s $\}$ out of a decider for $\{ \mathbf{0}^n\mathbf{1}^n \mid n \in \mathbb{N} \}$ 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();
    }
}
```

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: $\mathbf{A_{TM}}$ ***must not be decidable!***

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.

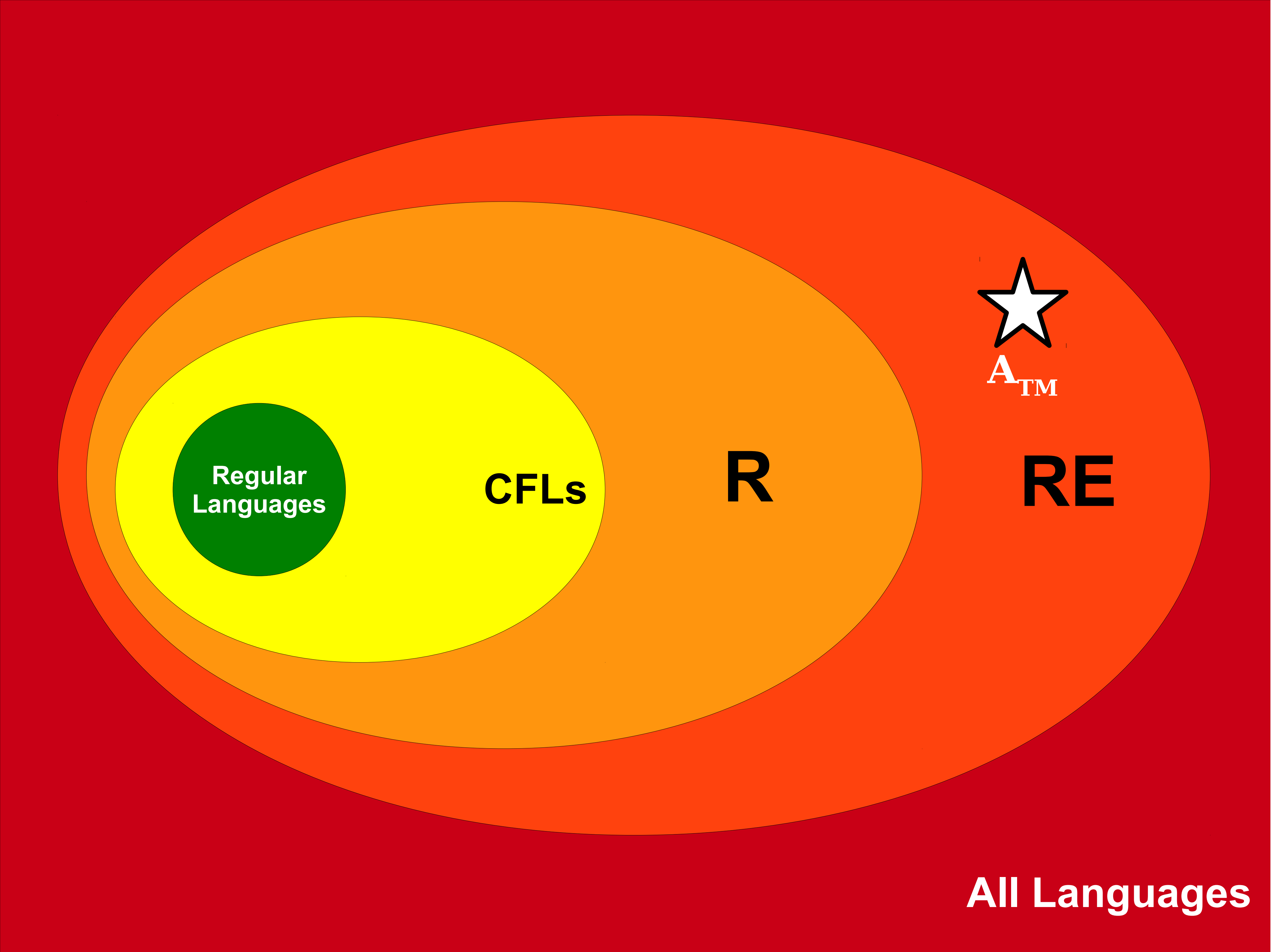
Given this, we could then construct this program P:

```
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_{TM} is ***undecidable***; there is no general algorithm that can determine whether a TM will accept a string.
  - $\mathbf{R} \neq \mathbf{RE}$, because $A_{TM} \notin \mathbf{R}$ but $A_{TM} \in \mathbf{RE}$.
- What do these two statements really mean? As in, why should you care?

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

- 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

$$\boldsymbol{HALT} = \{ \langle M, w \rangle \mid M \textbf{ is a TM that halts on } w \}$$

- How hard is this problem to solve?
- How do we know?

$HALT \notin \mathbf{R}$

- ***Claim:*** $HALT \notin \mathbf{R}$.
- 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();
    }
}
```

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: $HALT \notin \mathbf{R}$.

Proof: By contradiction; assume that $HALT \in \mathbf{R}$. 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.

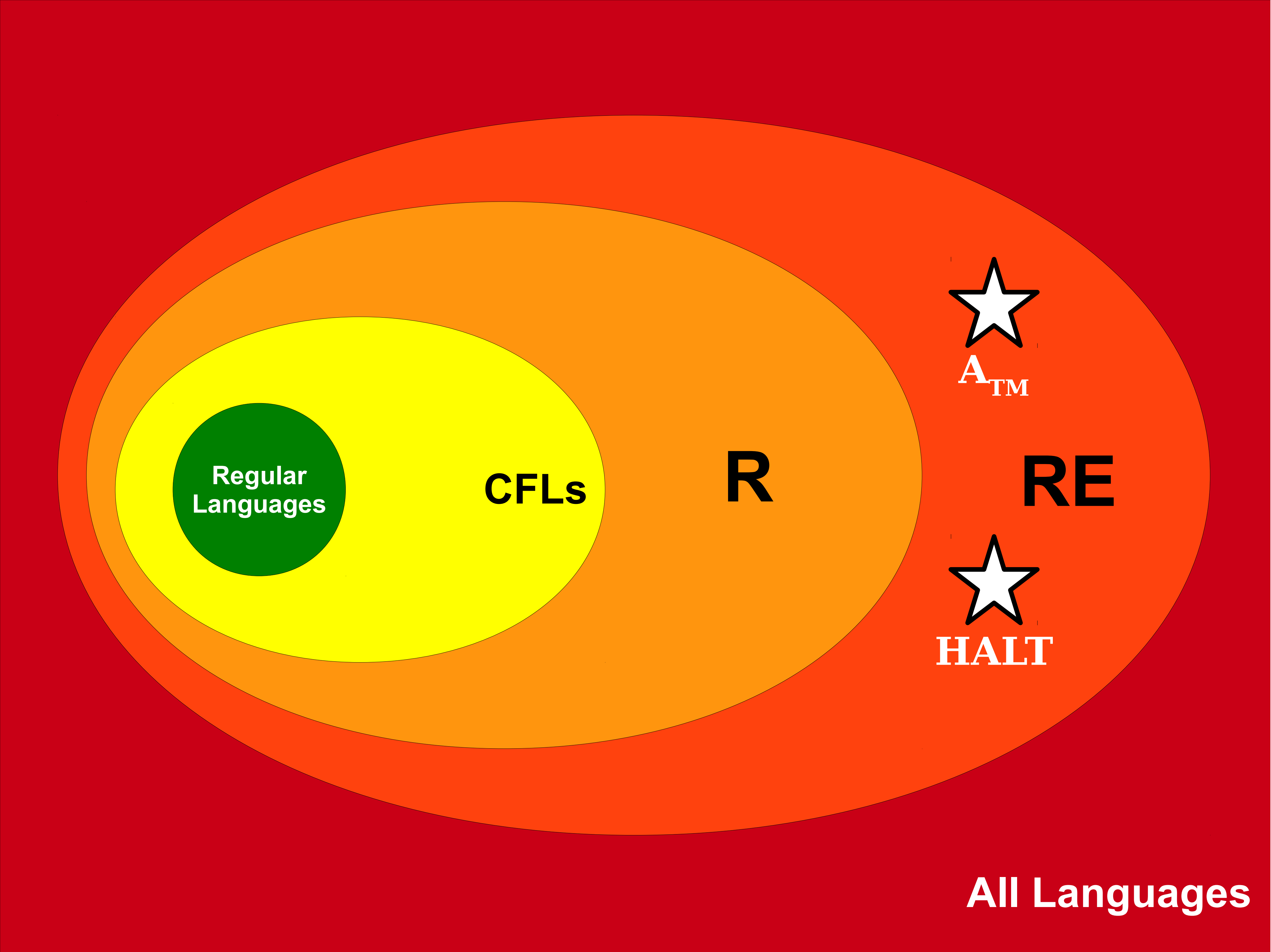
In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $HALT \notin \mathbf{R}$. ■

$\mathit{HALT} \in \mathbf{RE}$

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

So What?

- These problems might not seem all that exciting, so who cares if we can't solve them?
- Turns out, this same line of reasoning can be used to show that some very important problems are impossible to solve.

Secure Voting

- Suppose that you want to make a voting machine for use in an election between two parties.
- Let $\Sigma = \{$`r`, `d`$\}$. A string in w corresponds to a series of votes for the candidates.
- 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`."

Secure Voting

- A voting machine is a program that accepts a string of `r`'s and `d`'s, then reports whether person `r` won the election.
- Formally: a TM M is a secure voting machine if $\mathscr{L}(M) = \{ w \in \{\mathtt{r}, \mathtt{d}\}^* \mid w \text{ has more } \mathtt{r}\text{'s than } \mathtt{d}\text{'s} \}$
- ***Question:*** Given a TM that someone claims is a secure voting machine, could we automatically check whether it actually is a secure voting machine?
  - That is, is there an *algorithm* we can follow to determine this?

Secure Voting

- The ***secure voting problem*** is the following:

Given a TM *M*, is the language of *M*
$\{ w \in \{\texttt{r}, \texttt{d}\}^* \mid w$ has more `r`'s than `d`'s $\}$?

- ***Claim:*** This problem is not decidable – there is no algorithm that can check an arbitrary TM to verify that it's a secure voting machine!

Secure Voting

- Suppose that the secure voting problem is decidable. Then we could write a function

  ```
  bool isSecureVotingMachine(string program)
  ```

  that would accept as input a program and return whether or not it's a secure voting machine.
- As you might expect, this lets us do Cruel and Unusual Things...

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

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

   bool answer = countRs(input) > countDs(input);
   if (isSecureVotingMachine(me)) answer = !answer;

   if (answer) accept();
   else reject();
}
```

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

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

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

answer **true**

This program is a secure voting machine.

It should accept precisely the strings in $\{ w \in \{\mathbf{r}, \mathbf{d}\}^* \mid w$ has more **r**'s than **d**'s $\}$.

What happens if we run it on **ddd**?

What happens if…

… this program is a secure voting machine?
then it's not a secure voting machine!

```
bool isSecureVotingMachine(string program) {
    /* … some implementation … */
}

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

    bool answer = countRs(input) > countDs(input);
    if (isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

answer **false**

This isn't a secure voting machine, so its language is not
{ w ∈ {**r**, **d**}* | w has more **r**'s than **d**'s }.

What if we run it on a string ***not*** in
{ w ∈ {**r**, **d**}* | w has more **r**'s than **d**'s }?

What happens if…

… this program is a secure voting machine?
then it's not a secure voting machine!

… this program is not a secure voting machine?
then it is a secure voting machine!

Theorem: The secure voting problem is undecidable.

Proof: By contradiction; assume that 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 method `isSecureVotingMachine` that, given as input the source code of a program, returns true if the program is a secure voting machine and false otherwise.

Given this, we could then construct the following program P:

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

    bool answer = (countRs(input) > countDs(input));
if (!isSecureVotingMachine(me)) answer = !answer;

    if (answer) accept();
    else reject();
}
```

Now, either P is a secure voting machine or it isn't. If P is a secure voting machine, then `isSecureVotingMachine(me)` will return true. Therefore, when P is run, it will determine whether w has more r's than d's, flip the result, and accept strings with at least as many d's as r's and reject strings with more r's than d's. Thus, P is not a secure voting machine. On the other hand, if P is not a secure voting machine, then `isSecureVotingMachine(me)` will return false. Therefore, when P is run, it will accept all strings with at least as many r's as d's and reject all other strings, and so P is a secure voting machine.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, the secure voting problem is undecidable. ■

Interpreting this Result

- The previous argument 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.
- So what can we do?
  - Design algorithms that work in *some*, but not *all* cases. (This is often done in practice.)
  - Fall back on human verification of voting machines. (We do that too.)
  - Carry a healthy degree of skepticism about electronic voting machines. (Then again, did we even need the theoretical result for this?)

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.