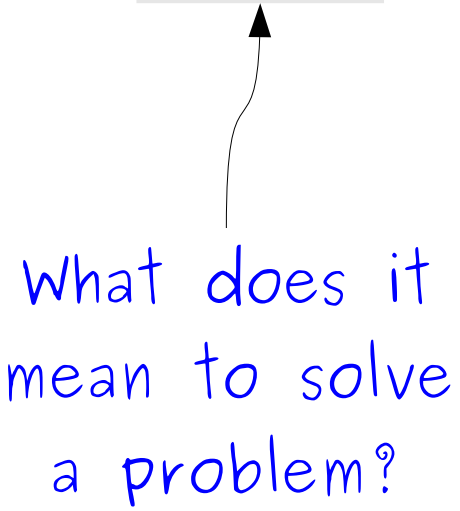
Unsolvable Problems

Part One

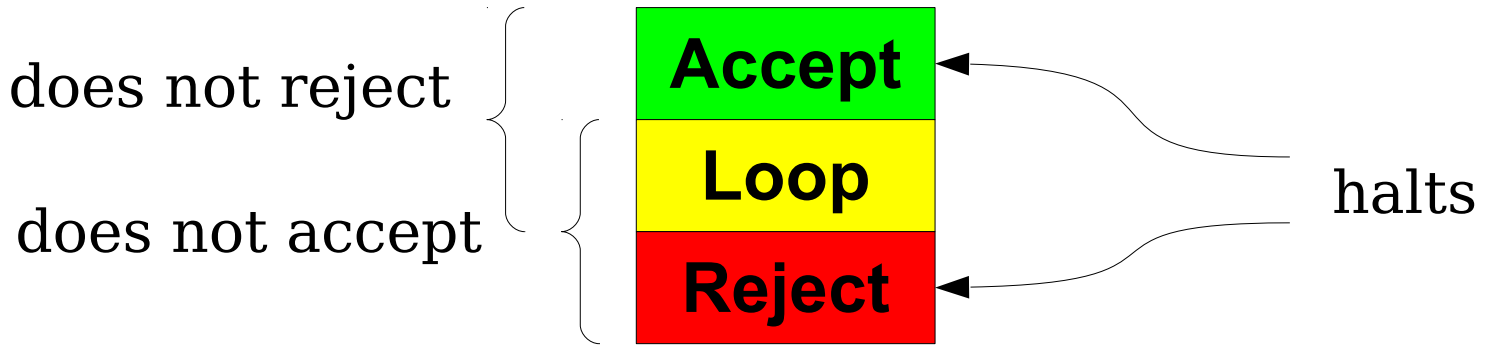
Recap from Last Time

What problems can we solve with a computer?



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:

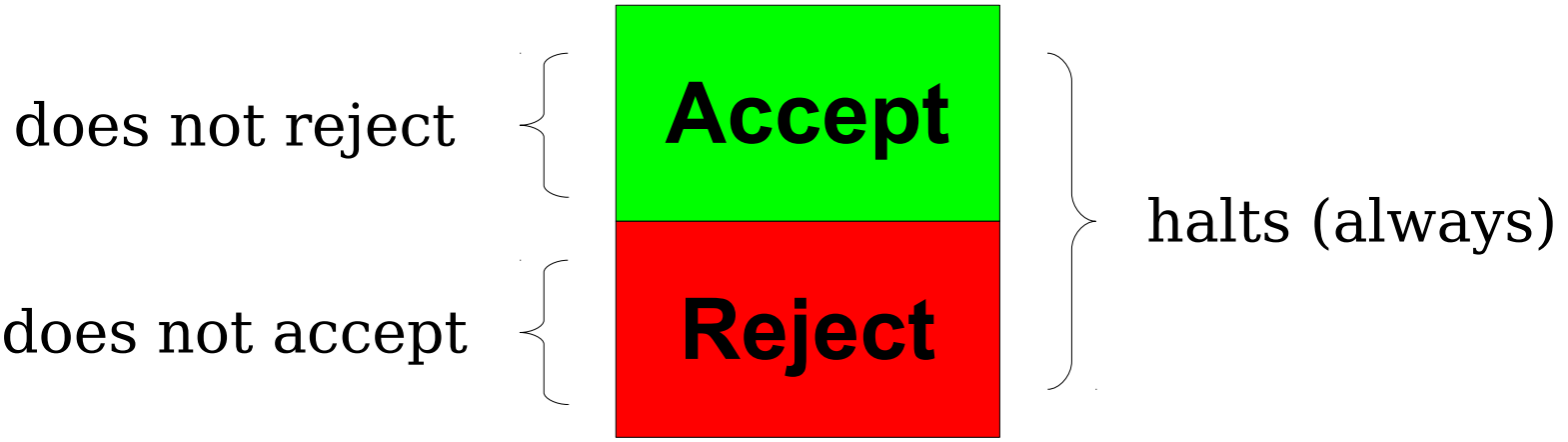
$$\mathscr{L}(M) = \{ w \in \Sigma^* \mid M \text{ accepts } w \}$$

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

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

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

Self-Referential Programs

- With sufficient effort, it's possible to build programs that operate over their own source code.
- ***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.

New Stuff!

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

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.

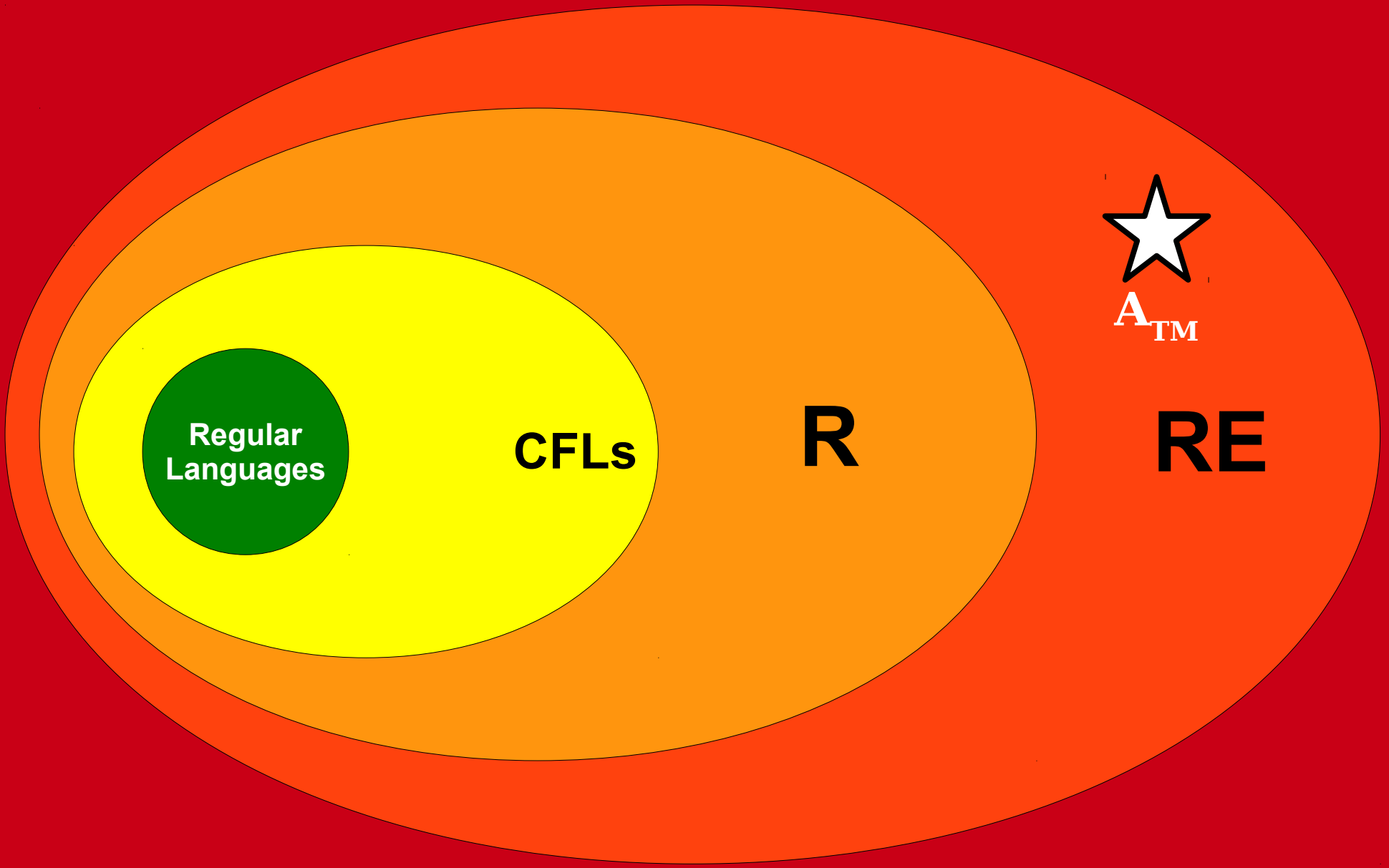
Given this, we could then construct the following TM:

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
RE
A_{TM}
All Languages

What Does This Mean?

- In one fell swoop, we've proven that
  - A_{TM} is ***undecidable***; there is no 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 algorithm worked. This means that we can rule out all possible algorithms!

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

- 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 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?

$HALT \in \mathbf{RE}$

- ***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;
    }
}
```

$HALT \notin \mathbf{R}$

- ***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();
   }
}
```

What happens if...

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

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

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

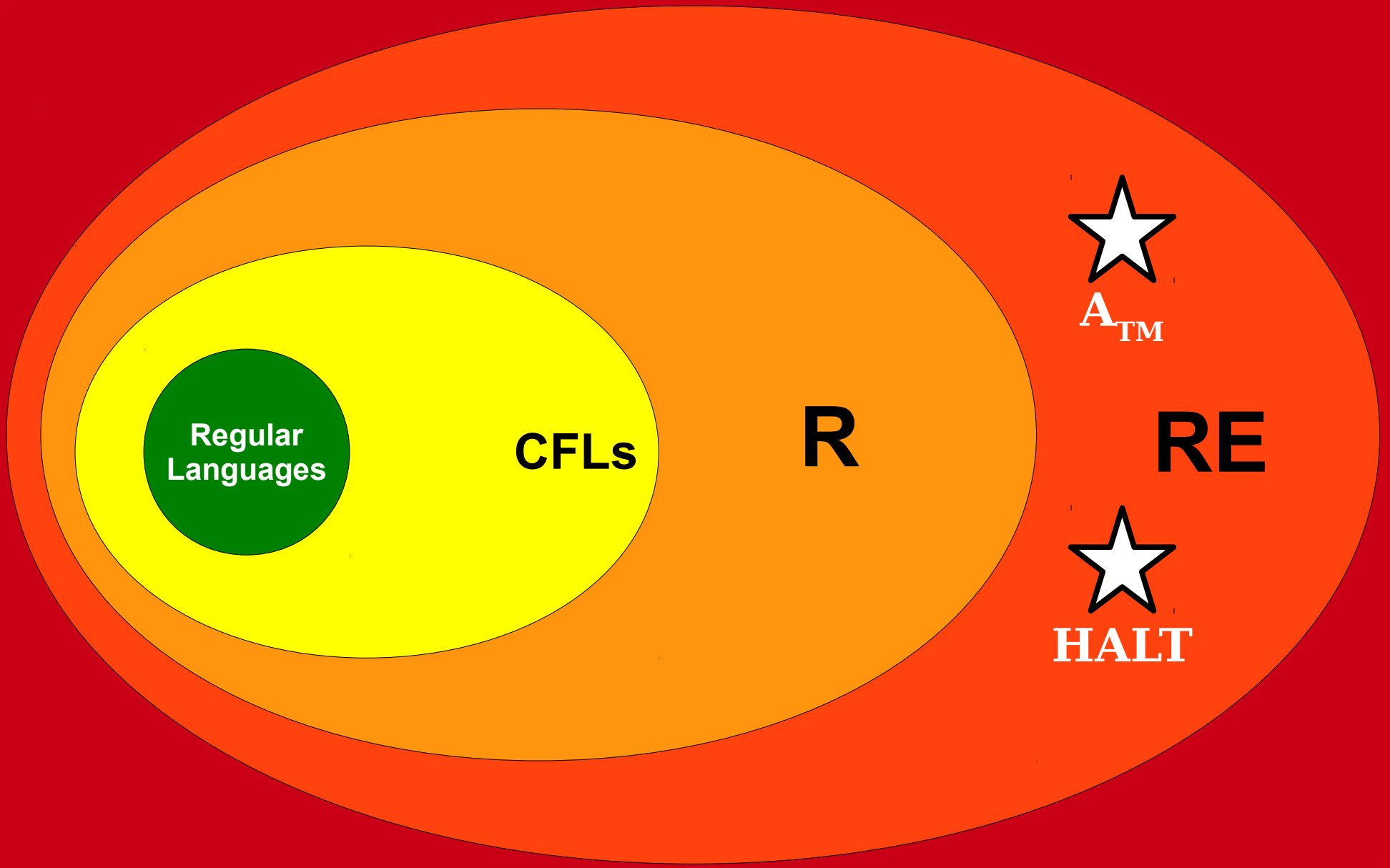
Proof: By contradiction; assume that $HALT \in \mathbf{R}$. Then there is some decider D for $HALT$. If this machine is given any TM/string pair, it will then determine whether the TM halts on the string and report back the answer.

Given this, we could then construct the following TM:

M = "On input w:
Have M obtain its own description, $\langle M \rangle$.
Run D on $\langle M, w \rangle$ 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, $HALT \notin \mathbf{R}$. ■


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 voting machine if $\mathscr{L}(M) = \{ w \in \{\texttt{r}, \texttt{d}\}^* \mid w$ has more `r`'s than `d`'s $\}$
- ***Question:*** Given a TM that claims to be a voting machine, could we check whether it actually is a fair voting machine?

Secure Voting

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

  Given a TM *M*, is the language of *M*
  { *w* ∈ {r, d}* | *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 actualAnswer =
       countRs(input) > countDs(input);

   if (isSecureVotingMachine(me)) {
      return !actualAnswer;
   } else {
      return actualAnswer;
   }
}
```

What happens if...

... this program is a secure voting machine?

It's not a secure machine!

... this program is not a secure voting machine?

It is a secure voting machine!

This previous example is not contrived!

This is a problem we really would like to be able to solve!

Yet it's provably impossible!

Time-Out for Announcements!

Second Midterm Exam

- The second midterm exam is ***tonight*** from 7PM – 10PM.
- Room assignments divvied up by last (family) name:
  - **`Aga – Ven`**: Go to Hewlett 200.
  - **`Ver – Zhe`**: Go to 370-370
- Cumulative exam. Focuses on material from Lectures 10 – 18 and PS4 – 6.
  - Main focus is on graphs, the pigeonhole principle, induction, finite automata, regular expressions, and regular languages.
  - Will *not* test the Myhill-Nerode theorem, CFGs, or Turing machines.
  - Since everything builds on itself in this course, the exam may require topics from earlier in the course.
- As before, you get one double-sided, 8.5” × 11” sheet of notes during the exam.

Problem Set Eight

- There's a problem set out right now (PS8) that's due on Friday. It explores TMs and the limits of computation.
- ***Clarification on the late policy:*** You're welcome to use late days here. However, if you do, please submit the assignment no later than Monday of next week (November 23); otherwise we won't be able to grade things on time.
- This is the last opportunity to use late days in this class, so if you want to offload some of the work a bit, go right ahead!

Your Questions!

“What are your thoughts on taking time off during undergrad?”

I think that for some people this can be a really great and worthwhile experience. I'd recommend thinking of it this way: what would you be doing instead? If you'd spend that time doing something you think is worthwhile (traveling, working, performing, engaging on a voyage of self-discovery, etc.), then go for it! The perspective will help and we'll always be here for when you're reading to come back. Plus, a lot of these things are a lot easier to do when you're still in college then when you've graduated.

"Do you typically curve up when assigning final grades in this class, or is a 90% average on class coursework usually required for an A-?"

I never curve grades down - that just seems mean - and there's usually a pretty generous curve in this course. (I've heard that the word on the street is that this class isn't curved, but trust me, it is). Remember that problem sets are probably the least important factor to your overall grade in this class. The final will account for the biggest share, then the exams, and then the problem sets.

"Is it possible to get an internship without CS161? Since I am planning to take algorithms during the Winter or Spring, and since many companies are already starting the hiring process, I feel hopeless in getting an internship for this summer."

CS161 is definitely not required for an internship! If so, about half the undergrads here couldn't get an internship because they put it off until Junior or Senior years. ☺

CS161 is useful for helping your general problem-solving skills, and that often makes it a bit easier to get through interviews, but it's by no means a requirement. Good luck on the job market!

“I saw an article online about a new algorithm for solving the graph isomorphism problem. What is the significance of this result?”

If we have time, we'll talk about this when we get back from the break. We need a bit more context before this result will make sense. That said, it is really big news!

“How did you learn to cook?”

Trial and error and YouTube. ☺

Plus, it's super hard to make something that's truly inedible. The worst case is you end up with something that wasn't worth the effort you put into it.

Give it a try! Hit me up for some good starter recipes.

Back to CS103!

A Quick Note

A Quick Note

- In the proofs we've just seen, we show that an *entire problem* is undecidable by giving one self-referential program that would break a given decider.
- However, that doesn't mean that the language is "decidable everywhere except in one case."
- ***Theorem:*** If a language L is undecidable, then given any attempt at a decider D for L, D will be wrong infinitely many times.
- Similar to Cantor's theorem: the theorem shows that a function isn't a bijection by finding one subset that isn't covered, but any bijection will miss *way* more than one string on any set of any reasonable size.

Beyond **R**

What exactly is the class **RE**?

RE, Formally

- Recall that the class **RE** is the class of all recognizable languages:

  $$\mathbf{RE} = \{ L \mid \text{there is a TM } M \text{ where } \mathscr{L}(M) = L \}$$

- I mentioned earlier a good intuition for **RE**: a language L is in **RE** if, when you know for certain that some string w belongs to L, there's a mechanical way to confirm this.
- Why exactly is that?
- More generally, what is the class **RE**?

Key Intuition:

A language L is in **RE** if for every $w \in L$, there is some way to prove that $w \in L$.

We haven't yet justified why this intuition is correct. That's what we're about to do next!

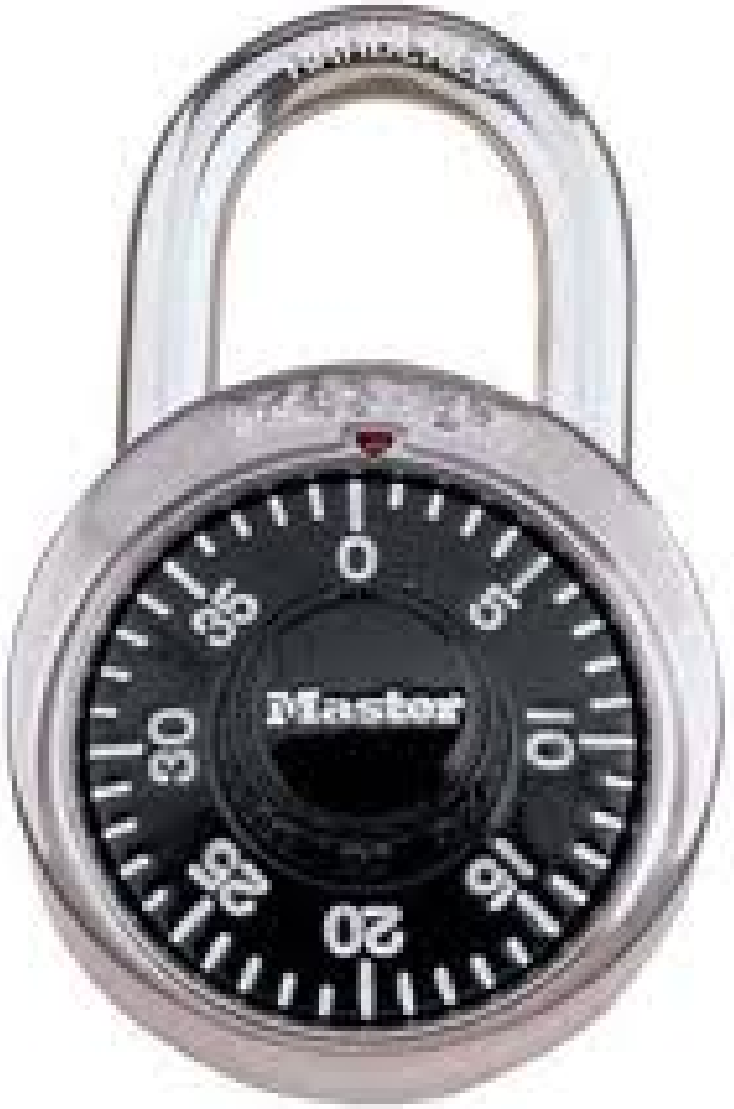
Verifiers

- A ***verifier*** for a language L is a TM V with the following properties:
  - V halts on all inputs.
  - For any string $w \in \Sigma^*$, the following is true:

$$\boldsymbol{w \in L \leftrightarrow \exists c \in \Sigma^*.\ V \text{ accepts } \langle w, c \rangle}$$

- Intuitively, what does this mean?

Intuiting Verifiers



Question:
Can this lock be opened?

Verifiers

- A ***verifier*** for a language L is a TM V with the following properties:
  - V halts on all inputs.
  - For any string $w \in \Sigma^*$, the following is true:

$$\boldsymbol{w \in L \quad \leftrightarrow \quad \exists c \in \Sigma^*.\ V \text{ accepts } \langle w, c \rangle}$$

- Some notes about V:
  - If V accepts $\langle w, c \rangle$, then we're guaranteed $w \in L$.
  - If V does not accept $\langle w, c \rangle$, then either
    - $w \in L$, but you gave the wrong c, or
    - $w \notin L$, so no possible c will work.

Verifiers

- A ***verifier*** for a language L is a TM V with the following properties:
  - V halts on all inputs.
  - For any string $w \in \Sigma^*$, the following is true:

$$\boldsymbol{w \in L \quad \leftrightarrow \quad \exists c \in \Sigma^*.\ V \text{ accepts } \langle w, c \rangle}$$

- Some notes about V:
  - If $w \in L$, a string c for which V accepts $\langle w, c \rangle$ is called a ***certificate*** for w.
  - V is required to halt, so given any potential certificate c for w, you can check whether the certificate is correct.

Verifiers

- A ***verifier*** for a language L is a TM V with the following properties:
  - V halts on all inputs.
  - For any string $w \in \Sigma^*$, the following is true:

$$\boldsymbol{w \in L \quad \leftrightarrow \quad \exists c \in \Sigma^*.\ V \text{ accepts } \langle w, c \rangle}$$

- Some notes about V:
  - Notice that $\mathscr{L}(V) \neq L$. *(Good question: what <u>is</u> $\mathscr{L}(V)$?)*
  - The job of V is just to check certificates, not to decide membership in L.

Some Verifiers

- Let L be the following language:

  $L = \{ \langle G, w \rangle \mid G \text{ is a CFG that generates } w \}$

- Let's see how to build a verifier for L.
- A certificate for a grammar G string w should convince us that G accepts w. What kind of information would help us with that?
- One option: Let the certificate be a possible derivation of w from the start symbol.
- Our verifier then just needs to check whether the derivation is valid.

Some Verifiers

- Let L be the following language:

  $L = \{ \langle G, w \rangle \mid G \text{ is a CFG that generates } w \}$

- Here is one possible verifier for L:

> V = "On input $\langle G, w, c \rangle$, where G is a CFG:
> Check whether c is a valid derivation of w
> from the start symbol of G.
> If so, accept. If not, reject."

- If the certificate is a correct derivation, we know for a fact that G can generate w.
- If not, we can't tell whether we got a bad certificate or whether G doesn't generate w.

Some Verifiers

- Let L be the following language:

$$L = \{ \langle n \rangle \mid n \in \mathbb{N} \text{ and the hailstone sequence terminates for } n \}$$

- Let's see how to build a verifier for L.
- A certificate for $\langle n \rangle$ should convince us that the hailstone sequence terminates for n. A bad certificate shouldn't leave us running forever.
- A thought: if the hailstone sequence terminates for n, then it has to terminate in some number of steps.
- Let the certificate be that number of steps.

Some Verifiers

- Let L be the following language:

$$L = \{ \langle n \rangle \mid n \in \mathbb{N} \text{ and the hailstone sequence terminates for } n \}$$

V = "On input $\langle n, k \rangle$, where $n, k \in \mathbb{N}$.
Check that $n \neq 0$.
Run the hailstone sequence, starting at n, for at most k steps.
If after k steps we reach 1, accept.
Otherwise, reject."

- Do you see why $\langle n \rangle \in L$ iff there is some k such that V accepts $\langle n, k \rangle$?

What languages are verifiable?

Theorem: If L is a language, then there is a verifier for L if and only if $L \in \mathbf{RE}$.