

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 $\Sigma = \{\mathbf{a}, \mathbf{b}\}$.
- 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
```

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 Computer

- Programming languages provide a set of simple constructs.
 - Think things like variables, arrays, loops, functions, classes, etc.
- ***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.

Can TMs do: Loops?

- 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 \mathbf{ba} and replaces it with \mathbf{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.

Can TMs do: arithmetic?

- 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.

Can TMs do: variables?

- We've seen TMs that maintain variables.
 - You can think of our TM for $\{ \mathbf{a}^n \mathbf{b}^n \mid n \in \mathbb{N} \}$ as storing two variables – one that counts a number of \mathbf{a} 's, and one that counts a number of \mathbf{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.

Can TMs do: helper functions?

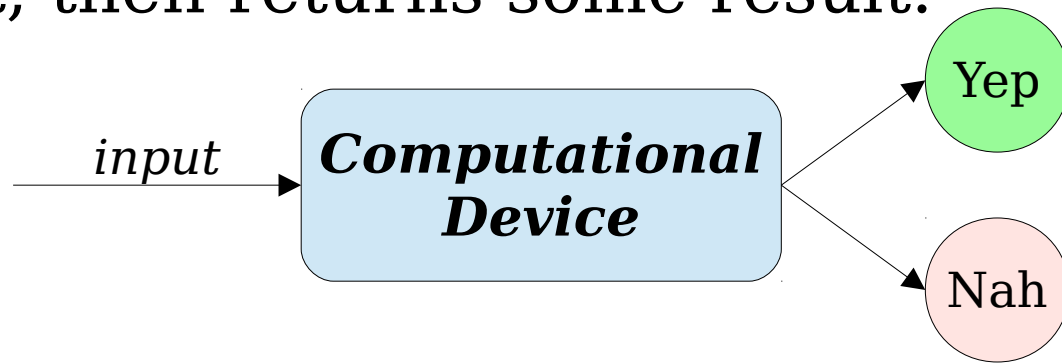
- We've seen TMs with helper functions.
 - We saw how to check for equal numbers of **a**'s and **b**'s by first sorting the string, then checking if the string has the form **aⁿbⁿ**.
 - We can check if a decimal number is a Fibonacci number by converting it to unary, then running our unary Fibonacci 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 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?

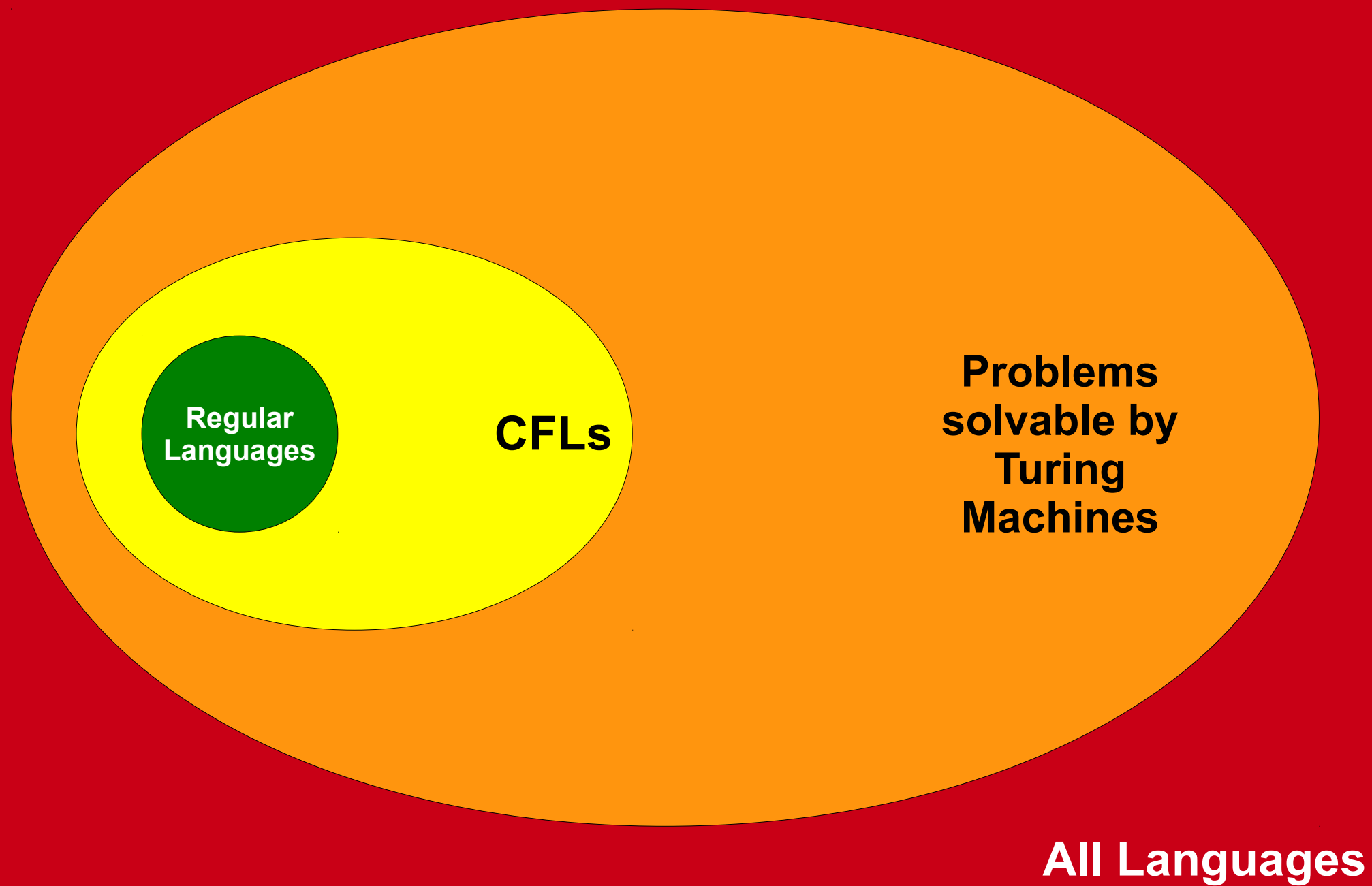
Effective Computation

- 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



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 more C++ or python-like 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!

Decidability and Recognizability

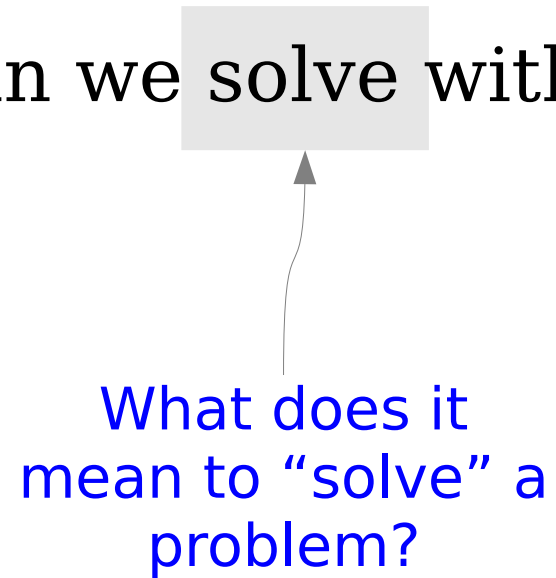
What problems can we solve with a computer?

What kind of
computer?



What problems can we solve with a computer?

What does it
mean to “solve” a
problem?

A diagram consisting of a main text block at the top and a callout block below it. The main text is 'What problems can we solve with a computer?'. The word 'solve' is highlighted with a light gray rectangular background. A thin, curved gray line with a small arrowhead at the top points from the callout text to the highlighted word 'solve'. The callout text is 'What does it mean to “solve” a problem?' and is colored blue.

A Simple Turing Machine

- Here's a TM.
- It receives inputs over the alphabet $\Sigma = \{\mathbf{a}, \mathbf{b}\}$.
- What strings does this TM accept?
- What happens when you give it these strings: \mathbf{a} , \mathbf{aaa} , \mathbf{bb} , \mathbf{baaa}

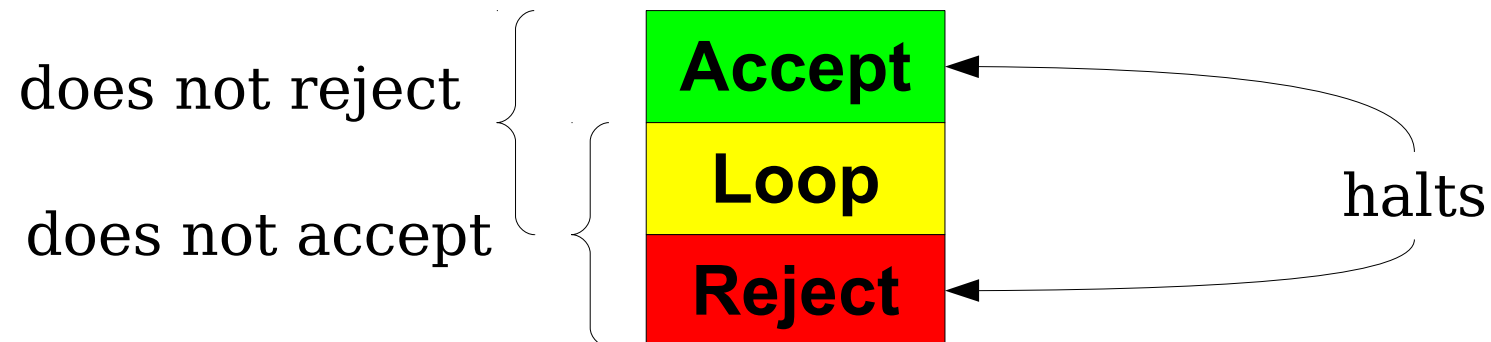
```
Start:  
    If 'a' Return True  
Loop:  
    Move Right  
    If Not 'b' Goto Loop
```

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

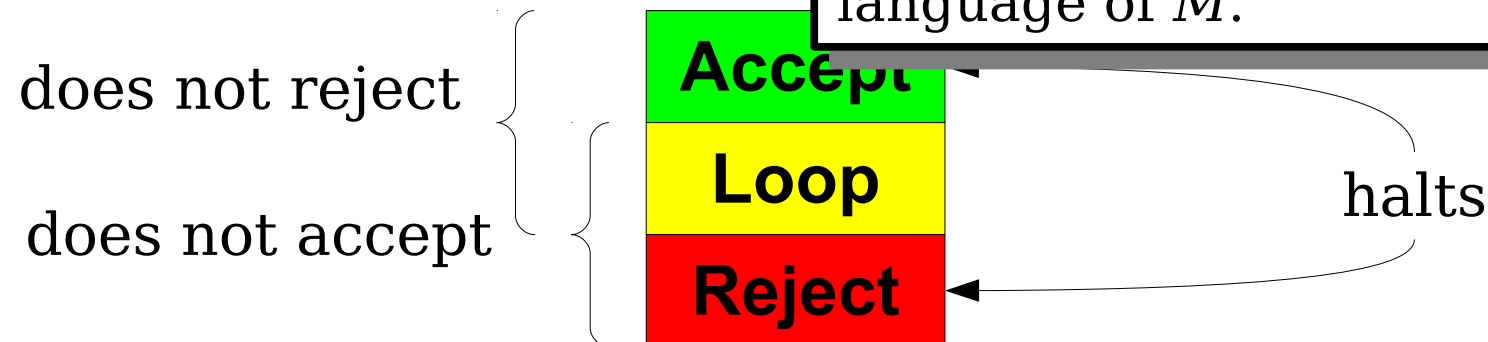
- Let M be a Turing machine.
- M **accepts** a string w if it returns true on w .
- M **rejects** a string w if it returns false on w .
- M **halts** on a string w if it returns on w (i.e., we don't care if it returns true or false, just that it returns at all).
- M **loops infinitely** (or just **loops**) on a string w if when run on w it neither returns true nor returns false (i.e., it doesn't halt).



Very Important Terminology

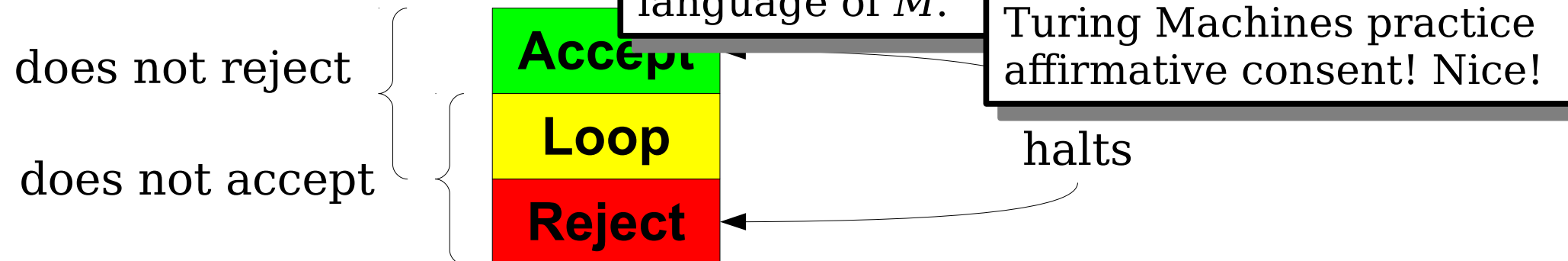
- Let M be a Turing machine.
- M **accepts** a string w if it returns true on w .
- M **rejects** a string w if it returns false on w .
- M **halts** on a string w if it returns on w (i.e., we don't care if it returns true or false, just that it returns at all).
- M **loops infinitely** (or just **loops**) on a string w if it neither returns true nor false on w .

To be in the language of M , a string must be accepted by M . No answer/looping is the same as rejection, in terms of meaning that the string is not in the language of M .



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 **halts** on a string w if it returns on w (i.e., we don't care if it returns true or false, just that it returns at all).
- M **loops infinitely** (or just **loops**) on a string w if it neither returns true nor false on 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 $w \in L$, then, eventually, M will accept w .
- For all strings not in L , M may reject or may infinite loop.
- Usefulness of this as a computer:
 - If you don't already 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.”
 - This is a very weak definition of “solving a problem,” but we are after all exploring the outer extremes of what computers can do

```
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;  
}
```

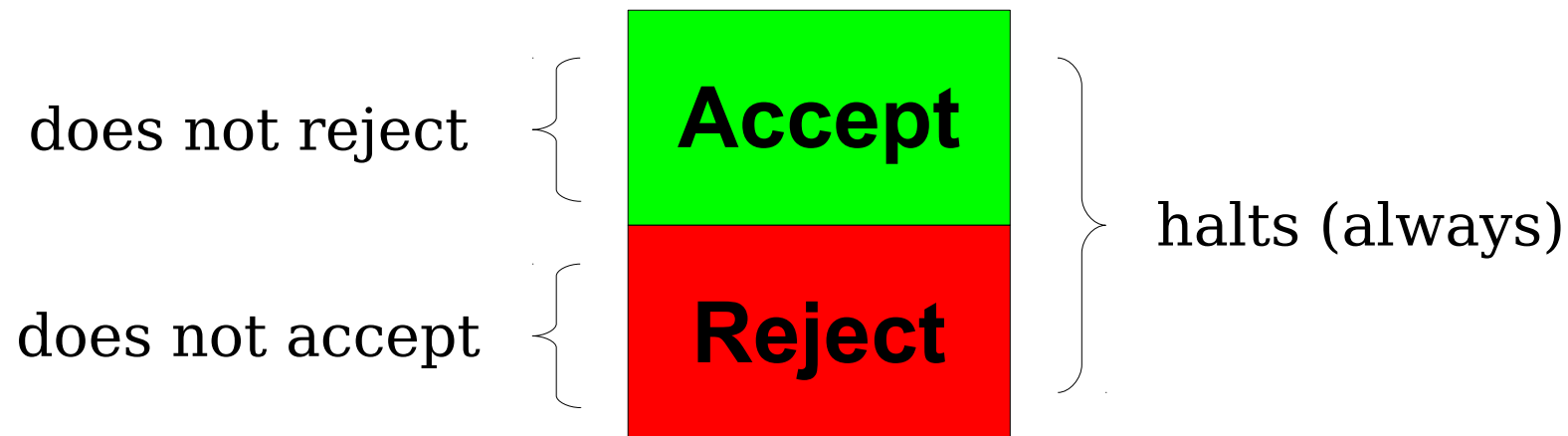
```
bool humblegwah(string input) {  
    if (input.size() % 2 != 0) return false;  
  
    for (int i = 0; i < input.size() / 2; i++) {  
        if (input[2 * i] != input[2 * i + 1]) {  
            return false;  
        }  
    }  
  
    return true;  
}
```

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

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

Deciders and Decidability

- Some, but not all, TMs have the following property: the TM halts on all inputs.
- Such a TM is called a “Decider.”
 - All deciders are recognizers.
 - Not all recognizers are deciders.



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.
- No matter what input string w you give a decider TM, you will always get a clear yes or no answer.

```
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;  
}
```

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

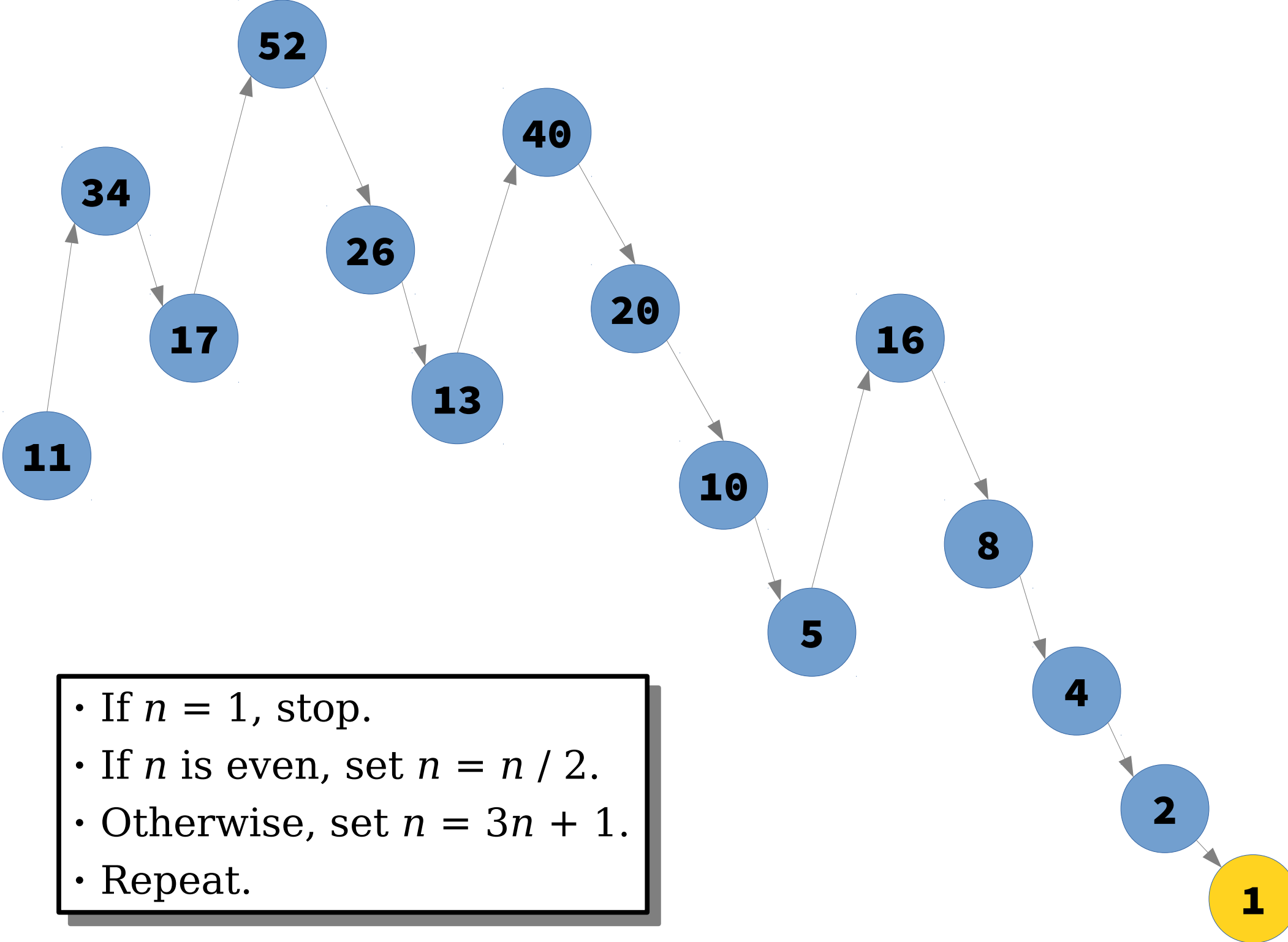
$\forall w \in \Sigma^*. (w \in L \leftrightarrow M \text{ accepts } w)$

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

A Tricky TM

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?



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 $\Sigma = \{\mathbf{a}\}$ and consider the language
$$L = \{ \mathbf{a}^n \mid n > 0 \text{ and the hailstone sequence terminates for } n \}.$$
- 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.

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 the 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.”

Hailstone Decider?

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

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

- If the hailstone sequence terminates for n , then M accepts \mathbf{a}^n . If it doesn't, then M does not accept \mathbf{a}^n .
- *Is it also a decider?*

Hailstone Decider?

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

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

- If the hailstone sequence terminates for n , then M accepts \mathbf{a}^n . If it doesn't, then M does not accept \mathbf{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 , and L turns out to be just all strings $\mathbf{a}^n \mid n > 0$ (a Regular language!).
 - If the hailstone sequence doesn't always terminate, then M will loop on some inputs and isn't a decider for L , and L is some strict subset of $\mathbf{a}^n \mid n > 0$.

Two new language classes

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.”

Deciders and Decidability

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

$$\mathbf{R} = \{ L \mid L \text{ is a language and there exists a decider for } L \}$$

- You can think of **R** as “all problems with yes/no answers that can be fully solved by computers.”
 -
- 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.

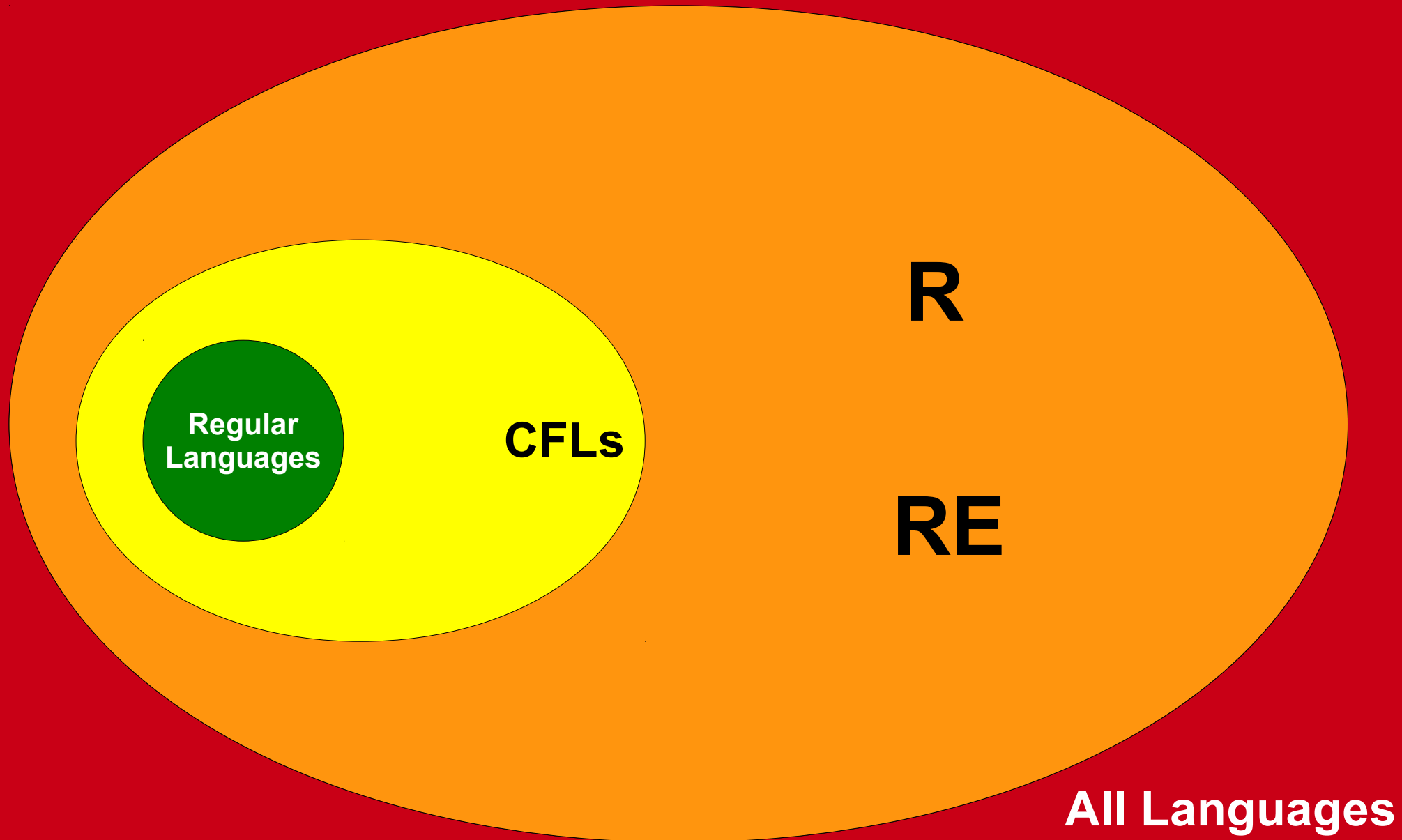
R and **RE** Languages

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

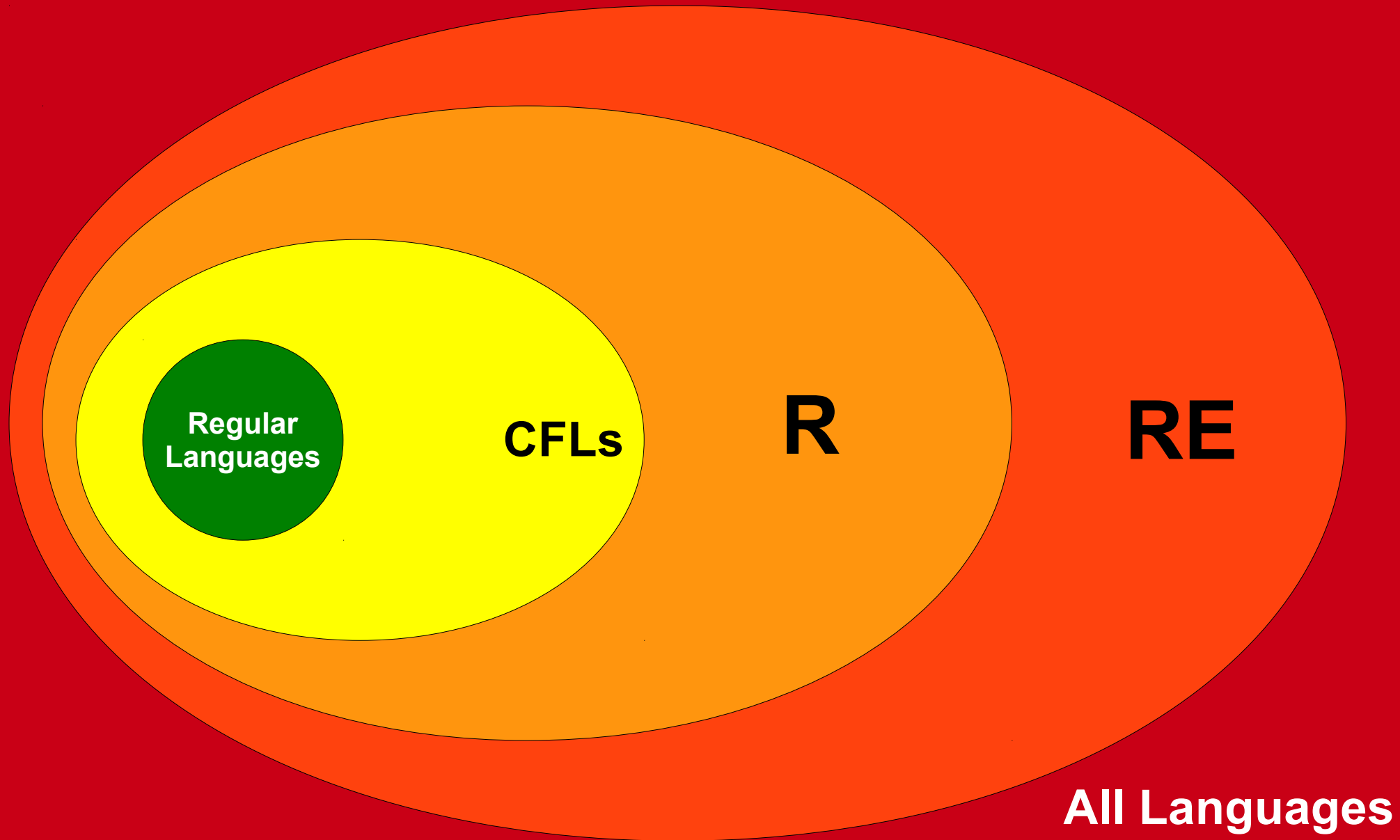
$$\mathbf{R} \stackrel{?}{=} \mathbf{RE}$$

- 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.