Turing Machines

Part Two

Outline for Today

- ***The Church-Turing Thesis***
  - Just how powerful are TMs?
- ***What Does it Mean to Solve a Problem?***
  - Rethinking what “solving” a problem means, and two possible answers to that question.

Recap from Last Time

Turing Machines

- A ***Turing machine*** is a program that controls a tape head as it moves around an infinite tape.
- There are six commands:
  - **`Move`** *direction*
  - **`Write`** *symbol*
  - **`Goto`** *label*
  - **`Return`** *boolean*
  - **`If`** *symbol command*
  - **`If Not`** *symbol command*
- Despite their limited vocabulary, TMs are surprisingly powerful.

A Sample Turing Machine

- Here's a sample TM.
- It receives inputs over the alphabet $\Sigma = \{\texttt{a}, \texttt{b}\}$.
- What strings does this TM accept?
- Can you write a regex that matches precisely the strings this TM accepts?

```
Start:
    If Not 'a' Return False

Loop:
    Move Right
    If Not Blank Goto Loop
    Move Left
    Move Left
    If Not 'b' Return False
    Return True
```

What Can We Do With a TM?

- Last time, we saw TMs that
  - check if a string has the form $\mathtt{a}^n\mathtt{b}^n$,
  - check if a string has the same number of **a**'s and **b**'s,
  - sort a string of **a**'s and **b**'s,
  - check if a string's length is a Fibonacci number,
  - convert the decimal number n into the string $\mathtt{a}^n$, and
  - check if a decimal number is a Fibonacci number.
- This hints at the idea that TMs might be more powerful than they look.

New Stuff!

Main Questions for Today:

Just how powerful are Turing machines?

What problems can you solve with a computer?

Real and “Ideal” Computers

- A real computer has memory limitations: you have a finite amount of RAM, a finite amount of disk space, etc.
- However, as computers get more and more powerful, the amount of memory available keeps increasing.
- An ***idealized computer*** is like a regular computer, but with unlimited RAM and disk space. It functions just like a regular computer, but never runs out of memory.

Theorem: Turing machines are equal in power to idealized computers. That is, any computation that can be done on a TM can be done on an idealized computer and vice-versa.

Key Idea: Two models of computation are equally powerful if they can simulate each other.

Simulating a TM

- The individual commands in a TM are simple and perform only basic operations:

```
Move   Write   Goto   Return   If
```

- The memory for a TM can be thought of as a string with some number keeping track of the current index.
- To simulate a TM, we need to
  - see which line of the program we're on,
  - determine what command it is, and
  - simulate that single command.
- ***Claim:*** This is reasonably straightforward to do on an idealized computer.
  - My "core" logic for the TM simulator is under fifty lines of code, including comments.

Simulating a TM

- Because a computer can simulate each individual TM instruction, a computer can do anything a TM can do.
- ***Key Idea:*** Even the most complicated TM is made out of individual instructions, and if we can simulate those instructions, we can simulate an arbitrarily complicated TM.

Simulating a Computer

- Programming languages provide a set of simple constructs.
  - Think things like variables, arrays, loops, functions, classes, etc.
- You, the programmer, then combine these basic constructs together to assemble larger programs.
- ***Key Idea:*** If a TM is powerful enough to simulate each of these individual pieces, it's powerful enough to simulate anything a real computer can do.

What We've Seen

- We've seen TMs use loops to solve problems.
  - Our $\{ \mathbf{a}^n\mathbf{b}^n \mid n \in \mathbb{N} \}$ TM repeatedly pulls off the first and last character from the string.
  - Our sorting TM repeatedly finds **ba** and replaces it with **ab**.
- In some sense, the existence of `Goto` and labels means that TMs have loops.
- Hopefully, it's not too much of a stretch to think that TMs can do `while` loops, `for` loops, etc.

What We've Seen

- We've seen TMs that perform basic arithmetic.
  - We can check if two numbers are equal.
  - We can check if a number is a Fibonacci number.
- Hopefully, it's not too much of a stretch to believe we could also do addition and subtraction, compute powers of numbers, do ceilings and floors, etc.

What We've Seen

- We've seen TMs that maintain variables.
  - You can think of our TM for $\{ \mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N} \}$ as storing two variables – one that counts a number of **a**'s, and one that counts a number of **b**'s.
  - Our TM for Fibonacci numbers kinda sorta ish tracks the last two Fibonacci numbers, plus the length of the input string.
- It's a bit larger of a jump to make, but hopefully you're comfortable with the idea that TMs, in principle, can maintain variables.

What We've Seen

- We've seen TMs with helper functions.
  - We saw how to check for equal numbers of **a**'s and **b**'s by first sorting the string, then checking of the string has the form $\mathbf{a}^n\mathbf{b}^n$.
  - We can check if a decimal number is a Fibonacci number by converting it to unary, then running our unary Fibonacci checker.
- Hopefully you're comfortable with the idea that a TM could have multiple "helper functions" that work together to solve some larger problem.

What Else Can TMs Do?

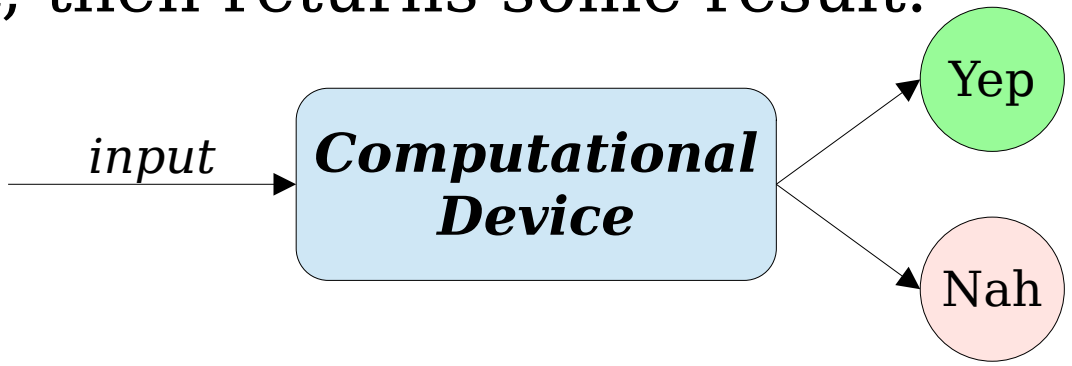
- Maintain strings and arrays.
  - Store their elements separated with some special separator character.
- Support pointers.
  - Maintain an array of what's in memory, where each item is tagged with its "memory address."
- Support function call and return.
  - It's hard, but you can do this if you can do helper functions and variables.

A CS107 Perspective

- Internally, computers execute by using basic operations like
  - simple arithmetic,
  - memory reads and writes,
  - branches and jumps,
  - register operations,
  - etc.
- Each of these are simple enough that they could be simulated by a Turing machine.

A Leap of Faith

- ***Claim:*** A TM is powerful enough to simulate any computer program that gets an input, processes that input, then returns some result.



- The resulting TM might be colossal, or really slow, or both, but it would still faithfully simulate the computer.
- We're going to take this as an article of faith in CS103. If you curious for more details, come talk to me after class.

Can a TM Work With...

"cat pictures?"

Sure! A picture is just a 2D array of colors, and a color can be represented as a series of numbers.

Can a TM Work With…

~~“cat pictures?”~~

“cat videos?”

If you think about it, a video is just a series of pictures!

Can a TM Work With…

"music?"

Sure! Music is encoded as a compressed waveform. That's just a list of numbers.

"deep learning?"

Sure! That's just applying a bunch of matrices and nonlinear functions to some input.

Just how powerful ***are*** Turing machines?

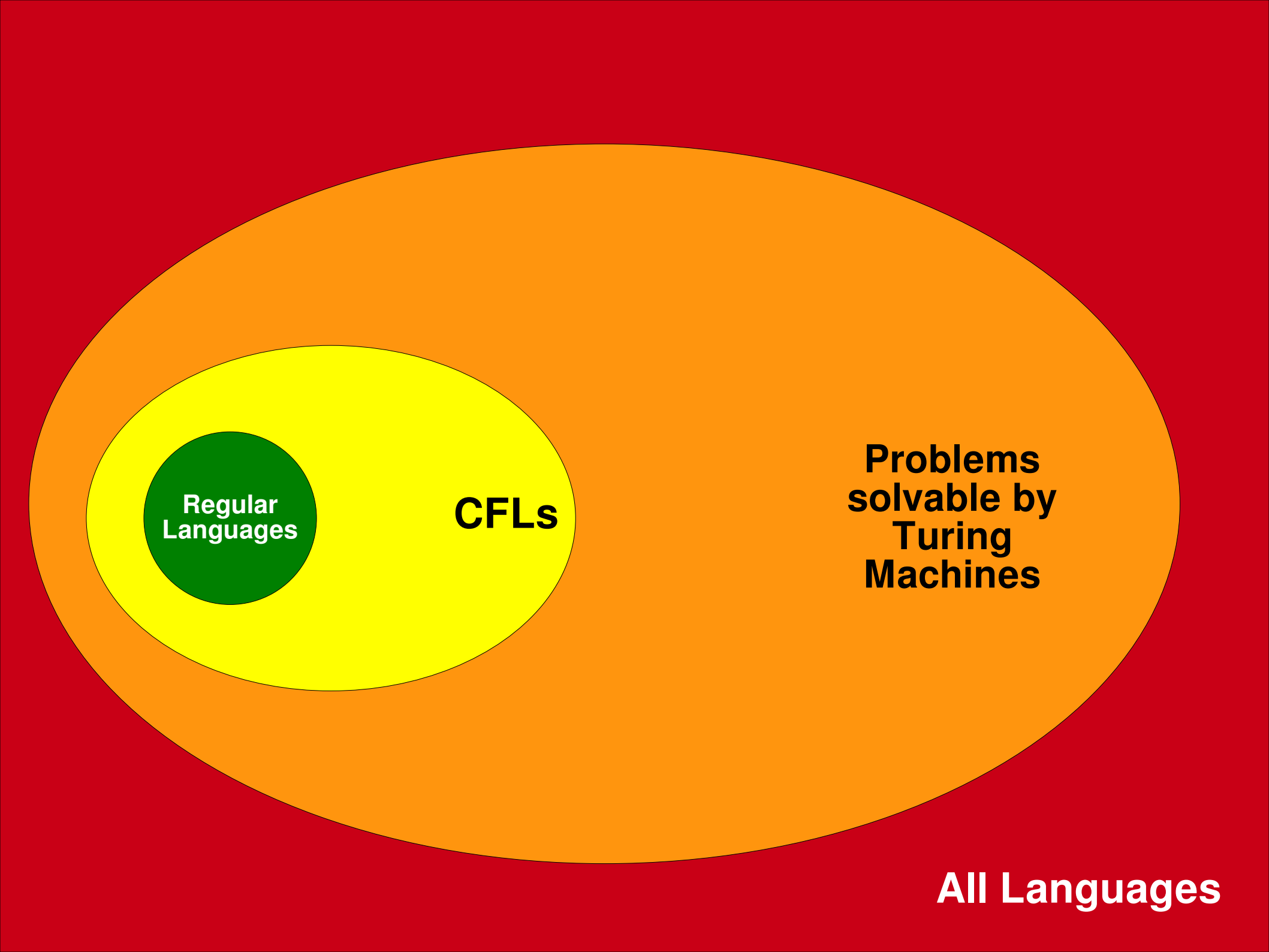
Effective Computation

- An ***effective method of computation*** is a form of computation with the following properties:
  - The computation consists of a set of steps.
  - There are fixed rules governing how one step leads to the next.
  - Any computation that yields an answer does so in finitely many steps.
  - Any computation that yields an answer always yields the correct answer.
- This is not a formal definition. Rather, it's a set of properties we expect out of a computational system.

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


Regular Languages
CFLs
Problems solvable by Turing Machines
All Languages

TMs and Computation

- Because Turing machines have the same computational powers as regular computers, we can (essentially) reason about Turing machines by reasoning about actual computer programs.
- Going forward, we're going to switch back and forth between TMs and computer programs based on whatever is most appropriate.
- In fact, our eventual proofs about the existence of impossible problems will involve a good amount of pseudocode. Stay tuned for details!

Time-Out for Announcements!

Problem Set 8

- Problem Set Seven was due at 2:30PM today.
- Problem Set Eight goes out today. It's due next Friday at 2:30PM.
  - Construct context-free grammars and explore their expressive power.
  - Probe the interface between regular and nonregular languages.
  - Tinker with TMs and what it's like to build all computation from smaller pieces.
- You know the drill: come talk to us if you have any questions, and let us know what we can do to help out.

Your Questions

“I ended up in the bottom quintile on midterm 2, significantly worse than I did on midterm 1. Any tips for not taking this as a sign that I should so something other than CS? I know that this sounds like an emotional overreaction, but I also really bricked a technical interview that I spent a lot of time preparing for earlier this week. I'm constantly worried that I'm worse at this than everyone else, and there seems to be enough evidence to back that worry up. :(“

For starters, it sounds like you've a rough week, and I'm sorry to hear that.

As much as is possible, distinguish between "this exam didn't go well for me" and "I am objectively not good at this." Everyone has the experience of an interview that didn't go well and an exam that didn't go well. It's not a fun experience (can confirm). Do the things you need to do to dust off and get back to a good frame of mind, then look back and do the analysis to see what you need to learn and what you can do differently for next time. You're always learning and picking up new skills - try to focus that growth in areas that have the biggest difference.

As for how other folks are doing: by definition half of the students in CS103 will end up in the bottom half of the course. Relative positioning is much less important than you might think. What matters in the long run is what you've learned and what you can do going forward, not whether other people can do it better than you.

Remember that...

... you are ***smart***,
... you are ***creative***,
... you are ***competent***, and that
... you are ***hardworking***.

The material in CS103 is challenging. It's legitimately tough. We're confident you can do this. Let us know how we can help.

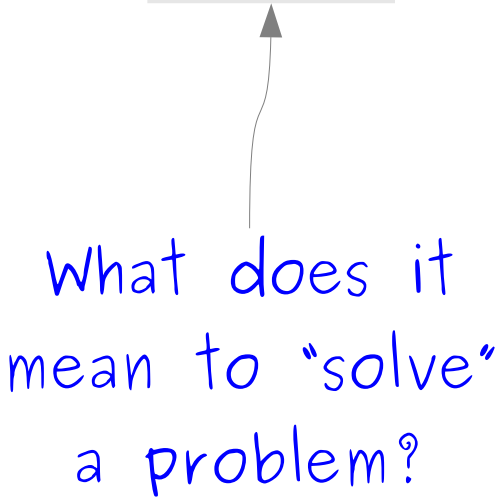
“Favorite movies?“

- “Black Orpheus:” Starts strong and never lets up.
- “The Darjeeling Limited:” Wonderful escapist family drama.
- “The Good, The Bad, and The Ugly:” It’s a classic for a reason.
- “Rules of the Game:” 1939 pre-war social commentary that’s still relevant.
- “12 Angry Men:” Proof that you can have incredible drama in one room.
- “Le Samourai:” Incredible character study .
- “Brazil:” Dystopian science fiction story that feels alarmingly recognizable.
- “Fantasia:” Sit back, relax, listen, and unwind.
- “Seven Samurai,” “Yojimbo,” “The Hidden Fortress,” “Ikiru:” There’s a reason Kurosawa is such a legend. These films influenced so many future ones.
- “City of God:” Funny, darkly serious, terrifying, and well-acted.
- “There Will Be Blood:” Daniel Day Lewis’s performance will stick with you for a long time after watching this one.
- “Lawrence of Arabia:” Sweeping epic in the best sense of the word.
- “Amelie:” Beautiful movie, beautiful soundtrack.
- “Unforgiven:” Deconstruction of the Western archetype.
- “Little Miss Sunshine:” One of the funniest movies I’ve ever seen.
- “My Neighbor Totoro:” The most wholesome movie I’ve ever seen.
- “Back to the Future:” Why is this movie so good?

Back to CS103!

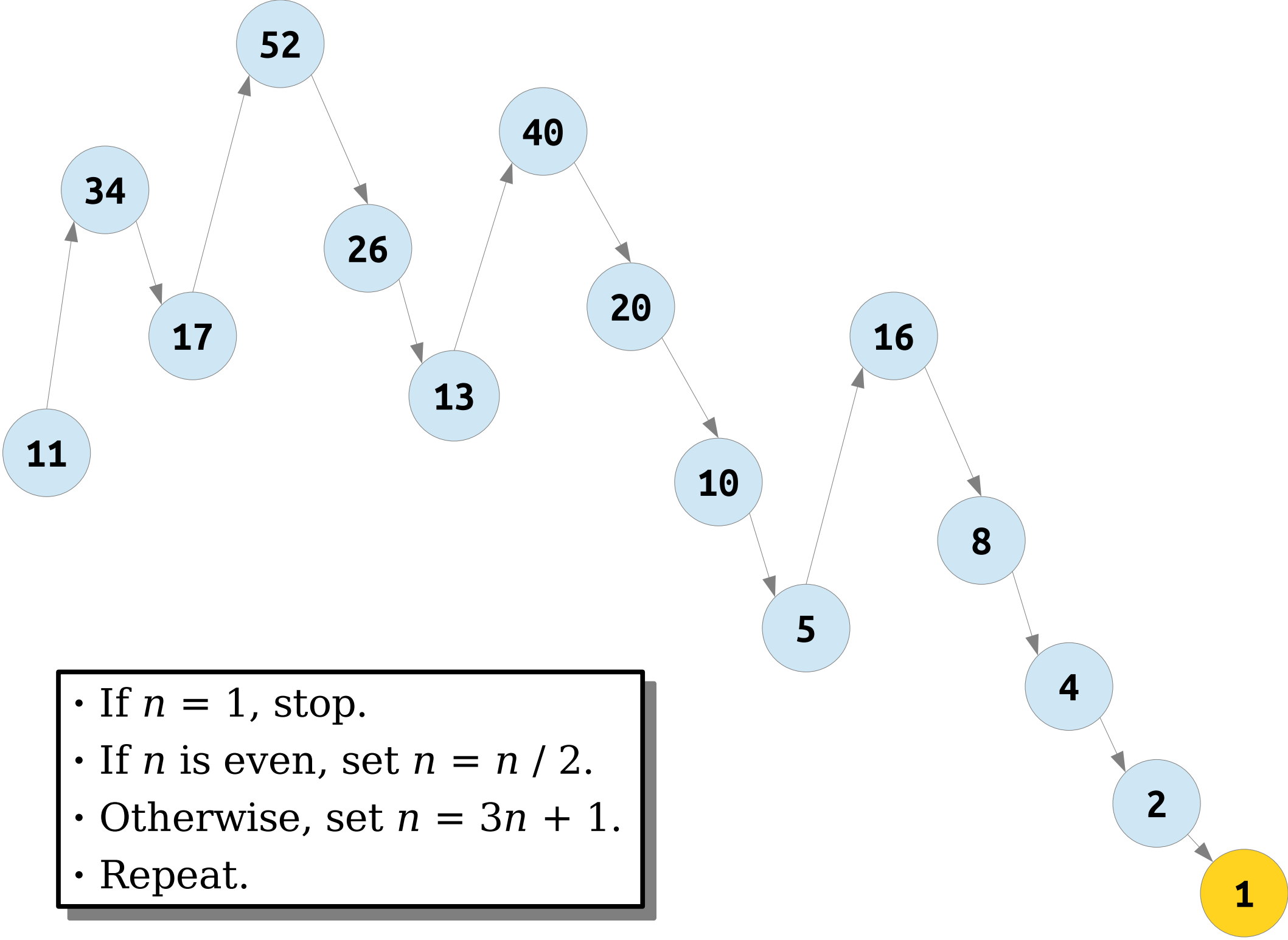
Decidability and Recognizability

What problems can we solve with a computer?



The Hailstone Sequence

- Consider the following procedure, starting with some $n \in \mathbb{N}$, where $n > 0$:
  - If $n = 1$, you are done.
  - If n is even, set $n = n / 2$.
  - Otherwise, set $n = 3n + 1$.
  - Repeat.
- ***Question:*** Given a natural number $n > 0$, does this process terminate?



- If $n = 1$, stop.
- If n is even, set $n = n / 2$.
- Otherwise, set $n = 3n + 1$.
- Repeat.

The Hailstone Sequence

- Consider the following procedure, starting with some $n \in \mathbb{N}$, where $n > 0$:
  - If $n = 1$, you are done.
  - If n is even, set $n = n / 2$.
  - Otherwise, set $n = 3n + 1$.
  - Repeat.
- Does the Hailstone Sequence terminate for…
  - $n = 5$?
  - $n = 20$?
  - $n = 7$?
  - $n = 27$?

The Hailstone Sequence

- Let $\Sigma = \{\texttt{a}\}$ and consider the language

  $L = \{ \texttt{a}^n \mid n > 0$ and the hailstone sequence terminates for $n \}$.

- Could we build a TM for L?

The Hailstone Turing Machine

- We can build a TM that works as follows:
  - If the input is ε, reject.
  - While the string is not **a**:
    - If the input has even length, halve the length of the string.
    - If the input has odd length, triple the length of the string and append a **a**.
  - Accept.

Does this Turing machine accept all nonempty strings?

The Collatz Conjecture

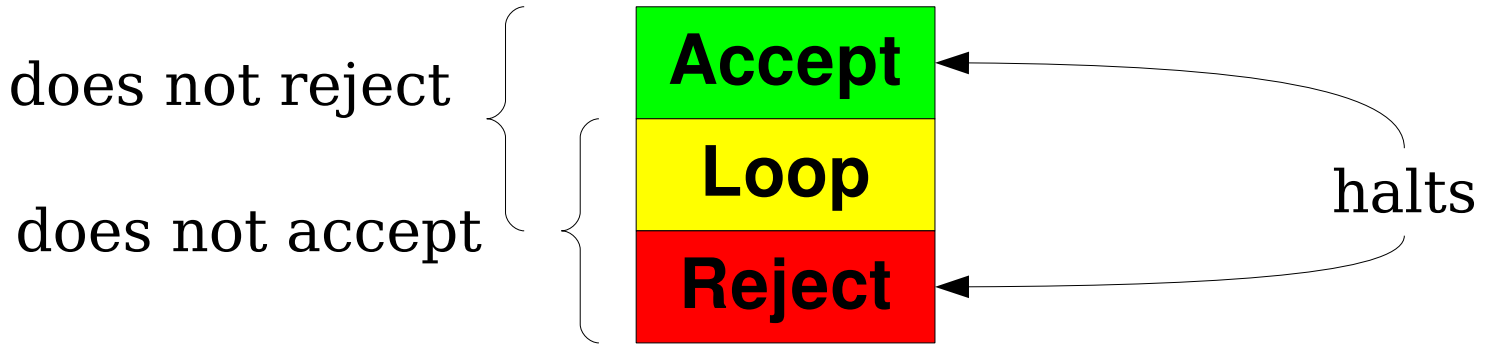
- It is *unknown* whether this process will terminate for all natural numbers.
- In other words, no one knows whether the TM described in the previous slides will always stop running!
- The conjecture (unproven claim) that th hailstone sequence always terminates is called the ***Collatz Conjecture***.
- This problem has eluded a solution for a long time. The influential mathematician Paul Erdős is reported to have said “Mathematics may not be ready for such problems.”

An Important Observation

- Unlike finite automata, which automatically halt after all the input is read, TMs keep running until they explicitly return true or return false.
- As a result, it's possible for a TM to run forever without accepting or rejecting.
- This leads to several important questions:
  - How do we formally define what it means to build a TM for a language?
  - What implications does this have about problem-solving?

Very Important Terminology

- Let M be a Turing machine.
- M ***accepts*** a string w if it returns true on w.
- M ***rejects*** a string w if it returns false on w.
- M ***loops infinitely*** (or just ***loops***) on a string w if when run on w it neither returns true nor returns false.
- M ***does not accept w*** if it either rejects w or loops 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.



Recognizers and Recognizability

- A TM M is called a ***recognizer*** for a language L over Σ if the following statement is true:

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

- If you are absolutely certain that $w \in L$, then running a recognizer for L on w will (eventually) confirm this.
  - Eventually, M will accept w.
- If you don't know whether $w \in L$, running M on w may never tell you anything.
  - M might loop on w – but you can't differentiate between "it'll never give an answer" and "just wait a bit more."
- Does that feel like "solving a problem" to you?

Recognizers and Recognizability

- The hailstone TM M we saw earlier is a recognizer for the language

$$L = \{ \texttt{a}^n \mid n > 0 \text{ and the hailstone sequence terminates for } n \}.$$

- If the sequence does terminate starting at n, then M accepts $\texttt{a}^n$.
- If the sequence doesn't terminate, then M loops forever on $\texttt{a}^n$. and never gives an answer.
- If you somehow knew the hailstone sequence terminated for n, this machine would (eventually) confirm this. If you didn't know, this machine might not tell you anything.

```
bool pizkwat(string input) {
  return false;
}
```

```
bool squigglebah(string input) {
  while (true) {
    // do nothing
  }
}
```

```
bool moozle(string input) {
  int oot = 1;
  while (input.size() != oot) {
    oot += oot;
  }
  return true;
}
```

```
bool humblegwah(string input) {
  if (input.size() % 2 != 0) return false;

  for (int i = 0; i < input.size() / 2; i++) {
    if (input[2 * i] != input[2 * i + 1]) {
      return false;
    }
  }

  return true;
}
```

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

Each of these pieces of code is a recognizer for some language.
What language does each recognizer recognize?

Recognizers and Recognizability

- Earlier this quarter you explored sums of four squares. Now, let's talk about sums of three cubes.
- Are there integers x, y, and z where...
  - $x^3 + y^3 + z^3 = 10$?
  - $x^3 + y^3 + z^3 = 11$?
  - $x^3 + y^3 + z^3 = 12$?
  - $x^3 + y^3 + z^3 = 13$?

Recognizers and Recognizability

- Surprising fact: until 2019, no one knew whether there were integers x, y, and z where

$$x^3 + y^3 + z^3 = 33.$$

- A heavily optimized computer search found this answer:

$$x = 8{,}866{,}128{,}975{,}287{,}528$$
$$y = -8{,}778{,}405{,}442{,}862{,}239$$
$$z = -2{,}736{,}111{,}468{,}807{,}040$$

- As of November 2021, no one knows whether there are integers x, y, and z where

$$x^3 + y^3 + z^3 = 114.$$

Recognizers and Recognizability

- Consider the language

$$L = \{ \mathtt{a}^n \mid \exists x \in \mathbb{Z}.\ \exists y \in \mathbb{Z}.\ \exists z \in \mathbb{Z}.\ x^3 + y^3 + z^3 = n \}$$

- Here's pseudocode for a recognizer to see whether such a triple exists:

```
for max = 0, 1, 2, …
  for x from -max to +max:
    for y from -max to +max:
      for z from -max to +max:
        if x³ + y³ + z³ = n: return true
```

- If you somehow knew there was a triple x, y, and z where $x^3 + y^3 + z^3 = n$, running this program will (eventually) convince you of this.
- If you weren't sure whether a triple exists, this recognizer might not be useful to you.

Recognizers and Recognizability

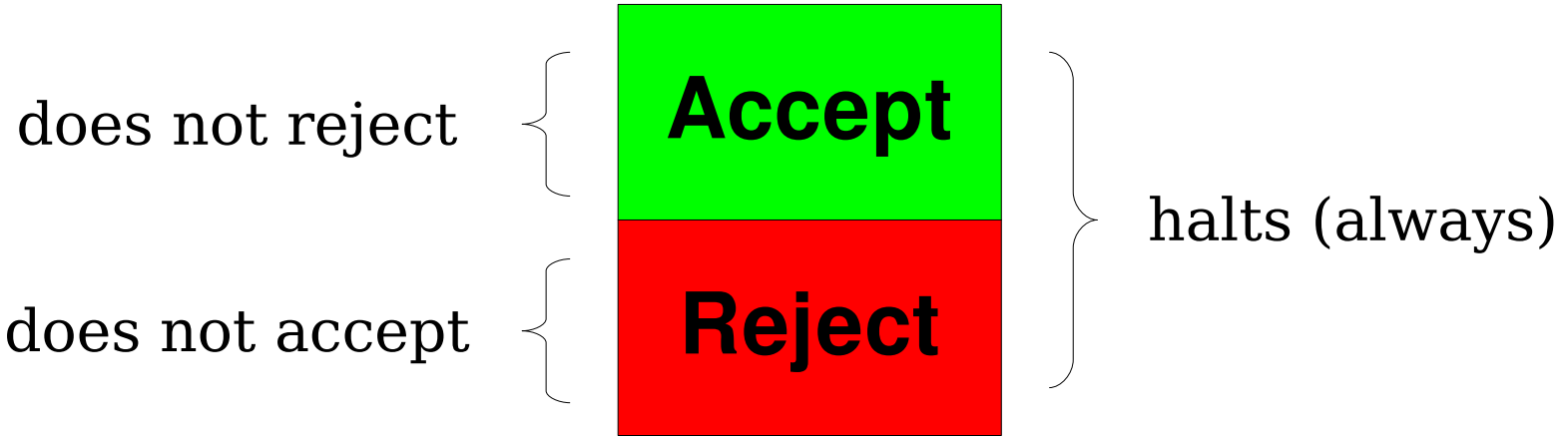
- The class **RE** consists of all recognizable languages.
- Formally speaking:

  $\mathbf{RE} = \{ L \mid L \text{ is a language and there's a recognizer for } L \}$

- You can think of **RE** as "all problems with yes/no answers where "yes" answers can be confirmed by a computer."
  - Given a recognizable language L and a string $w \in L$, running a recognizer for L on w will eventually confirm $w \in L$.
  - The recognizer will never have a "false positive" of saying that a string is in L when it isn't.
- This is a "weak" notion of solving a problem.
- Is there a "stronger" one?

Deciders and Decidability

- Some, but not all, TMs have the following property: the TM halts on all inputs.
- If you are given a TM M that always halts, then for the TM M, the statement "M does not accept w" means "M rejects w."



Deciders and Decidability

- A TM M is called a ***decider*** for a language L over Σ if the following statements are true:

$$\forall w \in \Sigma^*.\ M \text{ halts on } w.$$

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

- In other words, M accepts all strings in L and rejects all strings not in L.
- In other words, M is a recognizer for L, and M halts on all inputs.
- If you aren't sure whether $w \in L$, running M on w will (eventually) give you an answer to that question.

Deciders and Decidability

- The hailstone TM M we saw earlier is a ***recognizer*** for the language

$$L = \{ \mathtt{a}^n \mid n > 0 \text{ and the hailstone sequence terminates for } n \}.$$

- If the hailstone sequence terminates for n, then M accepts $\mathtt{a}^n$. If it doesn't, then M does not accept $\mathtt{a}^n$.
- We honestly don't know if M is a decider for this language.
  - If the hailstone sequence always terminates, then M always halts and is a decider for L.
  - If the hailstone sequence doesn't always terminate, then M will loop on some inputs and isn't a decider for L.

```
bool pizkwat(string input) {
  return false;
}
```

```
bool squigglebah(string input) {
  while (true) {
    // do nothing
  }
}
```

```
bool moozle(string input) {
  int oot = 1;
  while (input.size() != oot) {
    oot += oot;
  }
  return true;
}
```

```
bool humblegwah(string input) {
  if (input.size() % 2 != 0) return false;

  for (int i = 0; i < input.size() / 2; i++) {
    if (input[2 * i] != input[2 * i + 1]) {
      return false;
    }
  }

  return true;
}
```

$$\forall w \in \Sigma^*.\ M \text{ halts on } w$$

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

Each piece of code is a recognizer for a language.
Which are deciders?

Deciders and Decidability

- While no one knows whether there are integers x, y, and z where

$$x^3 + y^3 + z^3 = 114,$$

  it is very easy to figure out whether there are integers x, y, and z where

$$x^2 + y^2 + z^2 = 114.$$

- Take a minute to discuss – why is this?

Deciders and Decidability

- Consider the language

  $L = \{ \mathtt{a}^n \mid \exists x \in \mathbb{Z}.\ \exists y \in \mathbb{Z}.\ \exists z \in \mathbb{Z}.\ x^2 + y^2 + z^2 = n \}$.

- Here's pseudocode for a decider to see whether such a triple exists:

```
for x from 0 to n:
  for y from 0 to n:
    for z from 0 to n:
      if x² + y² + z² = n: return true
return false
```

- After trying all possible options, this program will either find a triple that works or report that none exists.

Deciders and Decidability

- The class **R** consists of all decidable languages.
- Formally speaking:

  R = { L | L is a language and there's a decider for L }

- You can think of **R** as "all problems with yes/no answers that can be fully solved by computers."
  - Given a decidable language, run a decider for L and see what happens.
  - Think of this as "knowledge creation" – if you don't know whether a string is in L, running the decider will, given enough time, tell you.
- The class **R** contains all the regular languages, all the context-free languages, most of CS161, etc.
- This is a "strong" notion of solving a problem.

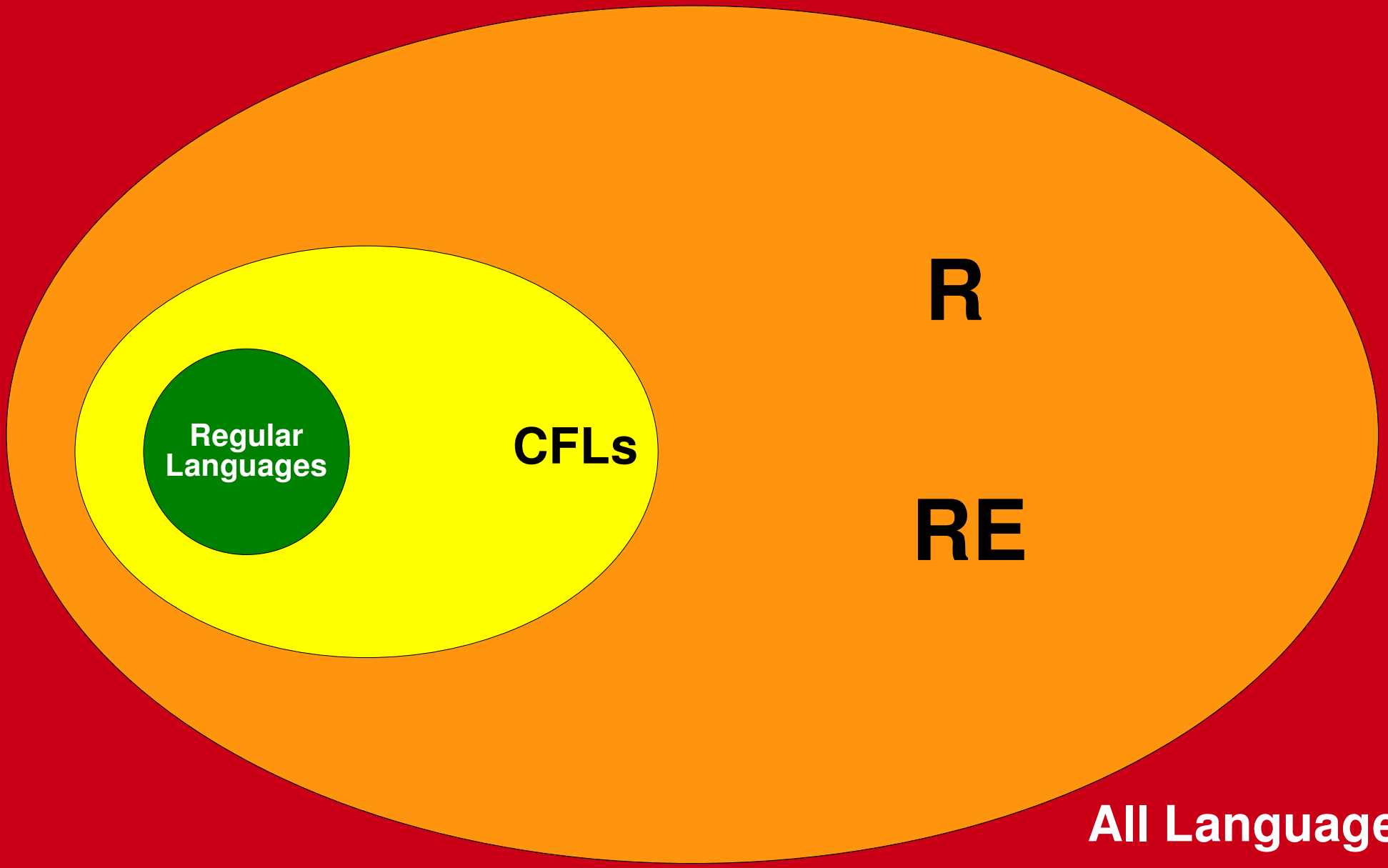
R and **RE** Languages

- Every decider for L is also a recognizer for L.
- This means that $\mathbf{R} \subseteq \mathbf{RE}$.
- Hugely important theoretical question:

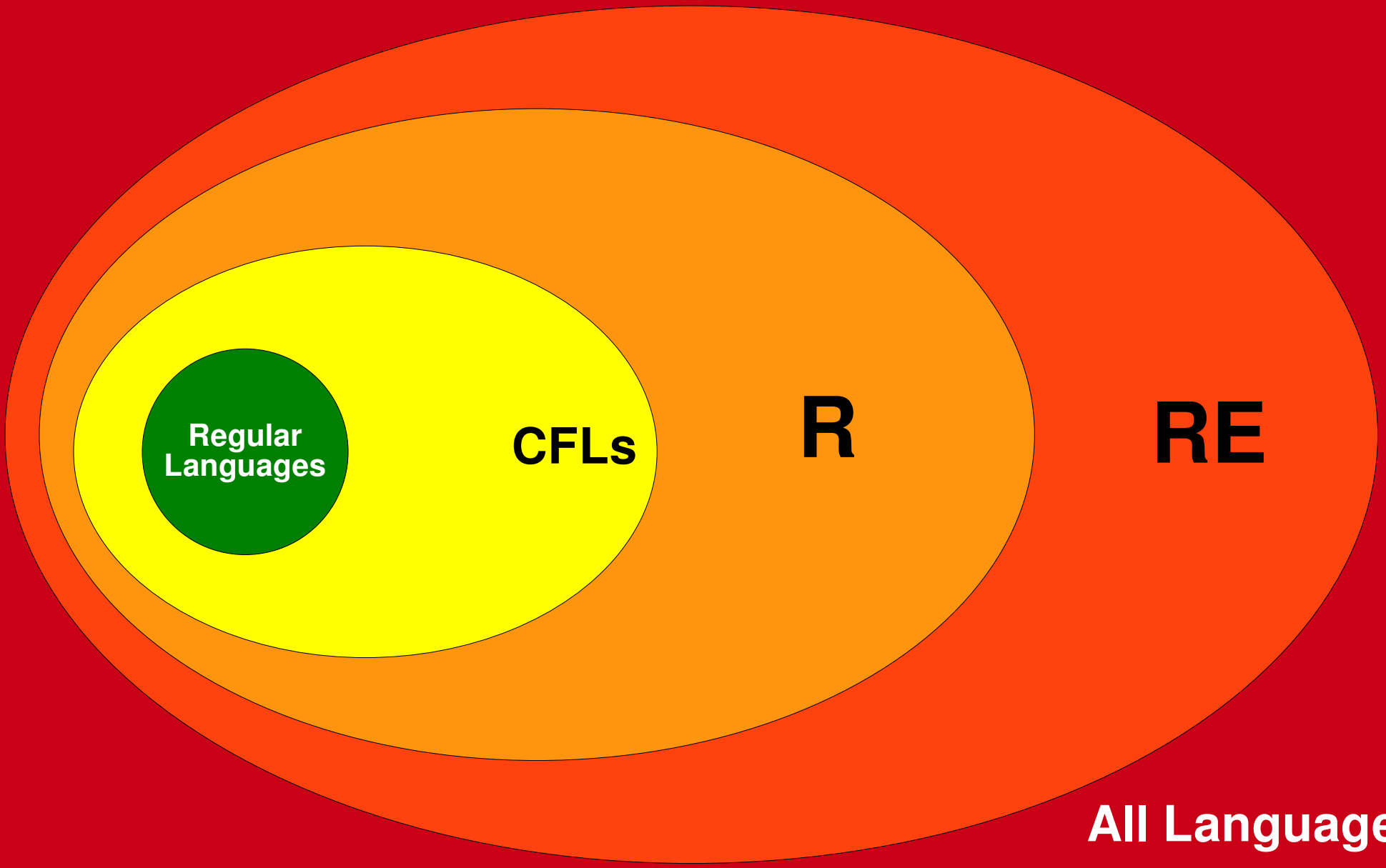
$$\mathbf{R} \stackrel{?}{=} \mathbf{RE}$$

- That is, if you can just confirm "yes" answers to a problem, can you necessarily *solve* that problem?

Which Picture is Correct?



Which Picture is Correct?



Unanswered Questions

- Why exactly is **RE** an interesting class of problems?
- What does the $\mathbf{R} \stackrel{?}{=} \mathbf{RE}$ question mean?
- Is $\mathbf{R} = \mathbf{RE}$?
- What lies beyond **R** and **RE**?
- We'll see the answers to each of these in due time.

Next Time

- ***Emergent Properties***
  - Larger phenomena made of smaller parts.
- ***Universal Machines***
  - A single, “most powerful” computer.
- ***Self-Reference***
  - Programs that ask questions about themselves.