CS103
Winter 2026


Lecture 24:

# Unsolvable Problems

**Part 2 of 2**

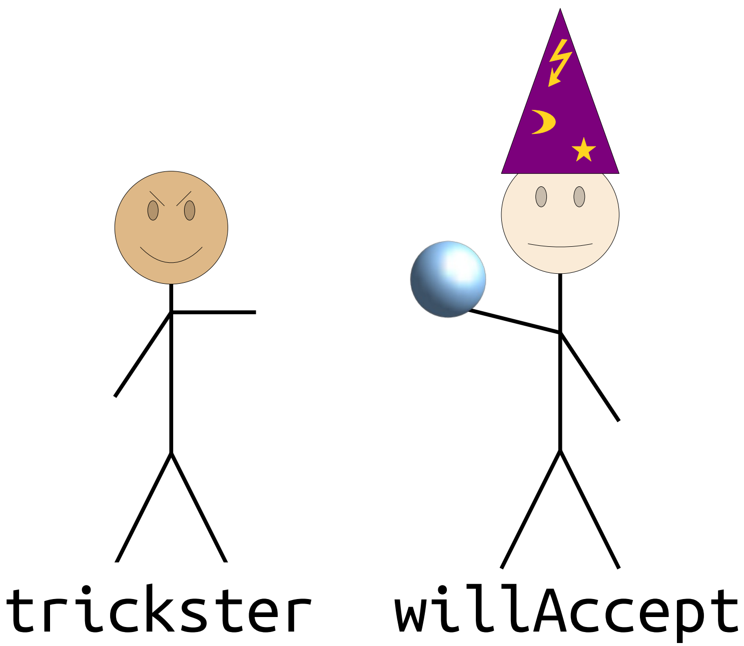
# Outline for Today

- ***More on Undecidability***
  - Even more problems we can’t solve.
- ***A Different Perspective on RE***
  - What exactly does “recognizability” mean?
- ***Verifiers***
  - A new approach to problem-solving.
- ***Beyond RE***
  - A beautiful example of an impossible problem.

# Recap from Last Time

```
bool willAccept(string function, string input) {
   // Returns true if function(input) returns true.
   // Returns false otherwise.
}

bool trickster(string input) {
   string me = /* source code of trickster */;
   return !willAccept(me, input);
}
```




Which of the following must be true?

(1) `trickster` is a decider for $A_{TM}$.

(2) `willAccept` is a decider for $A_{TM}$.

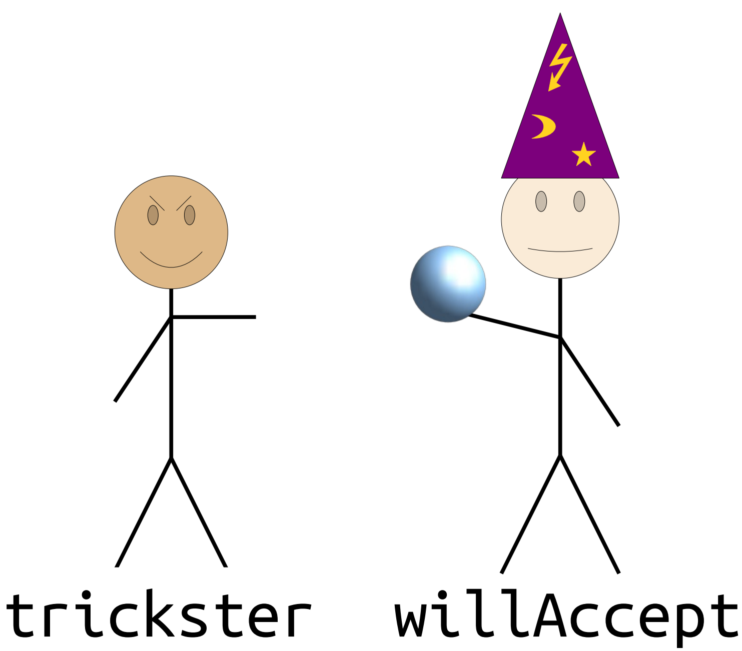
(3) `willAccept(me, input)` simulates `trickster` on `input` and does whatever `trickster` does to `input`.

(4) `trickster` loops on at least one input.

Answer at ***https://cs103.stanford.edu/pollev***

```
bool willAccept(string function, string input) {
    // Returns true if function(input) returns true.
    // Returns false otherwise.
}

bool trickster(string input) {
    string me = /* source code of trickster */;
    return !willAccept(me, input);
}
```




*A decider for $A_{TM}$ has to have this behavior.*

`trickster(input)` returns true

$\leftrightarrow$

`willAccept(me, input)` returns true

$\leftrightarrow$

`trickster(input)` does not return true

*Because of how we wrote* `trickster`*.*

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

***Proof:*** By contradiction; assume that $A_{TM} \in \mathbf{R}$. Then there is a decider $D$ for $A_{TM}$. We can represent $D$ as a function

```
bool willAccept(string function, string w);
```

that takes in the source code of a function `function` and a string `w`, then returns true if `function(w)` returns true and returns false otherwise. Given this, consider this function `trickster`:

```
bool trickster(string input) {
    string me = /* source code of trickster */;
    return !willAccept(me, input);
}
```

Since `willAccept` decides $A_{TM}$ and `me` holds the source of `trickster`, we know that

`willAccept(me, input)` returns true if and only if `trickster(input)` returns true.

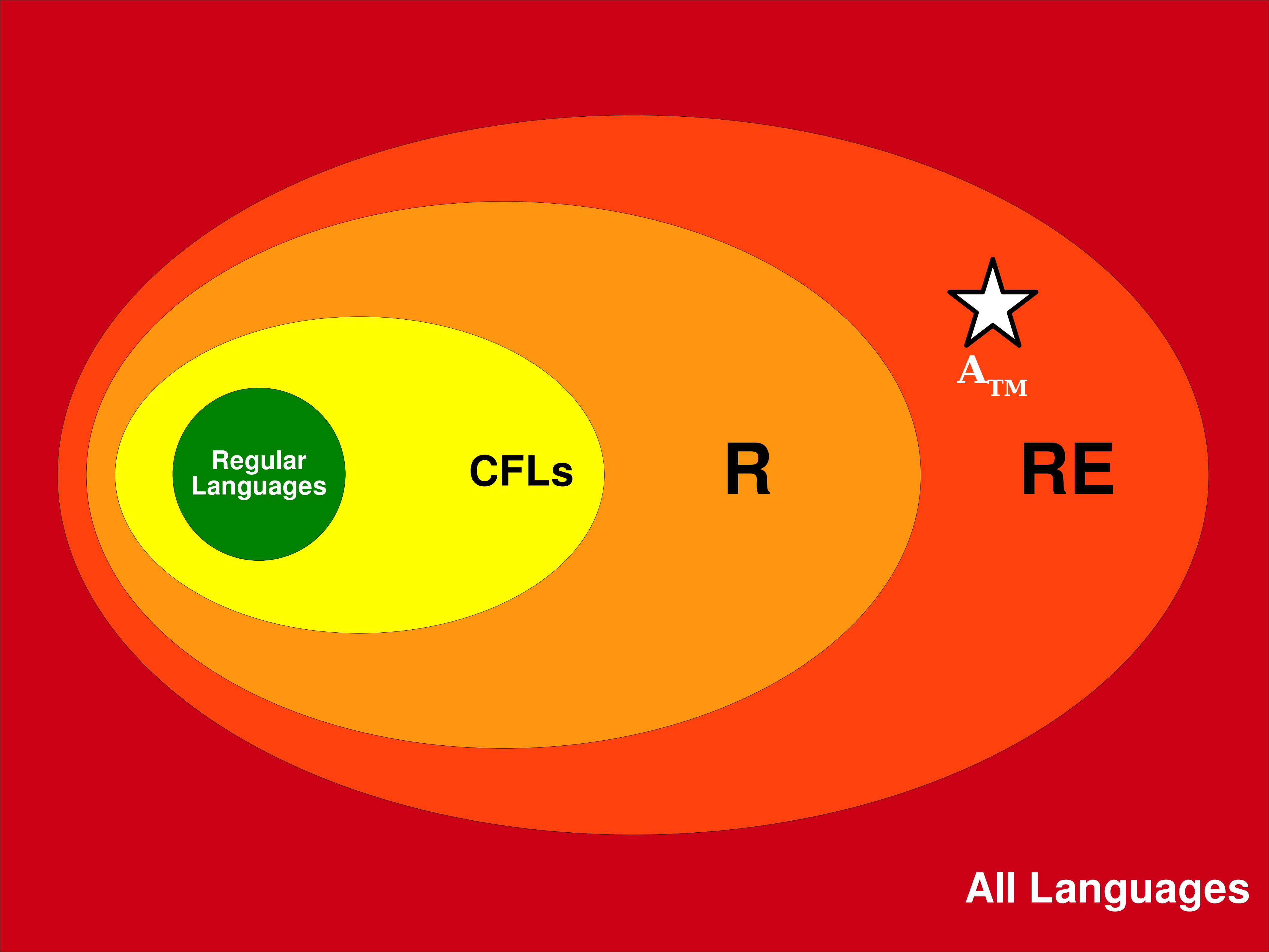
Given how `trickster` is written, we see that

`willAccept(me, input)` returns true if and only if `trickster(input)` doesn't return true.

This means that

`trickster(input)` returns true if and only if `trickster(input)` doesn't return true.

This is impossible. We've reached a contradiction, so our assumption was wrong and $A_{TM}$ is undecidable. ■



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

New Stuff!

# 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*?**

- Our goal isn't to build a TM *M* that halts on a string *w*. It's to check whether an arbitrary TM *M* halts on an arbitrary string *w*.
- As a formal language, this problem would be expressed as

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

- ***Theorem:*** *HALT* is recognizable, but undecidable.
  - There's a recognizer for *HALT*.
  - There is no decider for *HALT*.

# 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*?**

Our goal isn't to build a TM *M* that halts on a string *w*. It's to check whether an arbitrary TM *M* halts on an arbitrary string *w*.

As a formal language, this problem would be expressed as

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

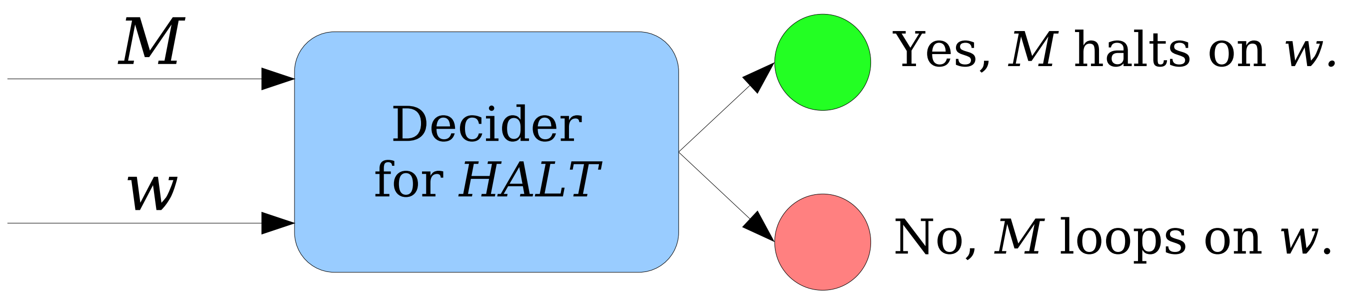
***Theorem:*** *HALT* is recognizable, but undecidable.

- There's a recognizer for *HALT*.
- There is no decider for *HALT*.

***Theorem:*** The halting problem is undecidable.

# A Decider for *HALT*

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




- We could represent this decider in software as a method

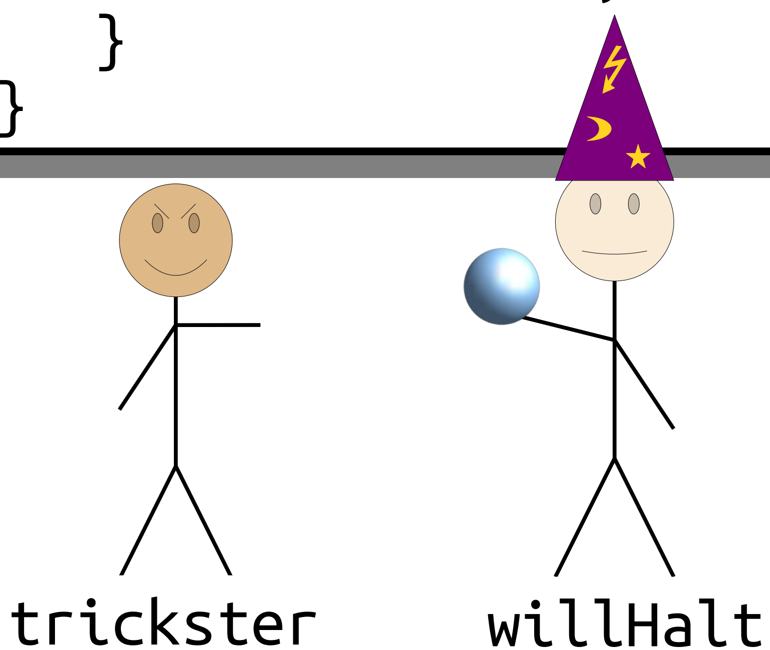
```
bool willHalt(string function, string input);
```

  that takes as input a function `function` and a string `input`, then
  - returns true if `function(input)` returns anything (halts), and
  - returns false if `function(input)` never returns anything (loops).

```
bool willHalt(string function, string input) {
   // Returns true if function(input) halts.
   // Returns false otherwise.
}

bool trickster(string input) {
   string me = /* source code of trickster */;

   if (willHalt(me, input)) {
      while (true) {
         // Do nothing
      }
   } else {
      return true;
   }
}
```




*A decider for HALT must do this.*

`trickster(input)` halts

↔

`willHalt(me, input)` returns true

↔

`trickster(input)` loops

*We wrote `trickster` to have this behavior.*

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

***Proof:*** By contradiction; assume that *HALT* $\in$ **R**. Then there is a decider *D* for *HALT*. We can represent *D* as a function

```
bool willHalt(string function, string w);
```

that takes in the source code of a function `function` and a string `w`, then returns true if `function(w)` halts and returns false otherwise. Given this, consider this function `trickster`:

```
bool trickster(string input) {
    string me = /* source code of trickster */;
    if (willHalt(me, input)) {
        while (true) { }
    } else {
        return true;
    }
}
```

Since `willHalt` decides *HALT* and `me` holds the source of `trickster`, we know that

`willHalt(me, input)` returns true   if and only if   `trickster(input)` halts.

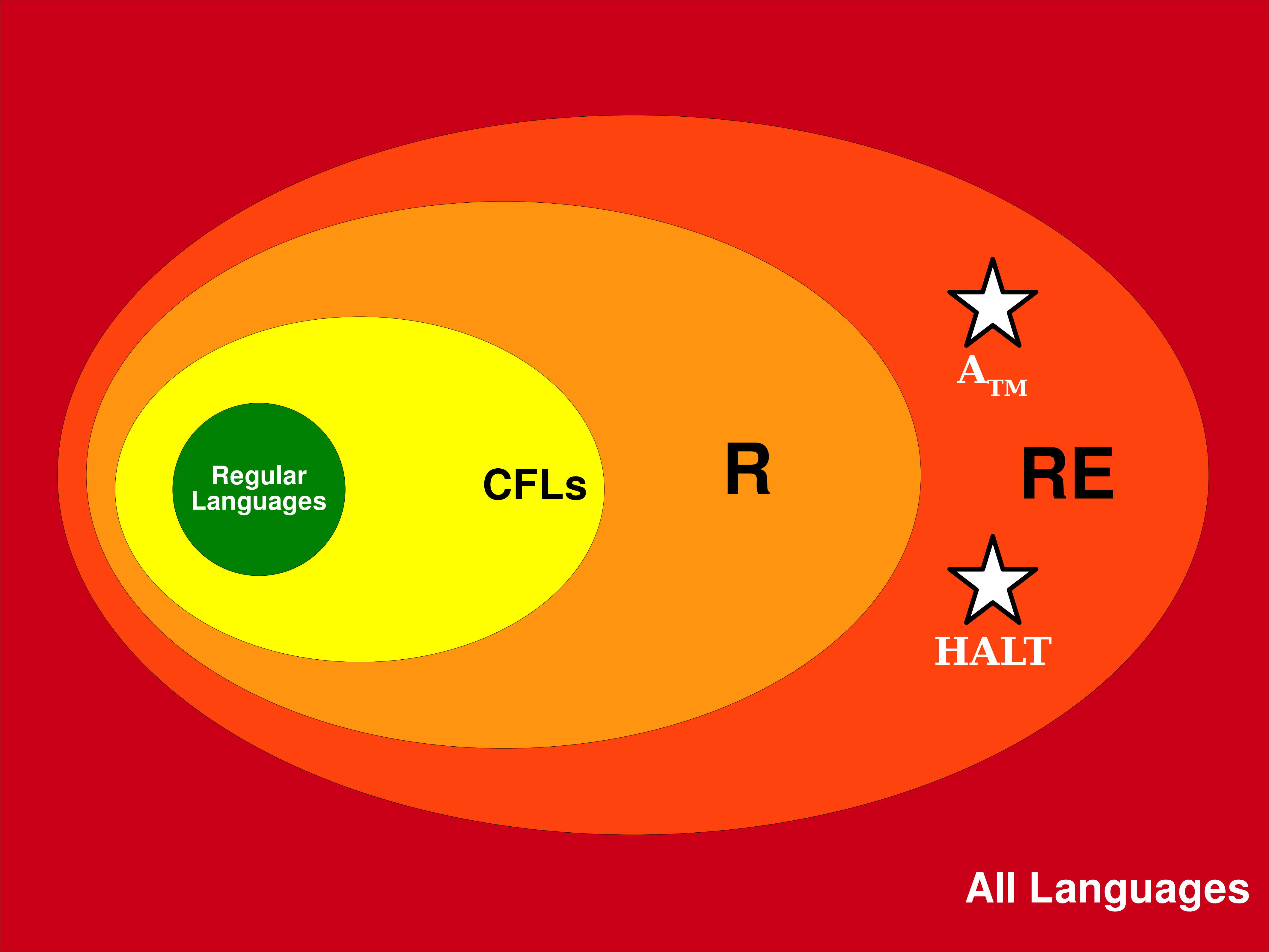
Given how `trickster` is written, we see that

`willHalt(me, input)` returns true   if and only if   `trickster(input)` loops.

This means that

`trickster(input)` halts   if and only if   `trickster(input)` loops.

This is impossible. We've reached a contradiction, so our assumption was wrong and *HALT* is undecidable. ■


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

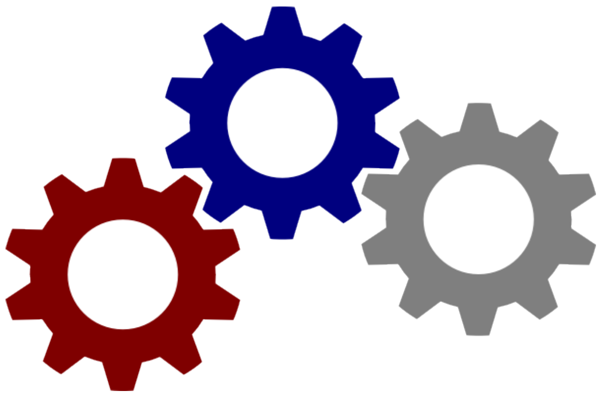
# So What?

- 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.

**Analogy Time!**

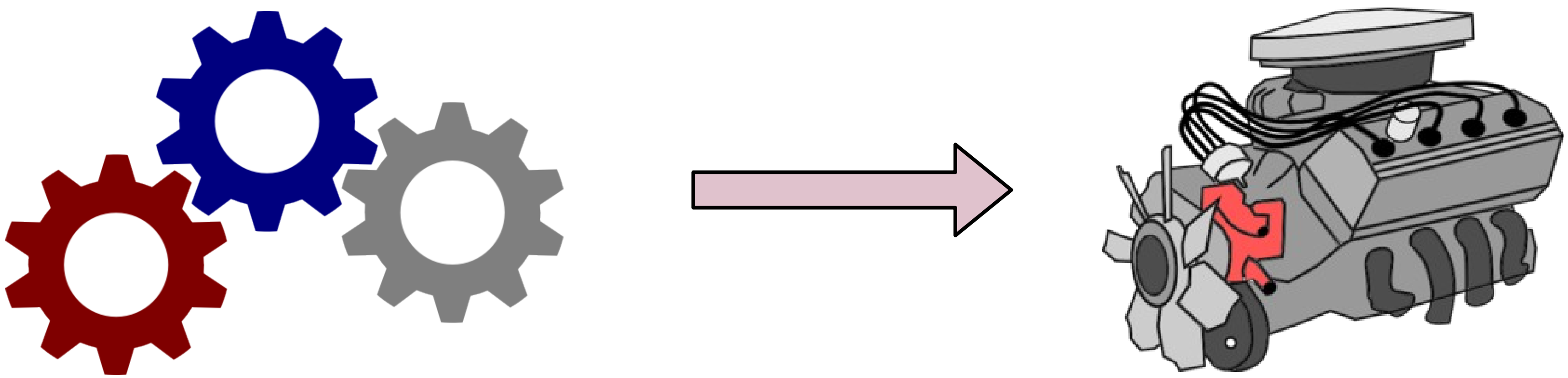
***Engineering Problem:*** Design a diesel engine that doesn't emit lots of $NO_x$ pollutants.

***Engineering Problem:*** Design a diesel engine that doesn't emit lots of $NO_x$ pollutants.



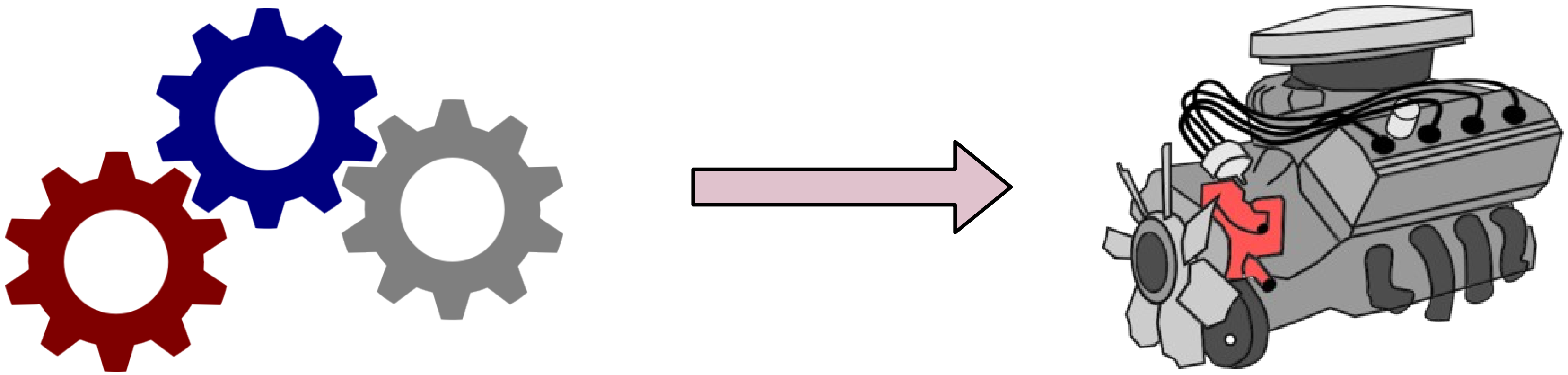
***Engineering Prowess!***

***Engineering Problem:*** Design a diesel engine that doesn't emit lots of $NO_x$ pollutants.



***Engineering Prowess!*** ***Awesome Engine!***

***Engineering Problem:*** Design a diesel engine that doesn't emit lots of $NO_x$ pollutants.

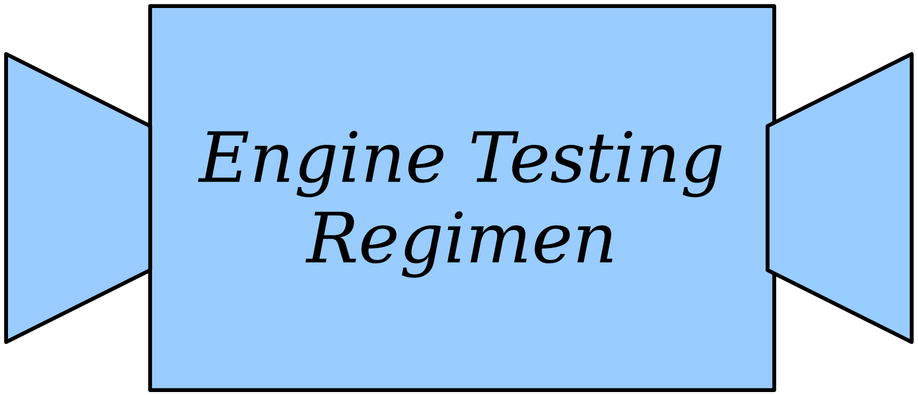


***Engineering Prowess!***

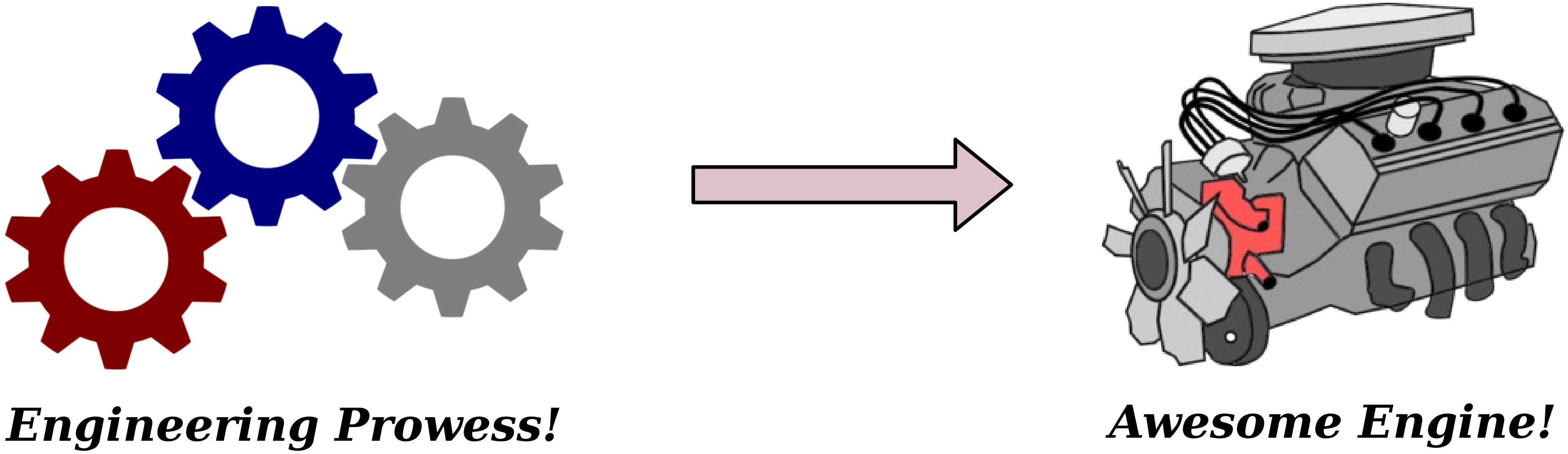
***Awesome Engine!***

---

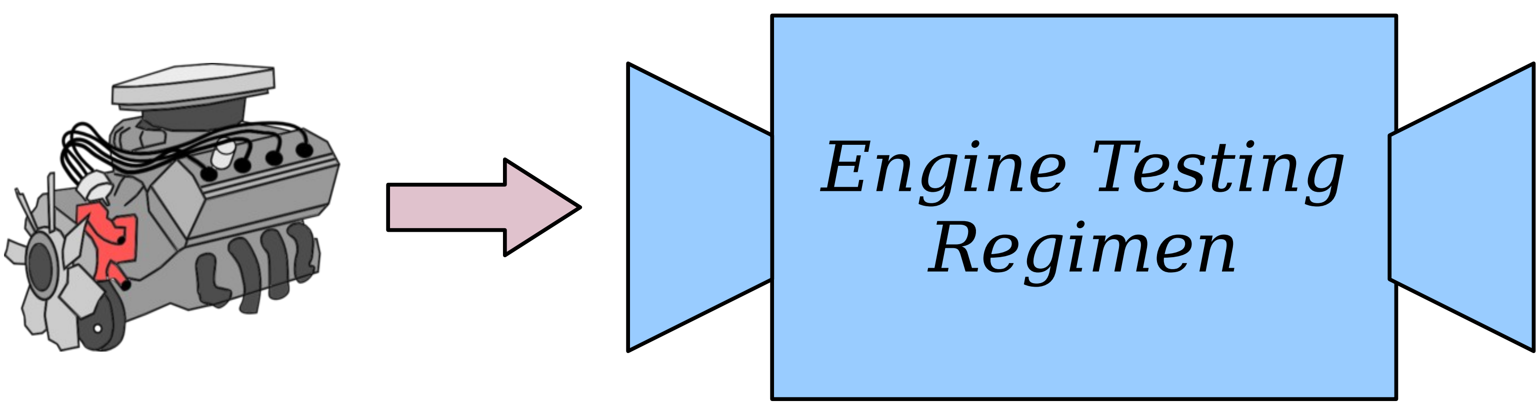
***Regulatory Problem:*** Design a testing procedure that, given a diesel engine, determines whether it emits lots of $NO_x$ pollutants.



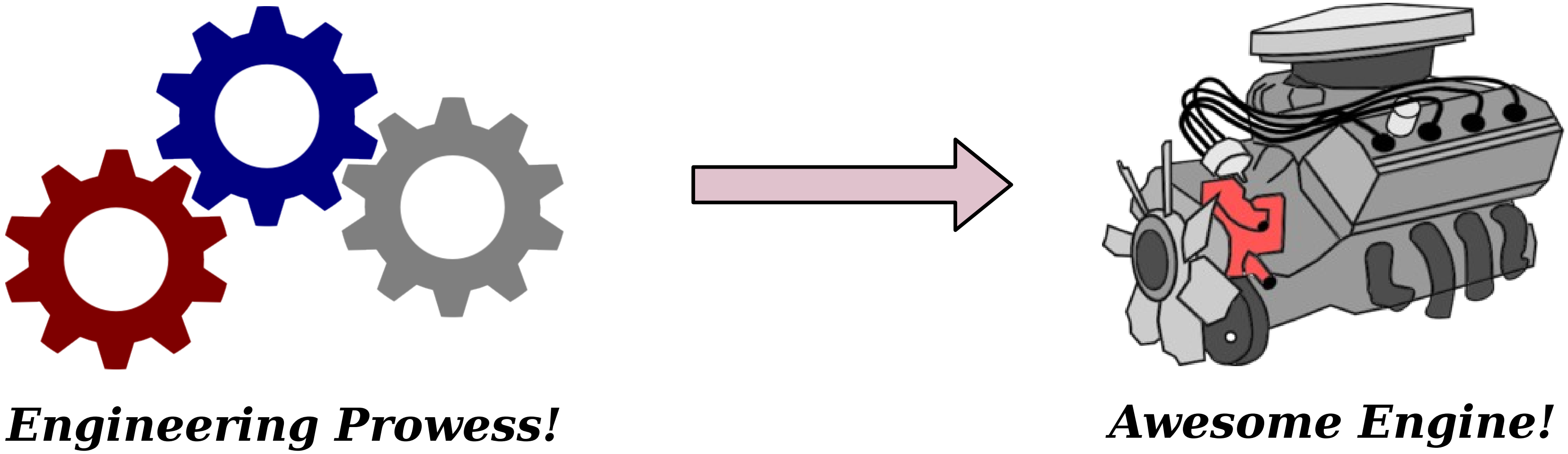
***Engineering Problem:*** Design a diesel engine that doesn't emit lots of $NO_x$ pollutants.



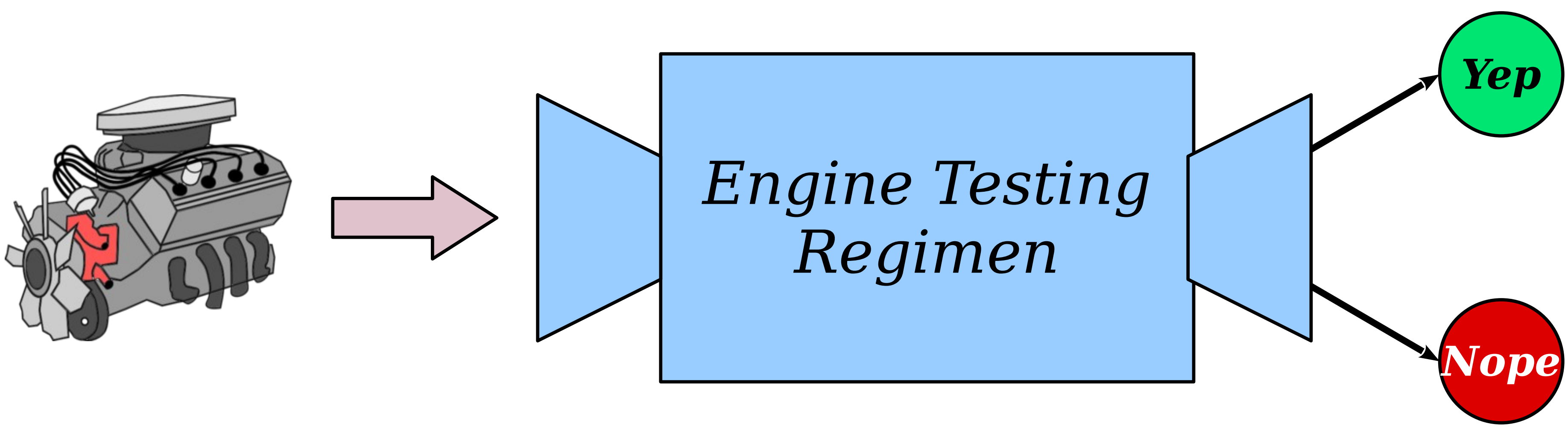

***Regulatory Problem:*** Design a testing procedure that, given a diesel engine, determines whether it emits lots of $NO_x$ pollutants.



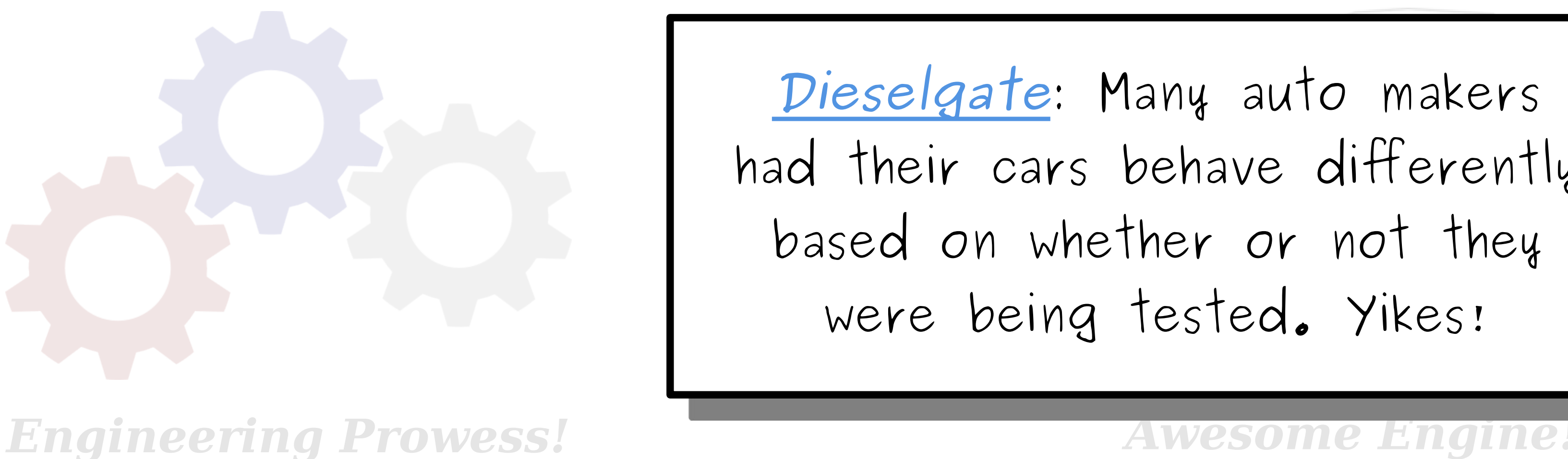
***Engineering Problem:*** Design a diesel engine that doesn't emit lots of $NO_x$ pollutants.




***Regulatory Problem:*** Design a testing procedure that, given a diesel engine, determines whether it emits lots of $NO_x$ pollutants.



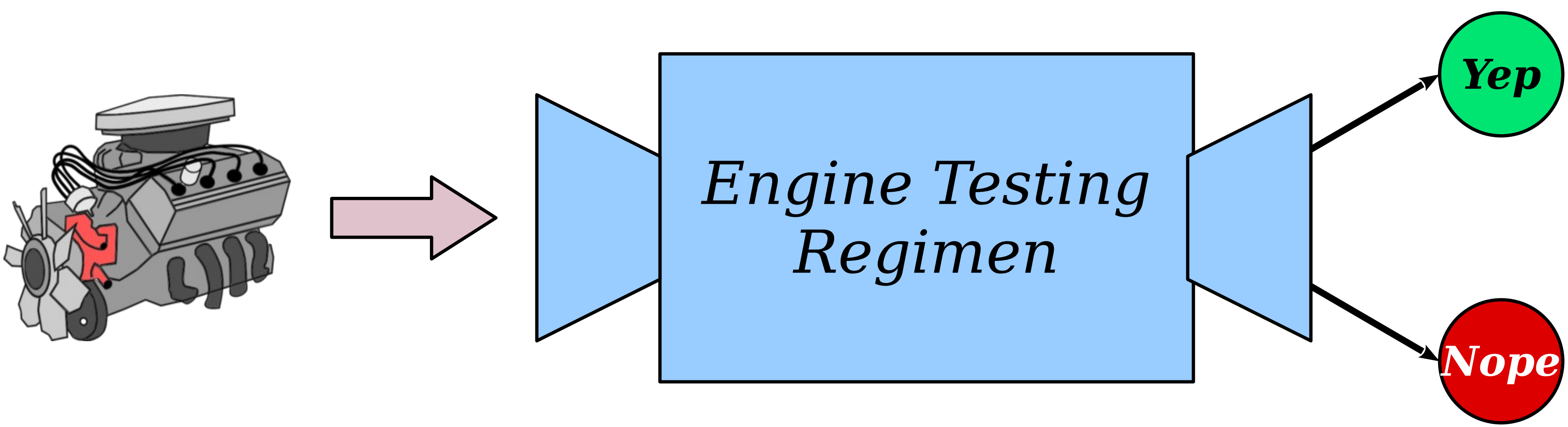
**Engineering Problem:** Design a diesel engine that doesn't emit lots of $NO_x$ pollutants.




Dieselgate: Many auto makers had their cars behave differently based on whether or not they were being tested. Yikes!

---

**Regulatory Problem:** Design a testing procedure that, given a diesel engine, determines whether it emits lots of $NO_x$ pollutants.



***Fact:*** Almost all "regulatory problems" about computer programs are undecidable.

That is, almost all problems of the form "does program $X$ have behavior $Y$?" are undecidable.

This can be formalized as ***Rice's Theorem***; take CS154 for details!

# A (Topical) Example

# Secure Voting

- Suppose that you want to make a voting machine for use in an election between two parties (the ***Zomp Party*** and the ***Puce Party***).
- Let Σ = {`z`, `p`}. A string *w* ∈ Σ* corresponds to a series of votes for the candidates.
- Example: `zzpppzp` means "two people voted for `z`, then three people voted for `p`, then one more person voted for `z`, then one more person voted for `p`."
- A ***secure voting machine*** is a TM that takes as input a string of `z`'s and `p`'s, then reports whether person `z` won the election.
  - "Secure" in the sense of "actually checks the vote totals" as opposed to rigging the election, discounting votes, etc.

A secure voting machine is a TM $M$ where
$M$ accepts $w \in \{$**z**, **p**$\}^*$ if and only if $w$ has more **z**'s than **p**'s.

```
bool bee(string input) {
   int numZs = countZsIn(input);
   int numPs = countPsIn(input);

   return numZs > numPs;
}
```

```
bool topaz(string input) {
   return input != "" &&
          input[0] == 'z';
}
```

Which of these are secure voting machines? Answer at ***https://cs103.stanford.edu/pollev***

```
bool anna(string input) {
   int numZs = countZsIn(input);
   int numPs = countPsIn(input);

   if (numZs = numPs) {
      return false;
   } else if (numZs < numPs) {
      return false;
   } else {
      return true;
   }
}
```

```
bool green(string input) {
   int n = input.length();
   while (n > 1) {
      if (n % 2 == 0) n /= 2;
      else n = 3*n + 1;
   }

   int numZs = countZsIn(input);
   int numPs = countPsIn(input);

   return numZs > numPs;
}
```

A secure voting machine is a TM $M$ where
$M$ accepts $w \in \{$**z**, **p**$\}^*$ if and only if $w$ has more **z**'s than **p**'s.

```
bool bee(string input) {
   int numZs = countZsIn(input);
   int numPs = countPsIn(input);

   return numZs > numPs;
}
```

*A (simple) secure voting machine.*

```
bool topaz(string input) {
   return input != "" &&
          input[0] == 'z';
}
```

*A (simple) insecure voting machine.*

```
bool anna(string input) {
   int numZs = countZsIn(input);
   int numPs = countPsIn(input);

   if (numZs = numPs) {
      return false;
   } else if (numZs < numPs) {
      return false;
   } else {
      return true;
   }
}
```

*An (evil) insecure voting machine.*

```
bool green(string input) {
   int n = input.length();
   while (n > 1) {
      if (n % 2 == 0) n /= 2;
      else n = 3*n + 1;
   }

   int numZs = countZsIn(input);
   int numPs = countPsIn(input);

   return numZs > numPs;
}
```

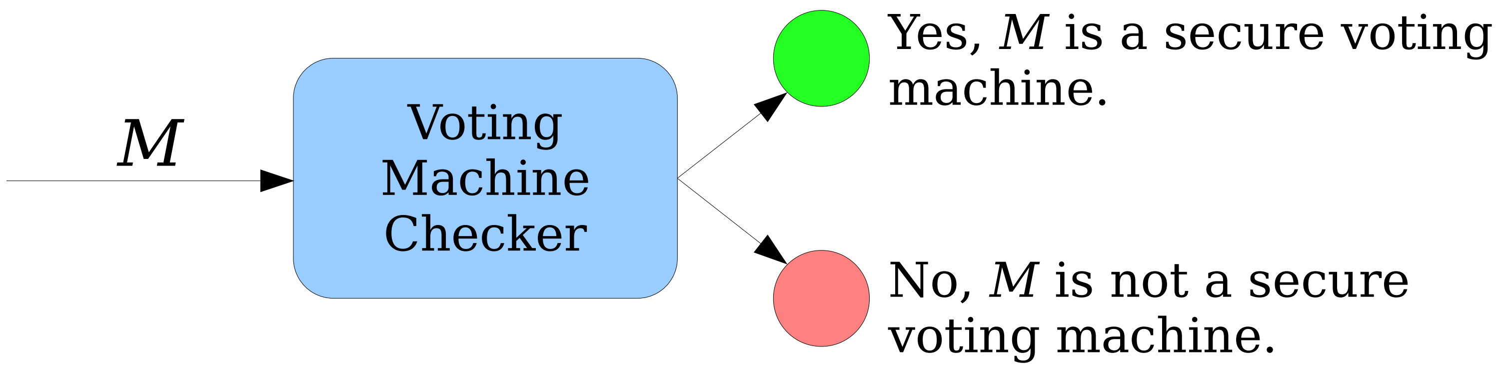
*No one knows!*

# Secure Voting

- Even human review isn't perfect for vetting voting software.
- ***Question:*** Could we design an algorithm to check voting software for us?
  - ***Input:*** A Turing machine $M$.
  - ***Output:*** YES if $M$ is a secure voting machine, NO if $M$ isn't.
- This is a "regulatory" problem, not an "engineering" problem.

# A Decider for Secure Voting

- Schematically, a "voting machine checker" would look like this:




- We'd represent this decider in software as a function

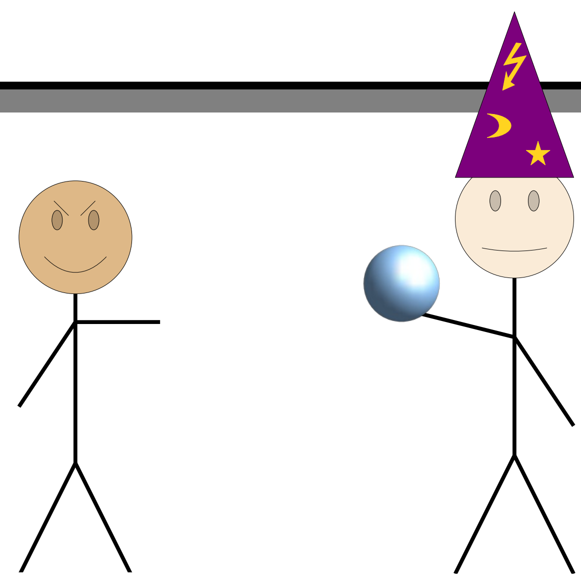
```
bool isSecureVotingMachine(string function);
```

that takes as input a function, then returns whether that function is a secure voting machine.

```
bool isSecureVotingMachine(string function) {
   // Returns whether function accepts only
   // strings with more z's than p's.
}

bool trickster(string input) {
   string me = /* source code of trickster */;

   if (isSecureVotingMachine(me)) {
       return countZsIn(input) <= countPsIn(input);
   } else {
       return countZsIn(input) > countPsIn(input);
   }
}
```



trickster isSecureVotingMachine

`trickster` is a secure voting machine

↔

`isSecureVotingMachine(me)` returns true

↔

`trickster` isn't a secure voting machine.

***Theorem:*** The secure voting problem is undecidable.

***Proof:*** By contradiction; there is a decider *D* for the secure voting problem. We can represent *D* as a function

```
bool isSecureVotingMachine(string function);
```

that takes in the source code of a function `function`, then returns whether function is a secure voting machine (that is, whether it accepts precisely the strings with more **`z`**'s than **`p`**'s). Given this, consider this function `trickster`:

```
bool trickster(string input) {
    string me = /* source code of trickster */;
    if (isSecureVotingMachine(me)) {
        return /* if input has at most as many z's as p's */;
    } else {
        return /* if input has more z's than p's */;
    }
}
```

Since `isSecureVotingMachine` decides the secure voting problem and `me` holds the source of `trickster`, we know that

`isSecureVotingMachine(me)` returns true if and only if `trickster` is a secure voting machine.

Given how `trickster` is written, we see that

`isSecureVotingMachine(me)` returns true if and only if `trickster` isn't a secure voting machine

This means that

`trickster` is a secure voting machine if and only if `trickster` isn't a secure voting machine.

This is impossible. We've reached a contradiction, so our assumption was and the secure voting problem is undecidable. ■

# Interpreting this Result

- The previous argument tells us that *there is no automated procedure* that can check if arbitrary voting software is correct.
- 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?)
- Worth a read: ***https://xkcd.com/2030/***

# Time-Out for Announcements!

# Problem Set 9

- Problem Set 8 was due today at 1:00 PM.
  - You can use a late day to extend the deadline to Saturday at 1:00 PM.
- Problem Set 9 is now posted. It's due next Friday (March 13).
  - This is a normally-sized problem set.
  - Late days can't be used here; this is university policy.

# Cumulative Practice Problems

- We've just released a ***massive*** bank of practice problems on the course website you can use to review topics from throughout the quarter and two practice exams.
- Feel free to ask us questions in office hours or on EdStem if you have them. That's what we're here for!
- Some exam prep thoughts:
  - It's great to study this material and get practice. Just make sure to do it in a way that's maximally conducive to learning.
  - You're not competing against anyone else in this course. As you review for the final, form study groups. Share ideas and insights with one another.
  - We assign grades to certify skills, not based on relative performance. A's are not a scarce resource; we'd love to give as many as we can.
- Best of luck on the home stretch!

# Important Tasks

- Check out the appendices in the finalized slide deck that I'll post later today!
- Work carefully through the guides to be posted after class today:
  - Guide to Self-Reference
  - Guide to the Lava Diagram
- These action items are critical to your final exam preparations.

Back to CS103!

# Beyond **R** and **RE**

What exactly is the class **RE**?

# **RE**, Formally

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

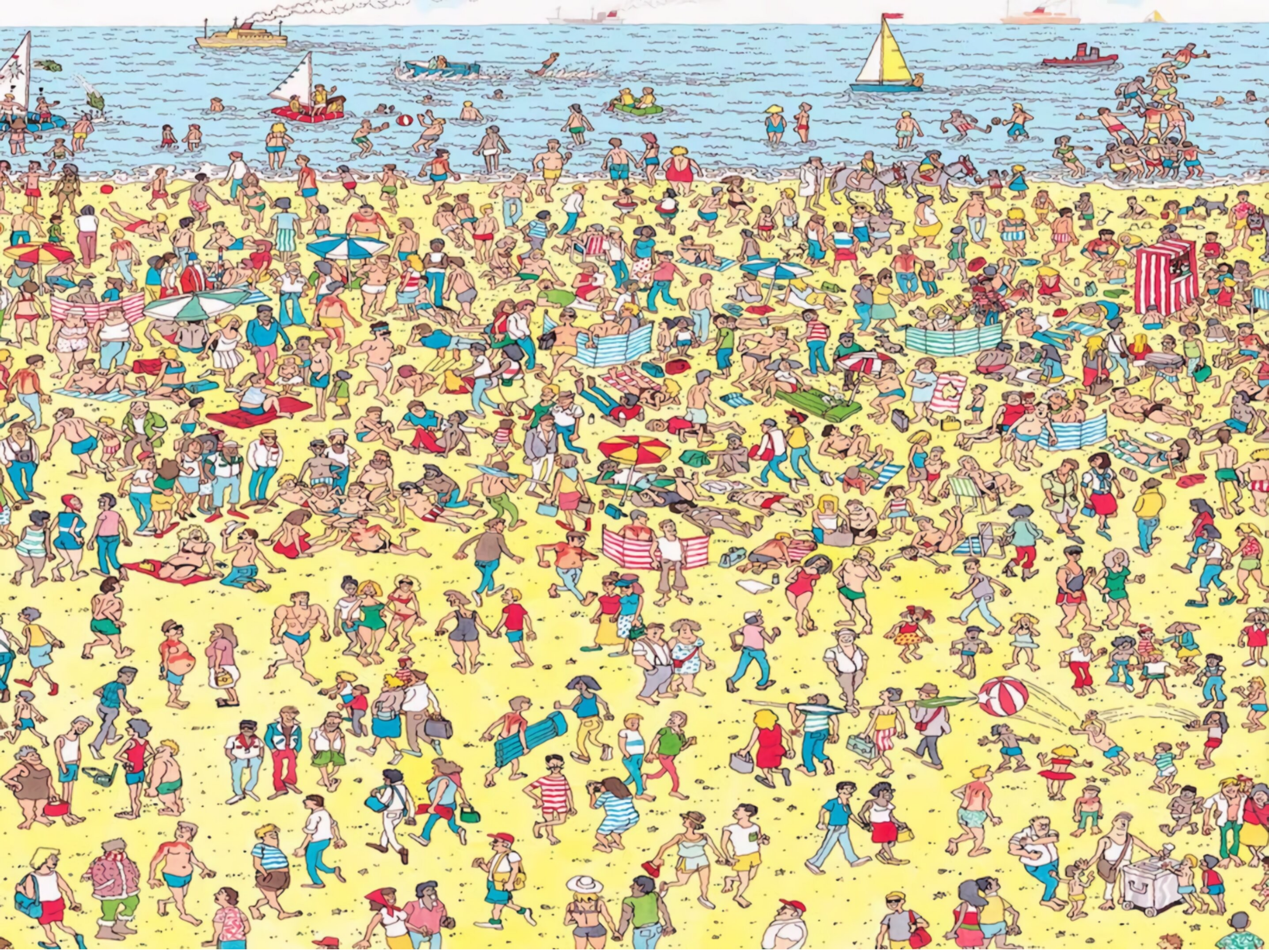
  **RE** = { *L* | there is a TM *M* that recognizes *L* }

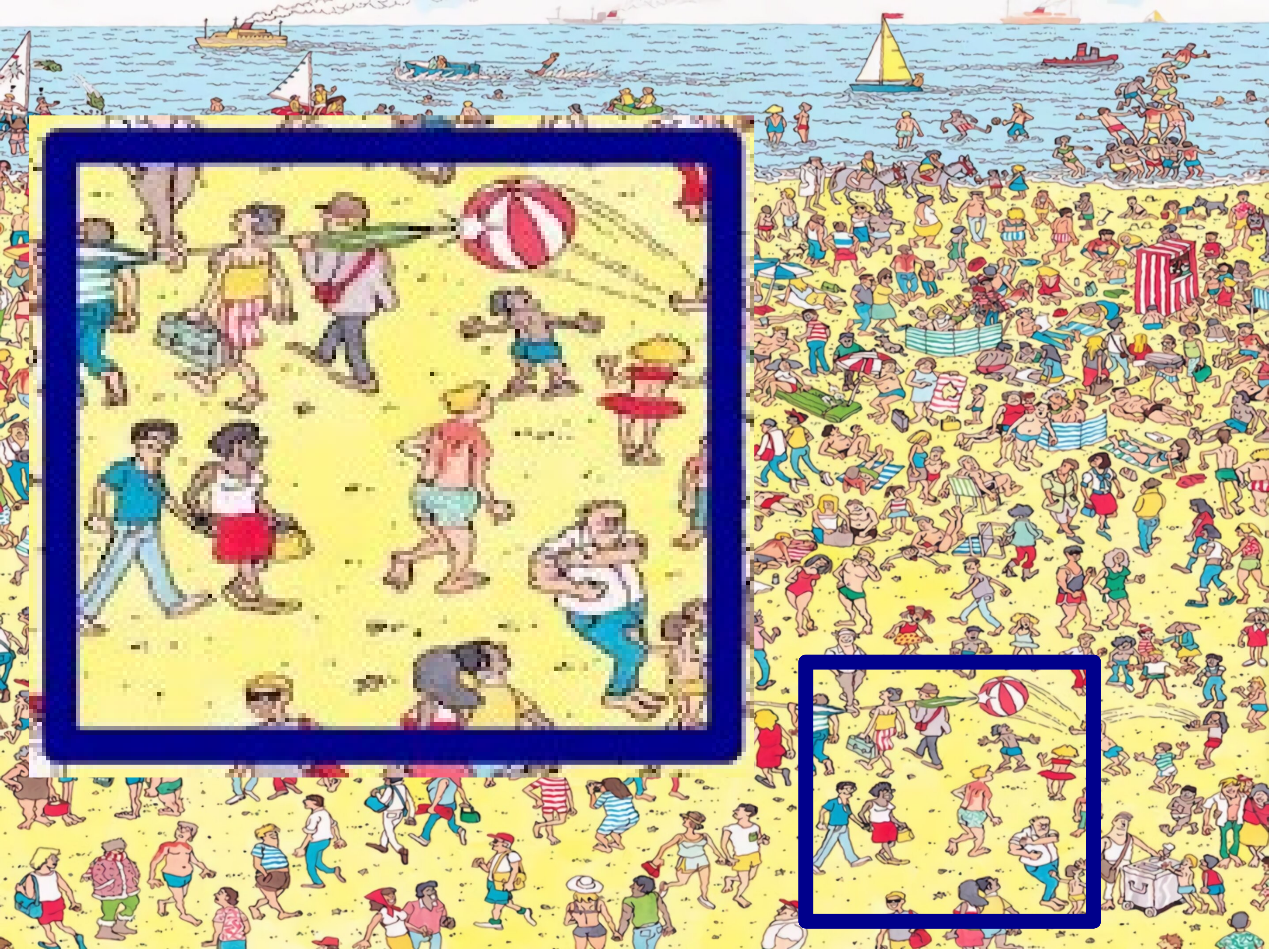
- Since **R** ≠ **RE**, there is no general way to "solve" problems in the class **RE**, if by "solve" you mean "make a computer program that can always tell you the correct answer."
- So what exactly *are* the sorts of languages in **RE**?

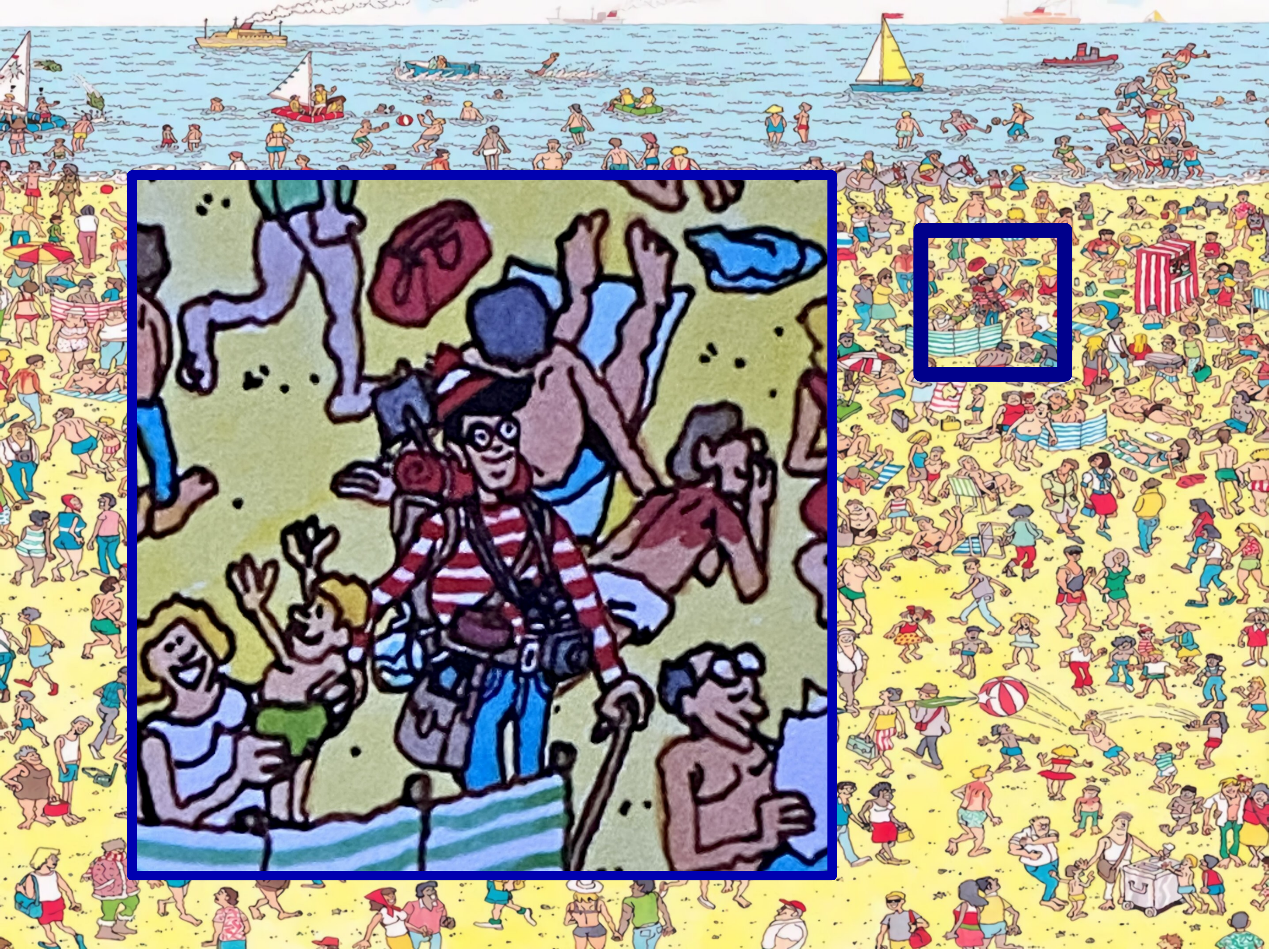
***Key Intuition:***

A language $L$ is in **RE** when, for any string $w$, if you're *convinced* that $w \in L$, there's a way you could prove that to someone else.

***Example:*** Where’s Waldo?







# Verification

**11**

Does the hailstone sequence
terminate for this number?

# Verification

**11**

Try running five steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

# Verification

**34**

Try running five steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

# Verification

**17**

Try running five steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**52**

Try running five steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

# Verification

**26**

Try running five steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**13**

Try running five steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**11**

Does the hailstone sequence
terminate for this number?

# Verification

**11**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**34**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**17**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**52**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

# Verification

**26**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**13**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**40**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

# Verification

**20**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

# Verification

**10**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**5**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

# Verification

**16**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**8**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

# Verification

**4**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

**2**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence terminate for this number?

# Verification

**1**

Try running fourteen steps of the Hailstone sequence.

Does the hailstone sequence
terminate for this number?

# Verification

$$x^3 + y^3 + z^3 = 137$$

Pick the following:

$x = 3 \quad y = -5 \quad z = 6$

Are there integers *x*, *y*, and *z* where the above statement is true?

# Verification

$$x^3 + y^3 + z^3 = 137$$

Pick the following:

$x = -9 \quad y = -11 \quad z = 13$

Are there integers *x*, *y*, and *z* where the above statement is true?

# Verification

- Here's code for simulating the hailstone sequence. No one knows whether it always terminates.

```
bool hailstone(int n) {
    if (n <= 0) return false;
    while (n != 1) {
        if (n % 2 == 0) n /= 2;
        else n = 3*n + 1;
    }
    return true;
}
```

- The following doesn't solve hailstone, but instead checks whether a given number of steps is correct. It always terminates.

```
bool checkHailstone(int n, int numSteps) {
    if (n <= 0) return false;
    for (int i = 0; i < numSteps; i++) {
        if (n % 2 == 0) n /= 2;
        else n = 3*n + 1;
    }
    return n == 1;
}
```

Note the extra parameter.

# Verification

- Here's code that searches for three cubes that sum to a target. It loops if the *n* isn't the sum of three cubes.

```
bool isCubeSum(int n) {
    for (int max = 0; ; max++)
        for (int x = -max; x <= max; x++)
            for (int y = -max; y <= max; y++)
                for (int z = -max; z <= max; z++)
                    if (x*x*x + y*y*y + z*z*z == n) return true;
}
```

- The following doesn't solve the sum of cubes problems, but instead checks whether three numbers sum to the target. It always terminates.

```
bool checkCubeSum(int n, int x, int y, int z) {
    return x*x*x + y*y*y + z*z*z == n;
}
```

Note the extra parameters.

# Verifiers

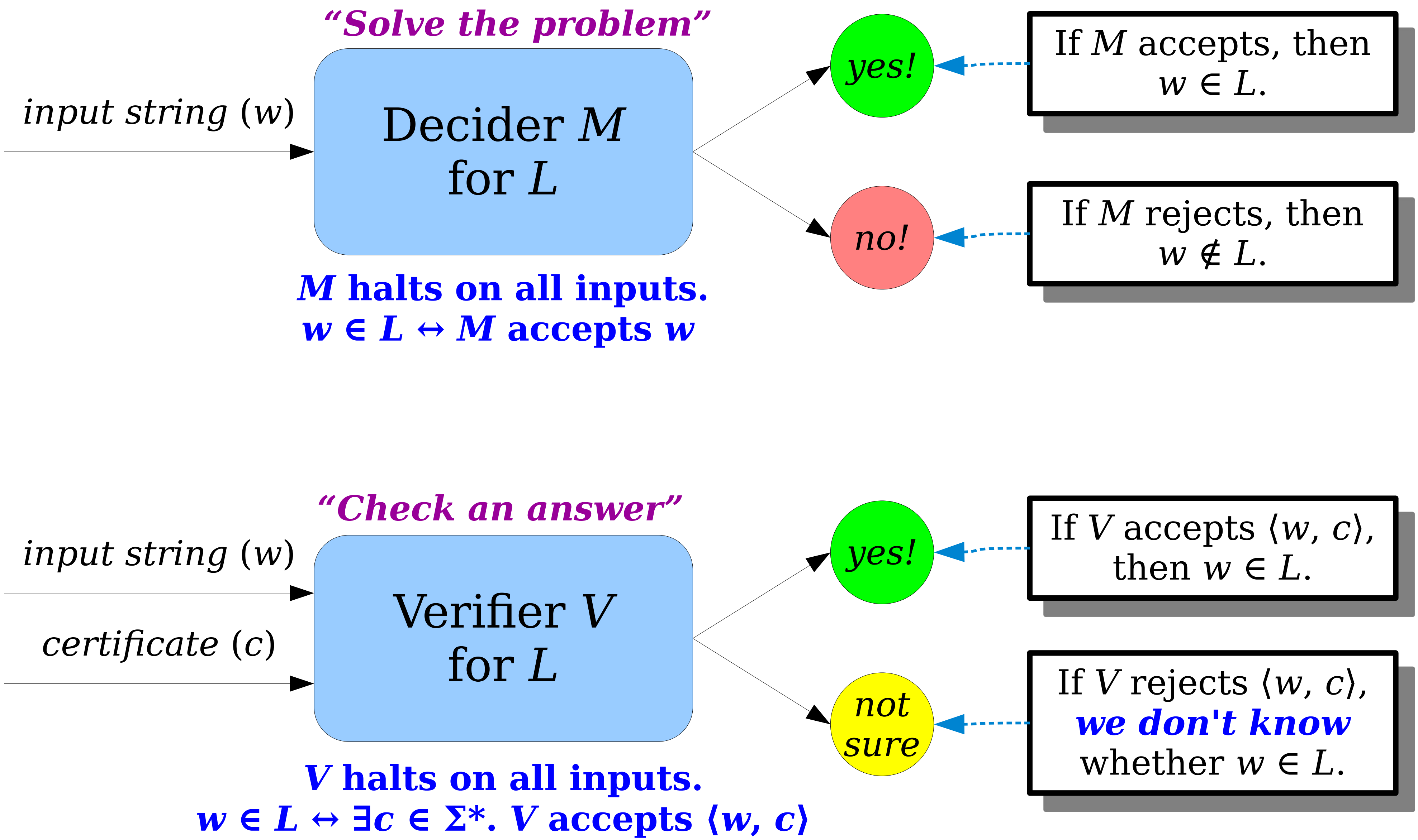
- A ***verifier*** for a language $L$ is a TM $V$ with the following two properties:

**$V$ halts on all inputs.**

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

- Intuitively, what does this mean?

# Deciders and Verifiers



```
bool checkHailstone(int n, int numSteps) {
    if (n <= 0) return false;
    for (int i = 0; i < numSteps; i++) {
        if (n % 2 == 0) n /= 2;
        else n = 3*n + 1;
    }
    return n == 1;
}


bool checkCubeSum(int n, int x, int y, int z) {
    return x*x*x + y*y*y + z*z*z == n;
}
```

# Verifiers

- A ***verifier*** for a language $L$ is a TM $V$ with the following properties:

**$V$ halts on all inputs.**

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

- Some notes about $V$:
  - If $V$ accepts $\langle w, c \rangle$, we're guaranteed $w \in L$.
  - If $V$ rejects $\langle w, c \rangle$, 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.**

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

- Some notes about $V$:
  - The certificate $c$ is existentially-quantified. Any string $w \in L$ must have at least one $c$ that causes $V$ to accept, and possibly more.
  - $V$ 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.**

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

- Some notes about $V$:
  - Although $V$ always halts, $V$ isn't a decider for $L$ and isn't a recognizer for $L$. *(Do you see why?)*
  - $V$ just checks certificates. It doesn't decide membership in $L$.

# Verifiers

- A ***verifier*** for a language $L$ is a TM $V$ with the following properties:

**$V$ halts on all inputs.**

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

- Some notes about $V$:
  - Remember that $c$ can be an encoding of some other object or objects.
  - In practice, $c$ will likely just be "some other auxiliary data that helps you out."

# What languages are verifiable?

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

***Proof:*** Appendix!

# **RE** and Proofs

- Verifiers and recognizers give two different perspectives on the “proof” intuition for **RE**.
- A verifier $V$ for $L$ checks proofs that $w \in L$.
  - If $w \in L$, there’s a proof $c$ where $V$ accepts $\langle w, c\rangle$
  - If $w \notin L$, then $V$ never accepts any certificate for $w$.
- A recognizer $R$ for $L$ searches for proof that $w \in L$.
  - If $w \in L$, then $R$ finds a proof and accepts.
  - If $w \notin L$, then $R$ never finds a proof and loops.
    - Or perhaps it finds a proof that $w \notin L$ and rejects.

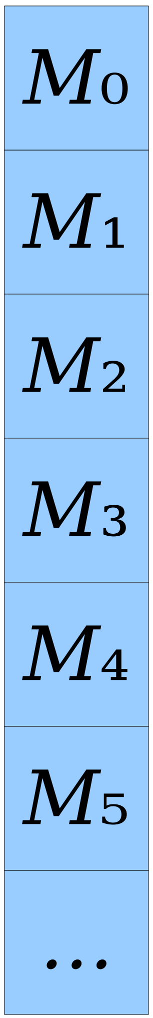
# Finding Non-**RE** Languages

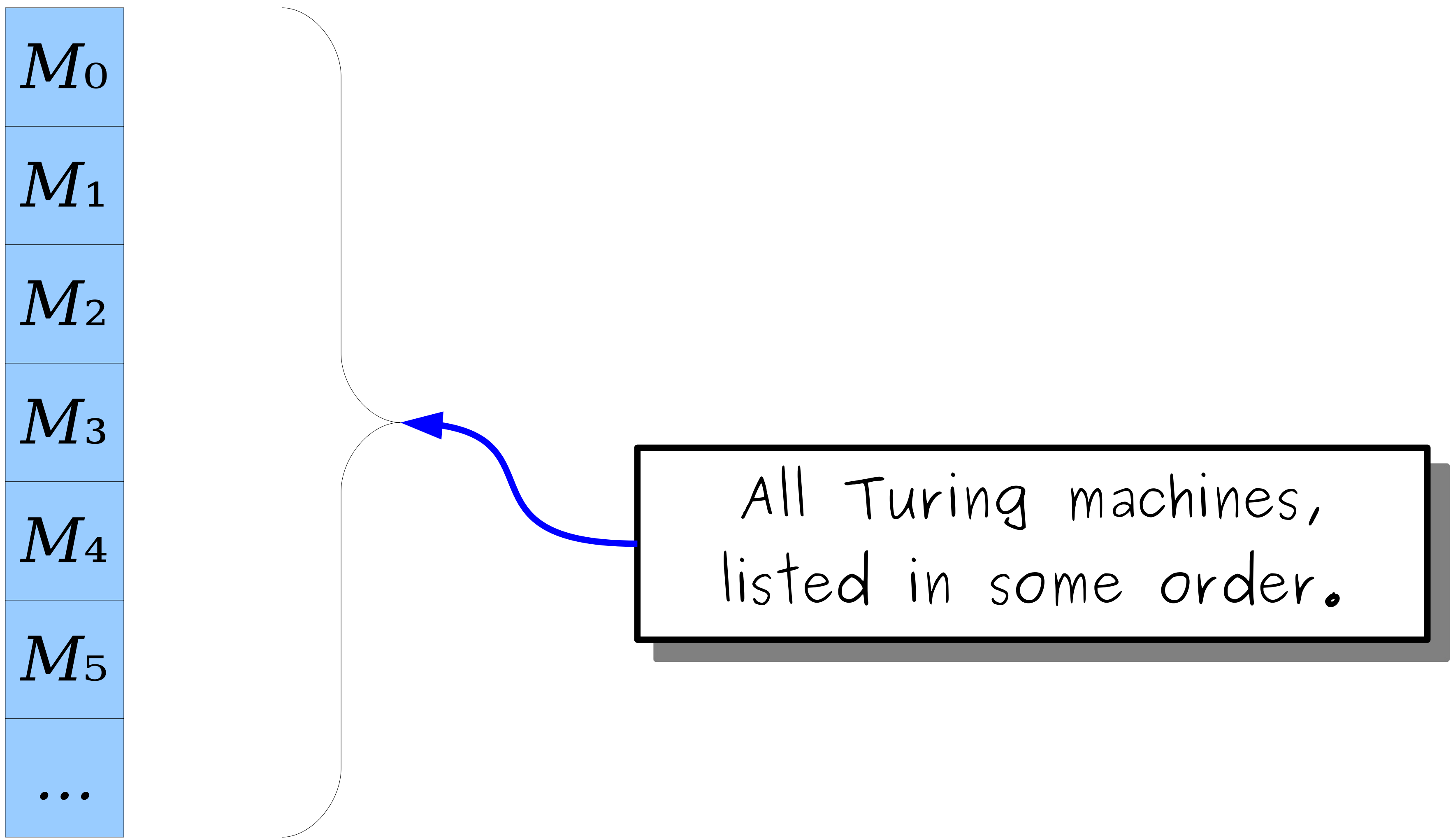
# Recognizers and Recognizability

- ***Recall:*** We say that *M* is a recognizer for *L* if the following is true:

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

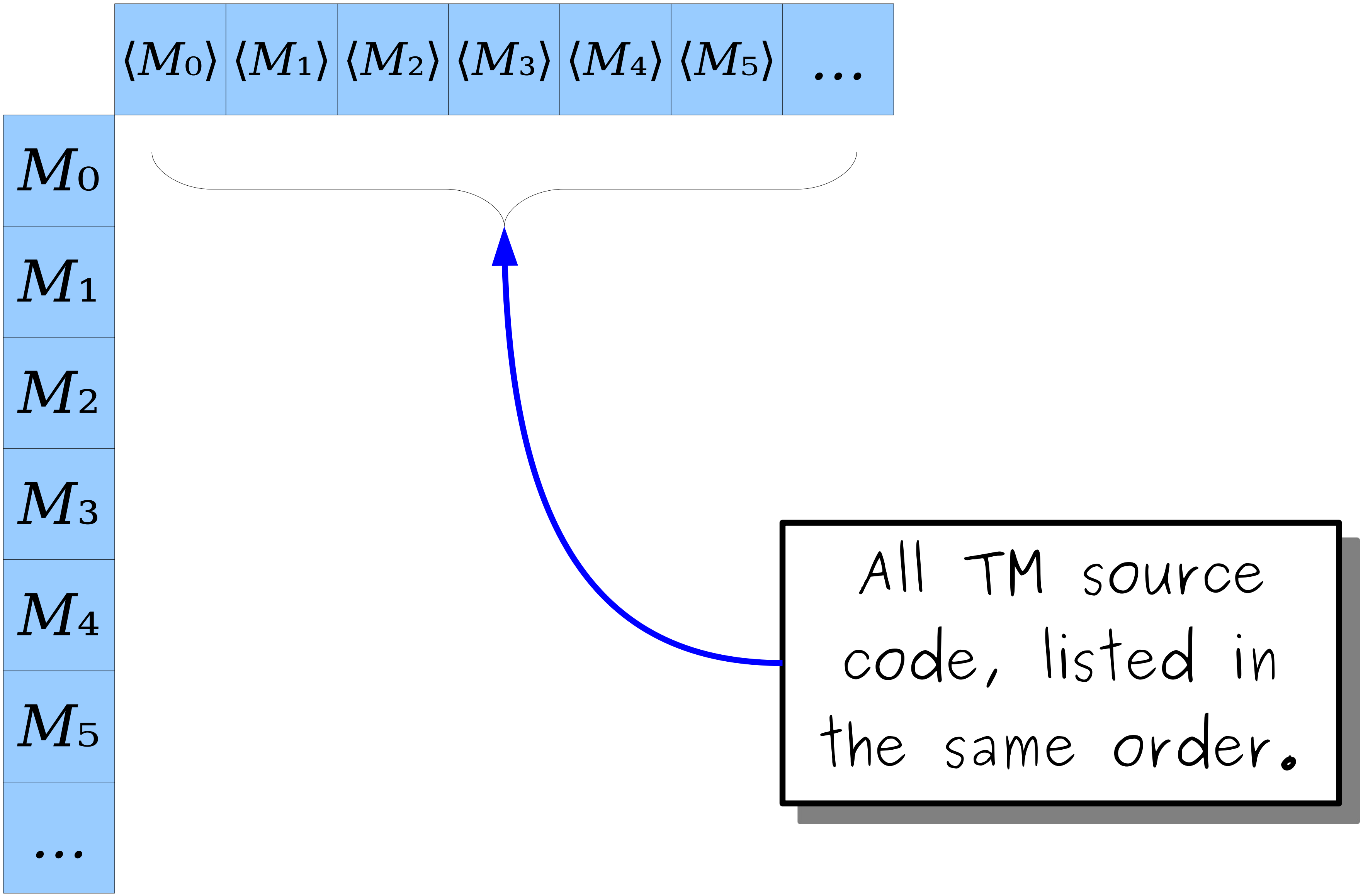
- Some of these strings *w*, by pure coincidence, will be encodings of Turing machines.
- What happens if we list off all Turing machines, looking at how those TMs behave given other TMs as input?


$M_0$
$M_1$
$M_2$
$M_3$
$M_4$
$M_5$
...



$M_0$
$M_1$
$M_2$
$M_3$
$M_4$
$M_5$
...
All Turing machines, listed in some order.

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | $\dots$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | | | | | | | |
| $M_1$ | | | | | | | |
| $M_2$ | | | | | | | |
| $M_3$ | | | | | | | |
| $M_4$ | | | | | | | |
| $M_5$ | | | | | | | |
| $\dots$ | | | | | | | |



⟨$M_0$⟩ ⟨$M_1$⟩ ⟨$M_2$⟩ ⟨$M_3$⟩ ⟨$M_4$⟩ ⟨$M_5$⟩ ...
$M_0$
$M_1$
$M_2$
$M_3$
$M_4$
$M_5$
...
All TM source code, listed in the same order.

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | $\ldots$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | ... |
| $M_1$ | | | | | | | |
| $M_2$ | | | | | | | |
| $M_3$ | | | | | | | |
| $M_4$ | | | | | | | |
| $M_5$ | | | | | | | |
| $\ldots$ | | | | | | | |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | $\dots$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | ... |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_2$ | | | | | | | |
| $M_3$ | | | | | | | |
| $M_4$ | | | | | | | |
| $M_5$ | | | | | | | |
| $\dots$ | | | | | | | |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | $\ldots$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | ... |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_3$ | | | | | | | |
| $M_4$ | | | | | | | |
| $M_5$ | | | | | | | |
| $\ldots$ | | | | | | | |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | $\dots$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | ... |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | ... |
| $M_4$ | | | | | | | |
| $M_5$ | | | | | | | |
| $\dots$ | | | | | | | |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | $\ldots$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | ... |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | ... |
| $M_4$ | Acc | No | Acc | No | Acc | No | ... |
| $M_5$ | | | | | | | |
| $\ldots$ | | | | | | | |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | $\dots$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | ... |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | ... |
| $M_4$ | Acc | No | Acc | No | Acc | No | ... |
| $M_5$ | No | No | Acc | Acc | No | No | ... |
| $\dots$ | | | | | | | |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | $\dots$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | ... |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | ... |
| $M_4$ | Acc | No | Acc | No | Acc | No | ... |
| $M_5$ | No | No | Acc | Acc | No | No | ... |
| $\dots$ | ... | ... | ... | ... | ... | ... | ... |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | $\dots$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | ... |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | ... |
| $M_4$ | Acc | No | Acc | No | Acc | No | ... |
| $M_5$ | No | No | Acc | Acc | No | No | ... |
| $\dots$ | ... | ... | ... | ... | ... | ... | ... |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | $\ldots$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | ... |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | ... |
| $M_4$ | Acc | No | Acc | No | Acc | No | ... |
| $M_5$ | No | No | Acc | Acc | No | No | ... |
| $\ldots$ | ... | ... | ... | ... | ... | ... | ... |

| | | | | | | |
|---|---|---|---|---|---|---|
| Acc | Acc | Acc | No | Acc | No | ... |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

| | | | | | | |
|---|---|---|---|---|---|---|
| Acc | Acc | Acc | No | Acc | No | … |

What are we going to do next?

Answer at https://cs103.stanford.edu/pollev

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

| | | | | | | |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | … |

Flip all "accept" to "no" and vice-versa

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | $\dots$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | ... |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | ... |
| $M_4$ | Acc | No | Acc | No | Acc | No | ... |
| $M_5$ | No | No | Acc | Acc | No | No | ... |
| $\dots$ | ... | ... | ... | ... | ... | ... | ... |

| | | | | | | |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | ... |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | $\dots$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | ... |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | ... |
| $M_4$ | Acc | No | Acc | No | Acc | No | ... |
| $M_5$ | No | No | Acc | Acc | No | No | ... |
| $\dots$ | ... | ... | ... | ... | ... | ... | ... |

| | | | | | | |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | ... |

What TM has this behavior?

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | $\dots$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | ... |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | ... |
| $M_4$ | Acc | No | Acc | No | Acc | No | ... |
| $M_5$ | No | No | Acc | Acc | No | No | ... |
| $\dots$ | ... | ... | ... | ... | ... | ... | ... |

| | | | | | | |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | ... |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | $\dots$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | ... |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | ... |
| $M_4$ | Acc | No | Acc | No | Acc | No | ... |
| $M_5$ | No | No | Acc | Acc | No | No | ... |
| $\dots$ | ... | ... | ... | ... | ... | ... | ... |

| | | | | | | |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | ... |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | $\ldots$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | ... |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | ... |
| $M_4$ | Acc | No | Acc | No | Acc | No | ... |
| $M_5$ | No | No | Acc | Acc | No | No | ... |
| $\ldots$ | ... | ... | ... | ... | ... | ... | ... |

| | | | | | | |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | ... |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | $\ldots$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | ... |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | ... |
| $M_4$ | Acc | No | Acc | No | Acc | No | ... |
| $M_5$ | No | No | Acc | Acc | No | No | ... |
| $\ldots$ | ... | ... | ... | ... | ... | ... | ... |

| | | | | | | |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | ... |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | $\dots$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | ... |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | ... |
| $M_4$ | Acc | No | Acc | No | Acc | No | ... |
| $M_5$ | No | No | Acc | Acc | No | No | ... |
| $\dots$ | ... | ... | ... | ... | ... | ... | ... |

| | | | | | | |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | ... |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | $\ldots$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | ... |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | ... |
| $M_4$ | Acc | No | Acc | No | Acc | No | ... |
| $M_5$ | No | No | Acc | Acc | No | No | ... |
| $\ldots$ | ... | ... | ... | ... | ... | ... | ... |

| | | | | | | |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | ... |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | $\ldots$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | ... |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | ... |
| $M_4$ | Acc | No | Acc | No | Acc | No | ... |
| $M_5$ | No | No | Acc | Acc | No | No | ... |
| $\ldots$ | ... | ... | ... | ... | ... | ... | ... |

| | | | | | | |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | ... |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | $\dots$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | ... |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | ... |
| $M_4$ | Acc | No | Acc | No | Acc | No | ... |
| $M_5$ | No | No | Acc | Acc | No | No | ... |
| $\dots$ | ... | ... | ... | ... | ... | ... | ... |

| | | | | | | |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | ... |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | $\dots$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | ... |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | ... |
| $M_4$ | Acc | No | Acc | No | Acc | No | ... |
| $M_5$ | No | No | Acc | Acc | No | No | ... |
| $\dots$ | ... | ... | ... | ... | ... | ... | ... |

| | | | | | | |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | ... |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | $\dots$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | ... |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | ... |
| $M_4$ | Acc | No | Acc | No | Acc | No | ... |
| $M_5$ | No | No | Acc | Acc | No | No | ... |
| $\dots$ | ... | ... | ... | ... | ... | ... | ... |

| | | | | | | |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | ... |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | **Acc** | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | **Acc** | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | **Acc** | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | **No** | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | **Acc** | No | … |
| $M_5$ | No | No | Acc | Acc | No | **No** | … |
| … | … | … | … | … | … | … | **…** |

| | | | | | | |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | … |

No TM has this behavior!

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | $\ldots$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | ... |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | ... |
| $M_4$ | Acc | No | Acc | No | Acc | No | ... |
| $M_5$ | No | No | Acc | Acc | No | No | ... |
| $\ldots$ | ... | ... | ... | ... | ... | ... | ... |

| | | | | | | |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | ... |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | $\ldots$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | ... |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | ... |
| $M_4$ | Acc | No | Acc | No | Acc | No | ... |
| $M_5$ | No | No | Acc | Acc | No | No | ... |
| $\ldots$ | ... | ... | ... | ... | ... | ... | ... |

| | | | | | | |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | ... |

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | ... |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | ... |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | ... |
| $M_4$ | Acc | No | Acc | No | Acc | No | ... |
| $M_5$ | No | No | Acc | Acc | No | No | ... |
| … | ... | ... | ... | ... | ... | ... | ... |

| | | | | | | |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | ... |

**"The language of all TMs that do not accept their encodings."**

| | $\langle M_0 \rangle$ | $\langle M_1 \rangle$ | $\langle M_2 \rangle$ | $\langle M_3 \rangle$ | $\langle M_4 \rangle$ | $\langle M_5 \rangle$ | … |
|---|---|---|---|---|---|---|---|
| $M_0$ | Acc | No | No | Acc | Acc | No | … |
| $M_1$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_2$ | Acc | Acc | Acc | Acc | Acc | Acc | … |
| $M_3$ | No | Acc | Acc | No | Acc | Acc | … |
| $M_4$ | Acc | No | Acc | No | Acc | No | … |
| $M_5$ | No | No | Acc | Acc | No | No | … |
| … | … | … | … | … | … | … | … |

| | | | | | | |
|---|---|---|---|---|---|---|
| No | No | No | Acc | No | Acc | … |

**{ $\langle M \rangle$ | $M$ is a TM that does not accept $\langle M \rangle$ }**

# Diagonalization Revisited

- The ***diagonalization language***, which we denote $\boldsymbol{L_{\text{D}}}$, is defined as

$$\boldsymbol{L_{\text{D}} = \{ \langle M \rangle \mid M \textbf{ is a TM and } M \textbf{ does not accept } \langle M \rangle \}}$$

- We constructed this language to be different from the language of every TM.
- Therefore, $L_{\text{D}} \notin \textbf{RE}$! Let's go prove this.

## $L_D = \{ \langle M \rangle \mid M$ is a TM and $M$ does not accept $\langle M \rangle \}$

***Theorem:*** $L_D \notin \mathbf{RE}$.

***Proof:*** Assume for the sake of contradiction that $L_D \in \mathbf{RE}$. This means that there is a recognizer $R$ for $L_D$.

What happens if we run $R$ on $\langle R \rangle$? Since $R$ recognizes $L_D$, we know that

$$R \text{ accepts } \langle R \rangle \quad \text{if and only if} \quad \langle R \rangle \in L_D.$$

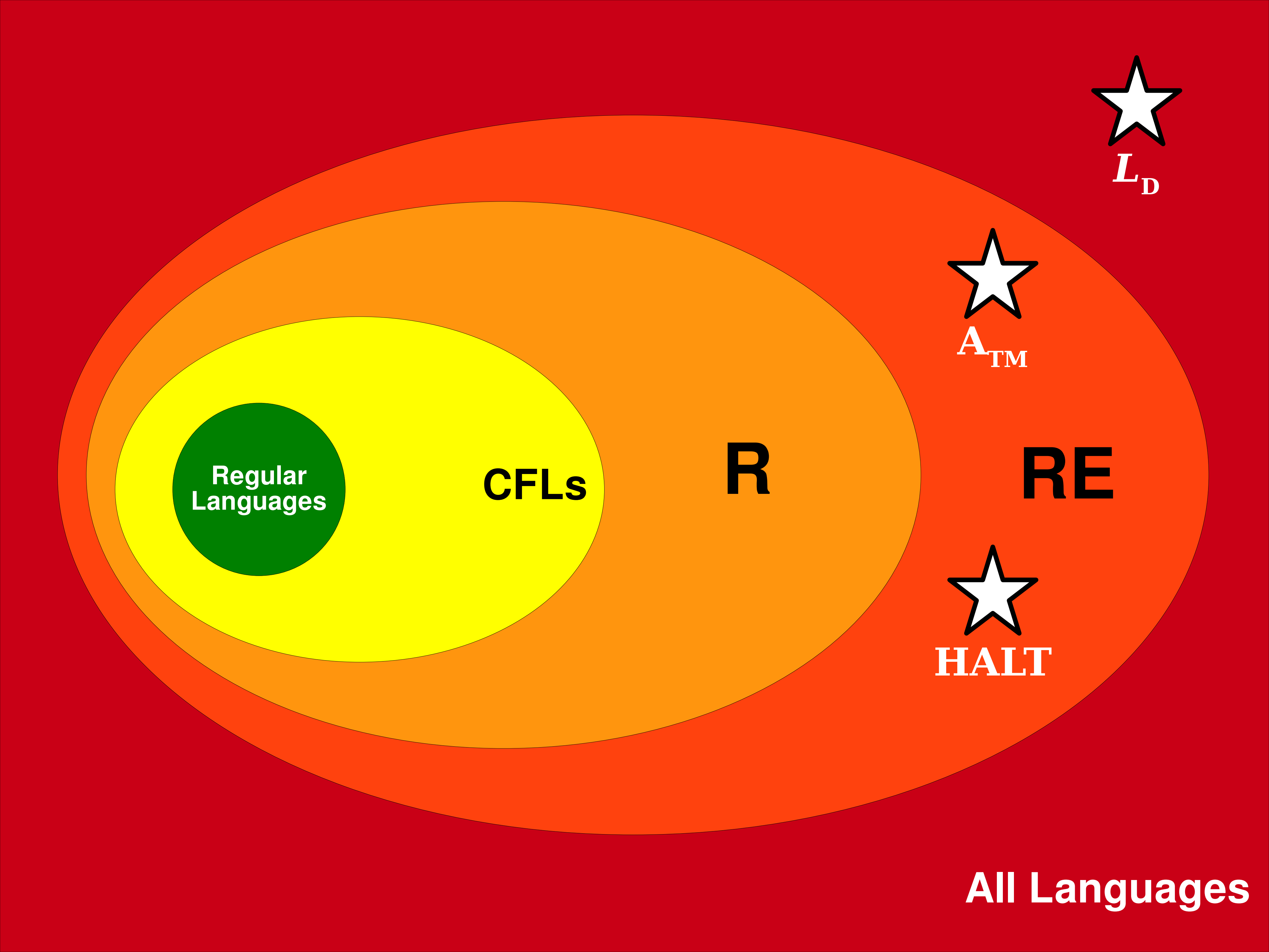
By definition of $L_D$, we also know that

$$\langle R \rangle \in L_D \quad \text{if and only if} \quad R \text{ does not accept } \langle R \rangle.$$

Combining the two above statements tells us that

$$R \text{ accepts } \langle R \rangle \quad \text{if and only if} \quad R \text{ does not accept } \langle R \rangle.$$

This is impossible. We've reached a contradiction, so our assumption was wrong, and so $L_D \notin \mathbf{RE}$. ■



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

# What This Means

- On a deeper philosophical level, the fact that non-**RE** languages exist supports the following claim:

  ***There are statements that are true but not provable.***

- This result can be formalized as a result called ***Gödel's incompleteness theorem***, one of the most important mathematical results of all time.
- Want to learn more? Take Phil 152 or CS154!

# What This Means

- On a more philosophical note, you could interpret the previous result in the following way:

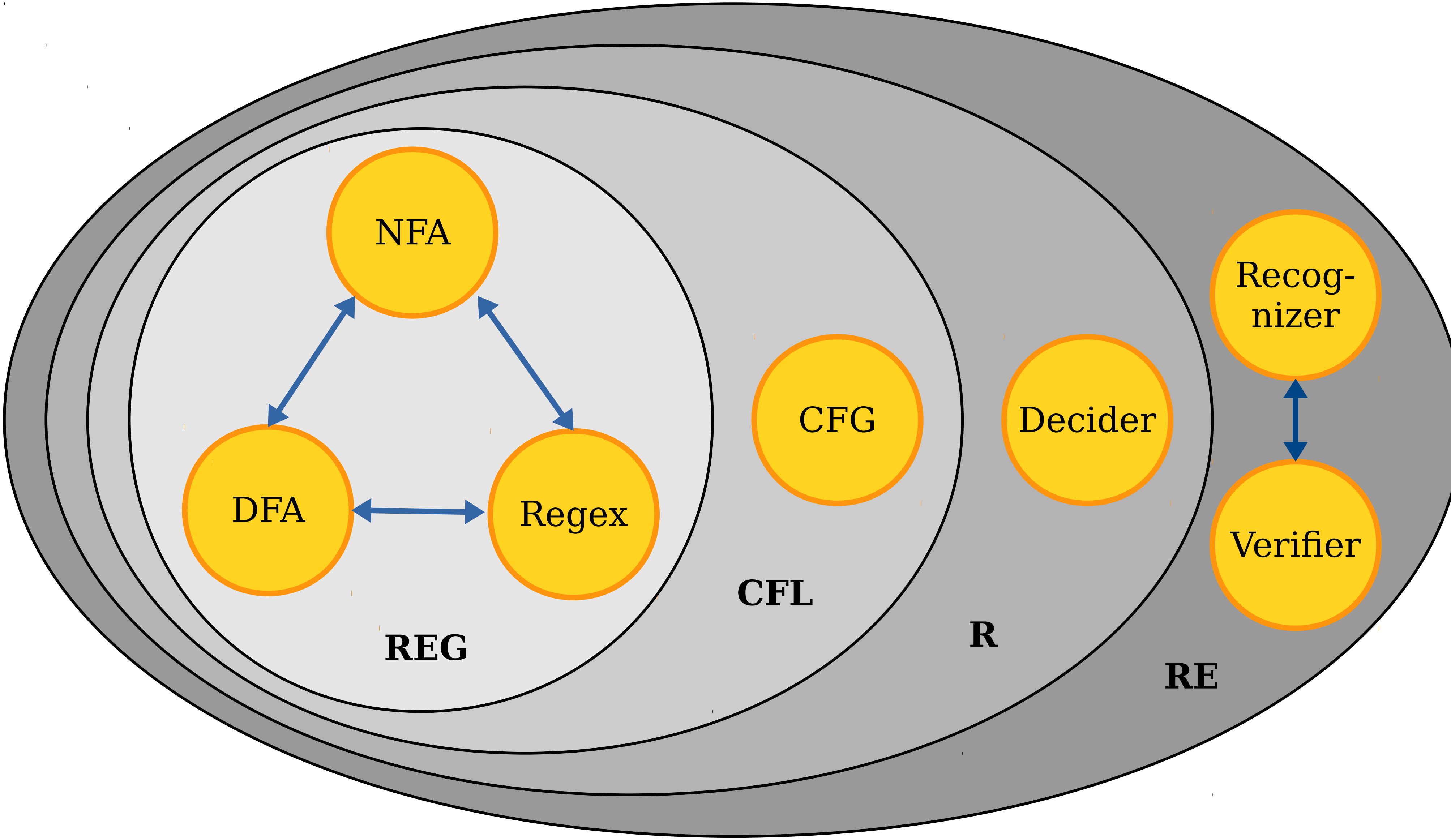
***There are inherent limits about what mathematics can teach us.***

- There's no automatic way to do math. There are true statements that we can't prove.
- That doesn't mean that mathematics is worthless. It just means that we need to temper our expectations about it.

# Where We Stand

- We've just done a whirlwind tour of computability theory:
  - ***The Church-Turing thesis*** tells us that TMs give us a mechanism for studying computation in the abstract.
  - ***Universal computers*** – computers as we know them – are not just a stroke of luck. The existence of the universal TM ensures that such computers must exist.
  - ***Self-reference*** is an inherent consequence of computational power.
  - ***Undecidable problems*** exist partially as a consequence of the above and indicate that there are statements whose truth can't be determined by computational processes.
  - ***Unrecognizable problems*** are out there and can be discovered via diagonalization. They imply there are limits to mathematical proof.

# The Big Picture



# Where We've Been

- The class **R** represents problems that can be solved by a computer.
- The class **RE** represents problems where “yes” answers can be verified by a computer.

# Where We're Going

- The class **P** represents problems that can be solved *efficiently* by a computer.
- The class **NP** represents problems where "yes" answers can be verified *efficiently* by a computer.

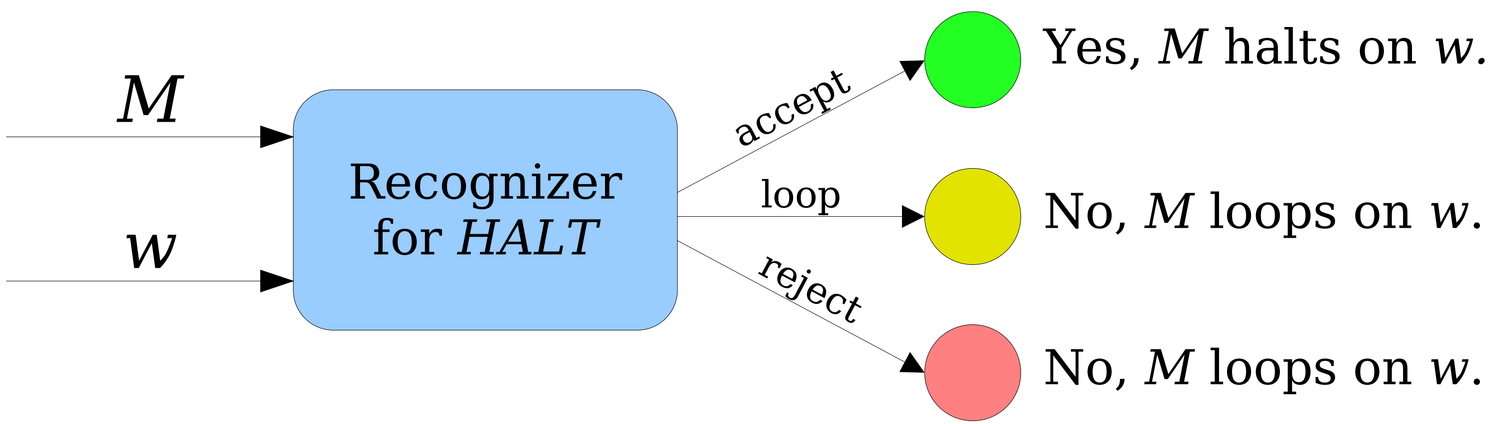
# Next Time

- ***Introduction to Complexity Theory***
  - Not all decidable problems are created equal!
- ***The Classes P and NP***
  - Two fundamental and important complexity classes.
- ***The $P \stackrel{?}{=} NP$ Question***
  - A literal million-dollar question!

# ***Appendix 1:*** $HALT \in \mathbf{RE}$

# *HALT* ∈ **RE**

- The halting problem is recognizable, meaning there's a recognizer for it.
- That recognizer would have the following abstract behavior:



# *HALT* ∈ **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!
- Here's that idea expressed as a recognizer:

```
bool recognizeIfHalts(string TM, string w) {
    set up a simulation of M running on w;
    while (true) {
        if (M returned true) return true;
        else if (M returned false) return true;
        else simulate one more step of M running on w;
    }
}
```

# $HALT \in \mathbf{RE}$

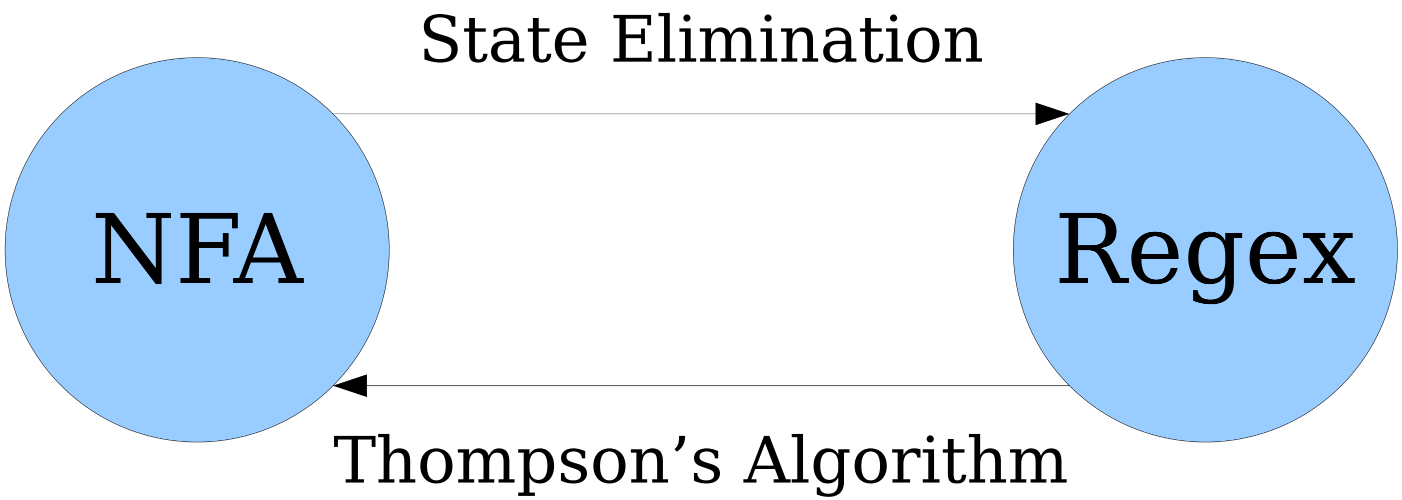
- How might we build a verifier for *HALT*?
- ***Idea:*** If a TM *M* halts on a string *w*, it must do so within some number of steps.
- Our verifier can then run *M* on *w* for that many steps and see if it halts:

```
bool checkAccepts(TM M, string w, int n) {
    set up a simulation of M running on w;
    for (int i = 0; i < n; i++) {
        simulate one more step of M running on w;
    }
    return whether M halted;
}
```

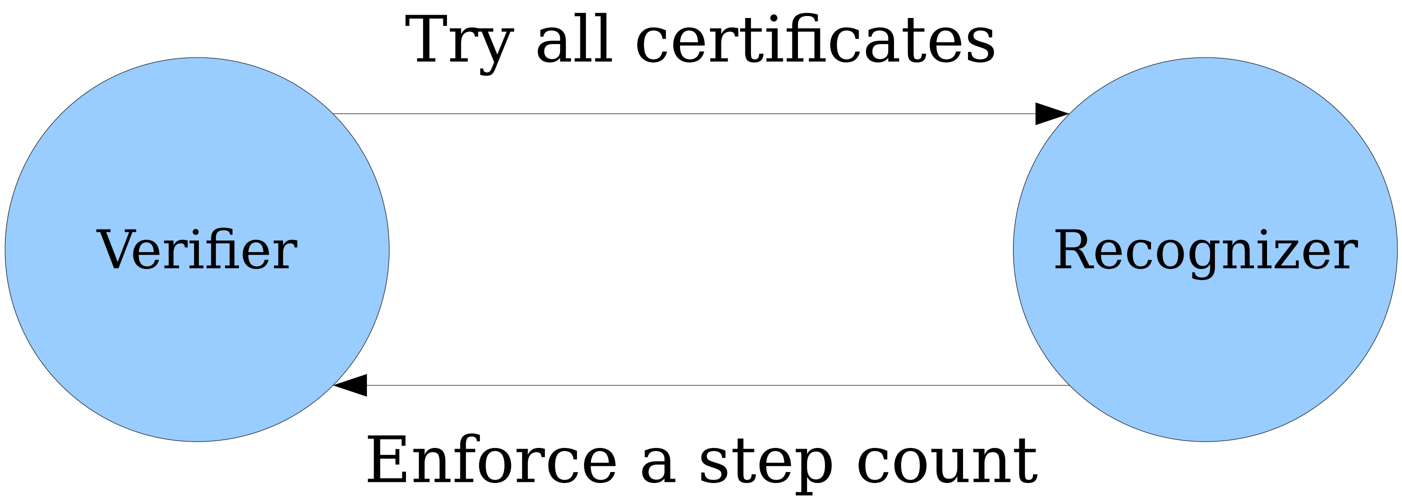
# ***Appendix 2:*** Verifiers and **RE** Languages

***Theorem:*** Let $L$ be a language. Then $L \in \mathbf{RE}$ if and only if there is a verifier $V$ for $L$.

# Where We've Been

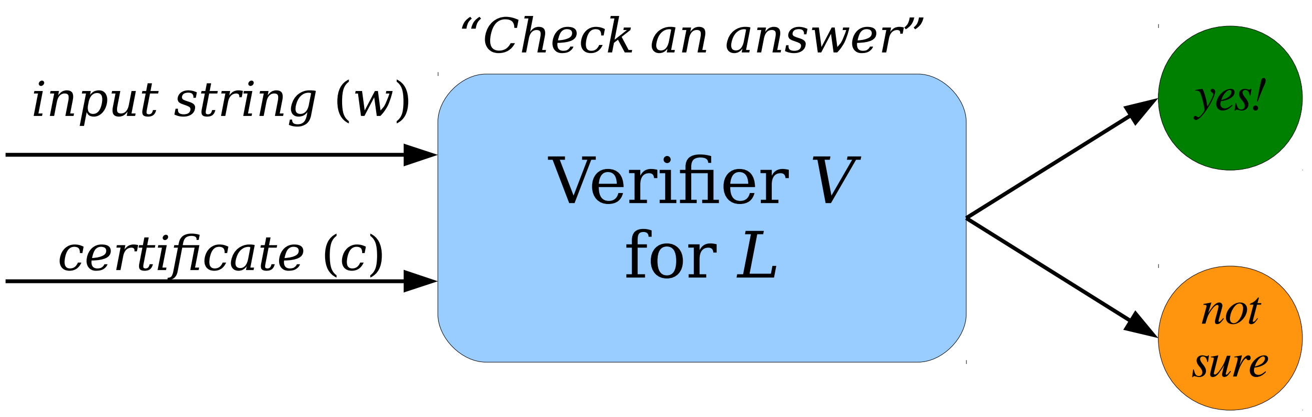


# Where We're Going



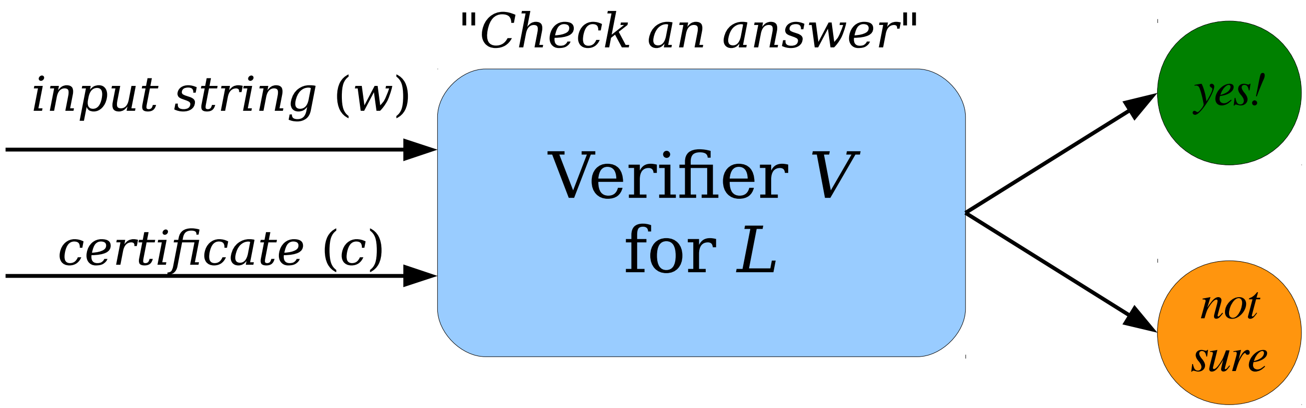
# Verifiers and **RE**

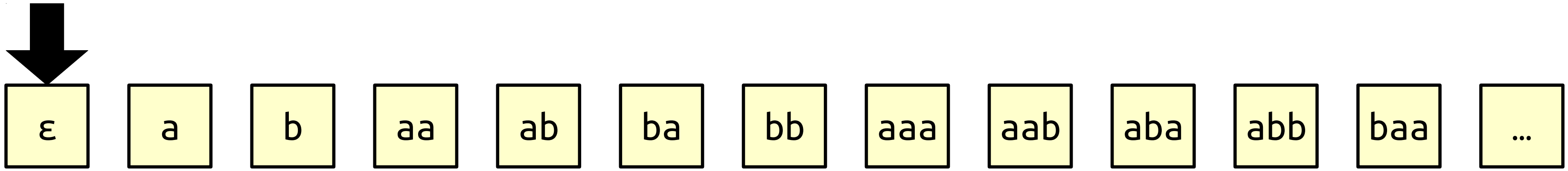
- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.



# Verifiers and **RE**

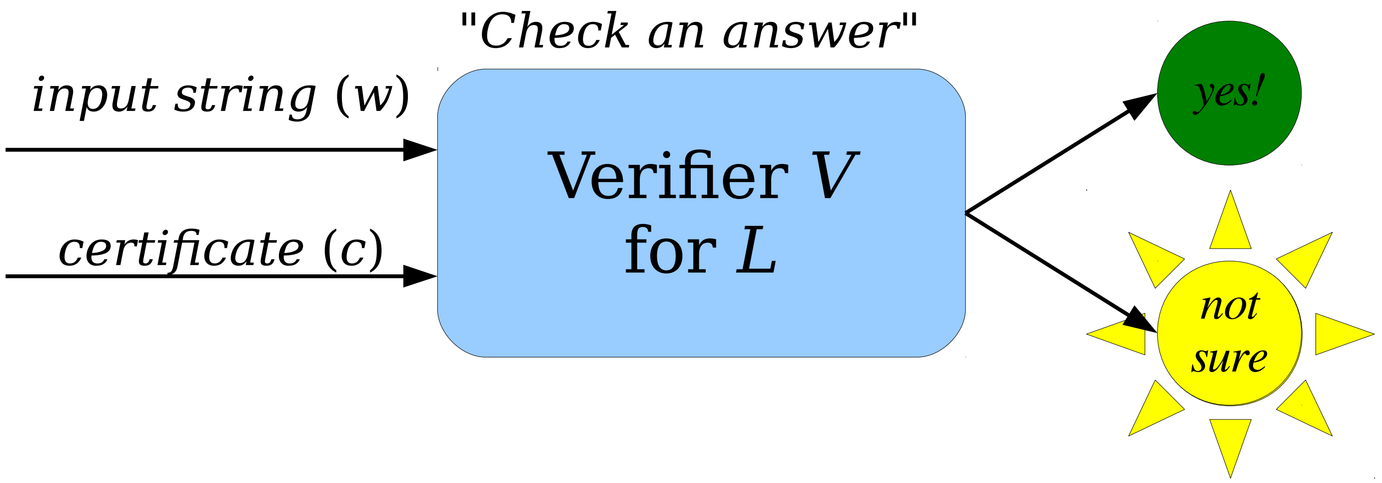
- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in$ **RE**.
- ***Proof goal:*** Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.

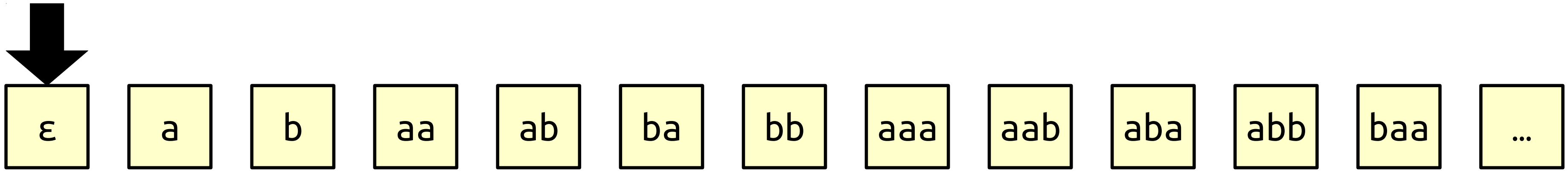


# Verifiers and **RE**

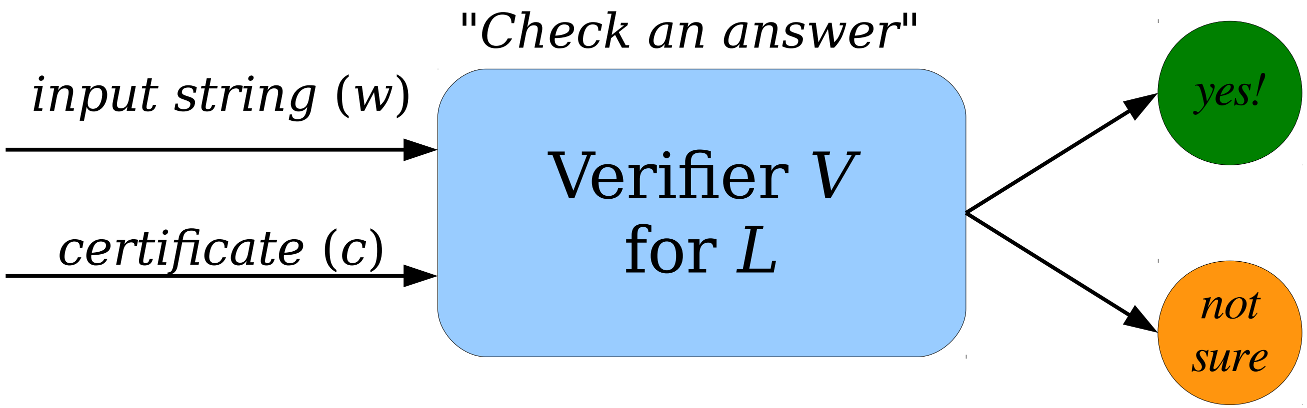
- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in$ **RE**.
- ***Proof goal:*** Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.

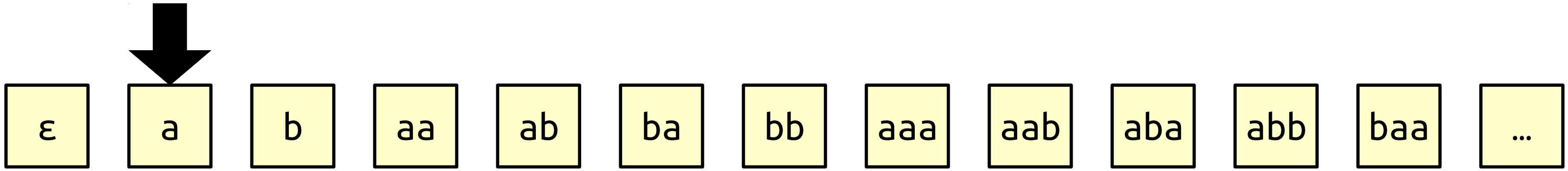


# Verifiers and **RE**

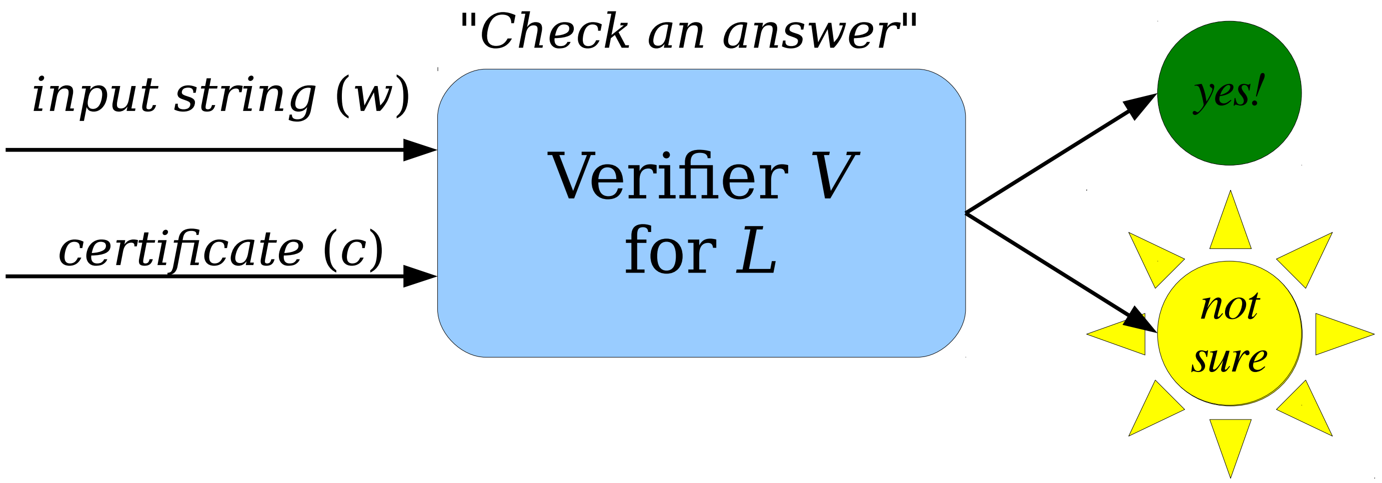
- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.

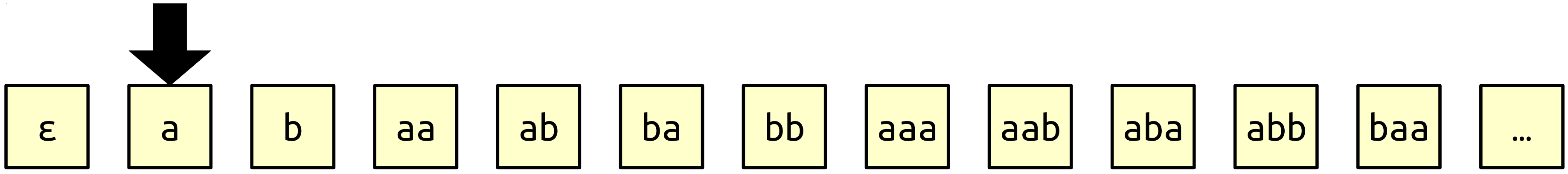


# Verifiers and **RE**

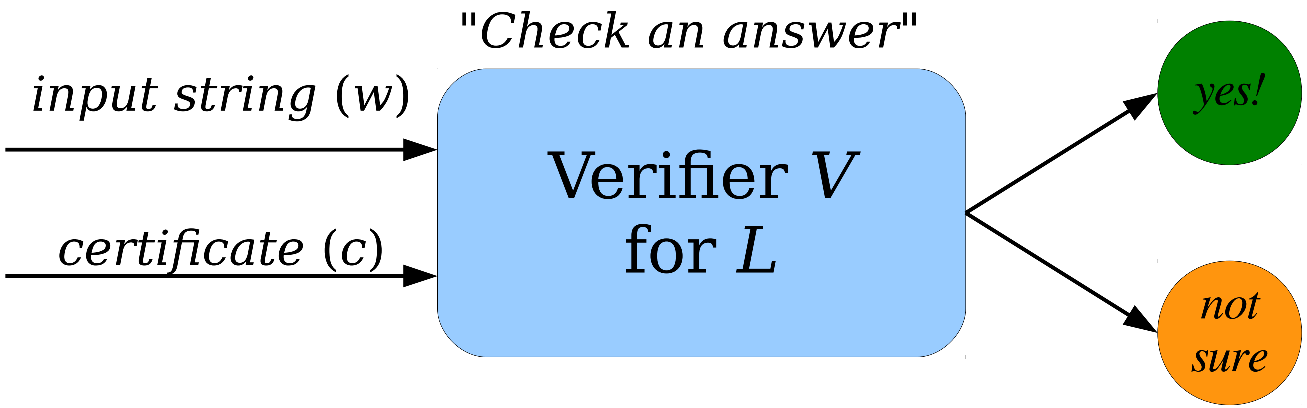
- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in$ **RE**.
- ***Proof goal:*** Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.

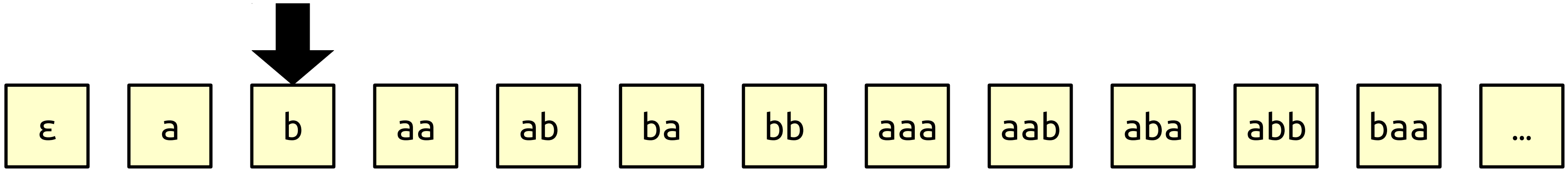


# Verifiers and **RE**

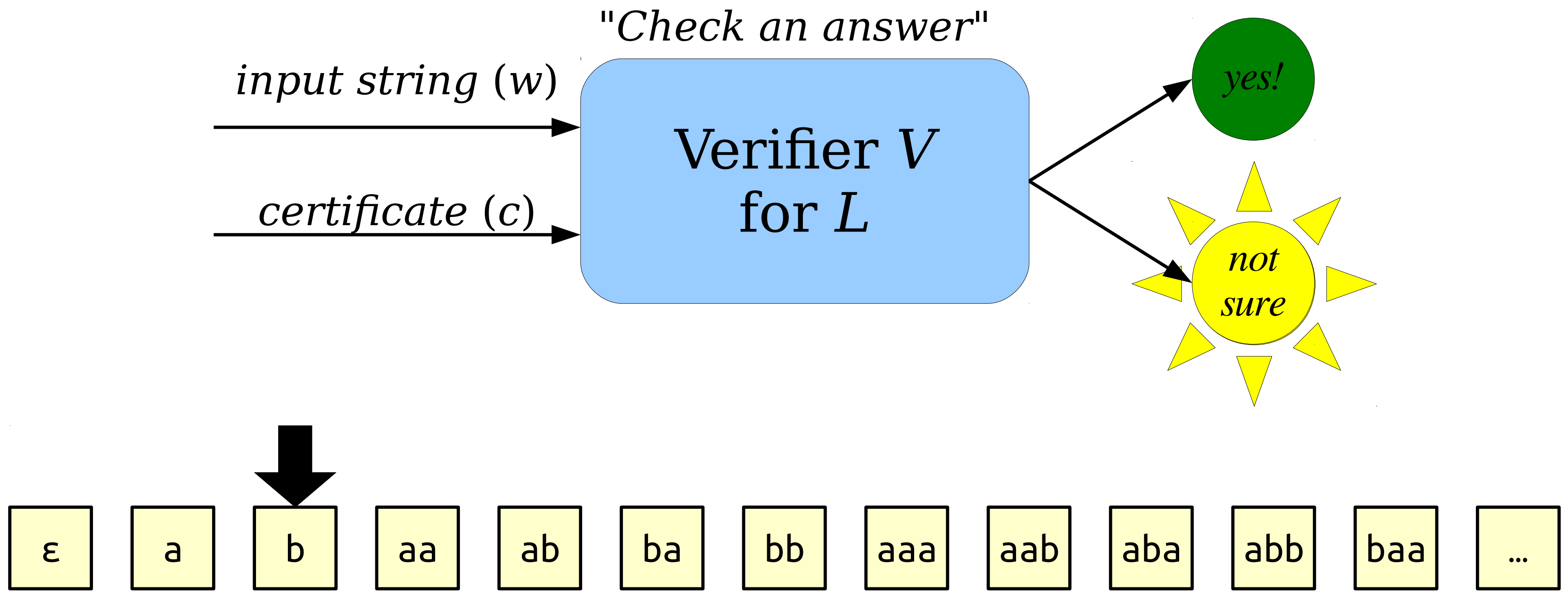
- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in$ **RE**.
- ***Proof goal:*** Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.



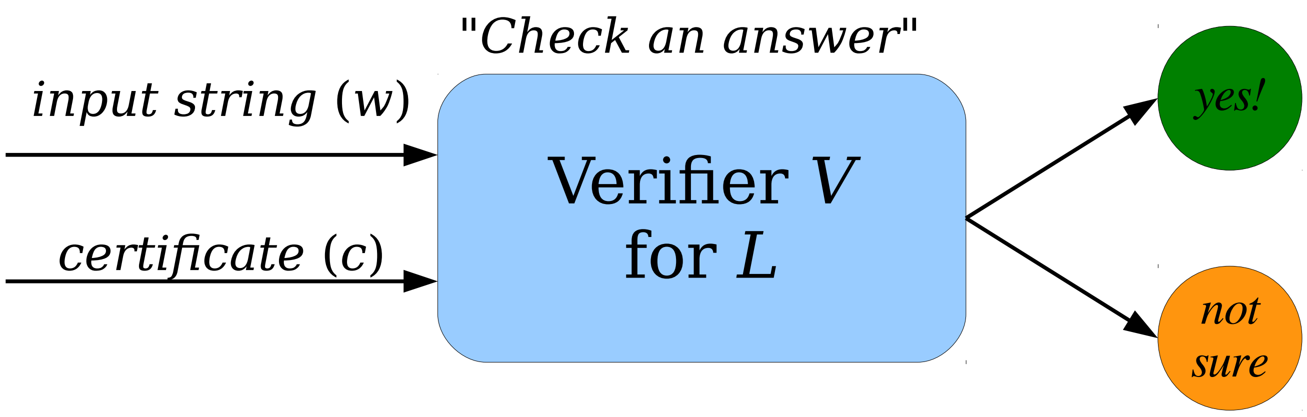
# Verifiers and **RE**

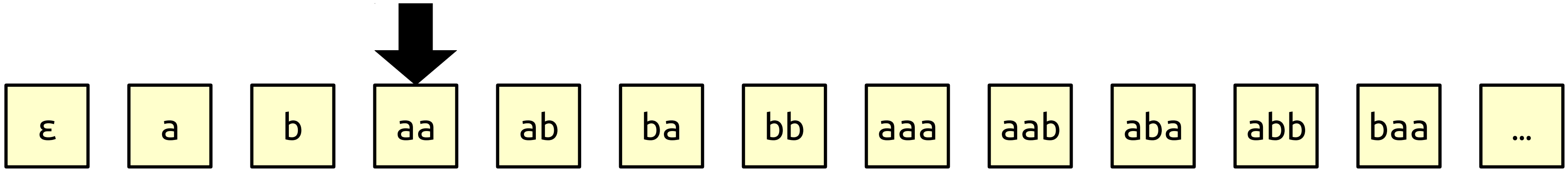
- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in$ **RE**.
- ***Proof goal:*** Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.



# Verifiers and **RE**

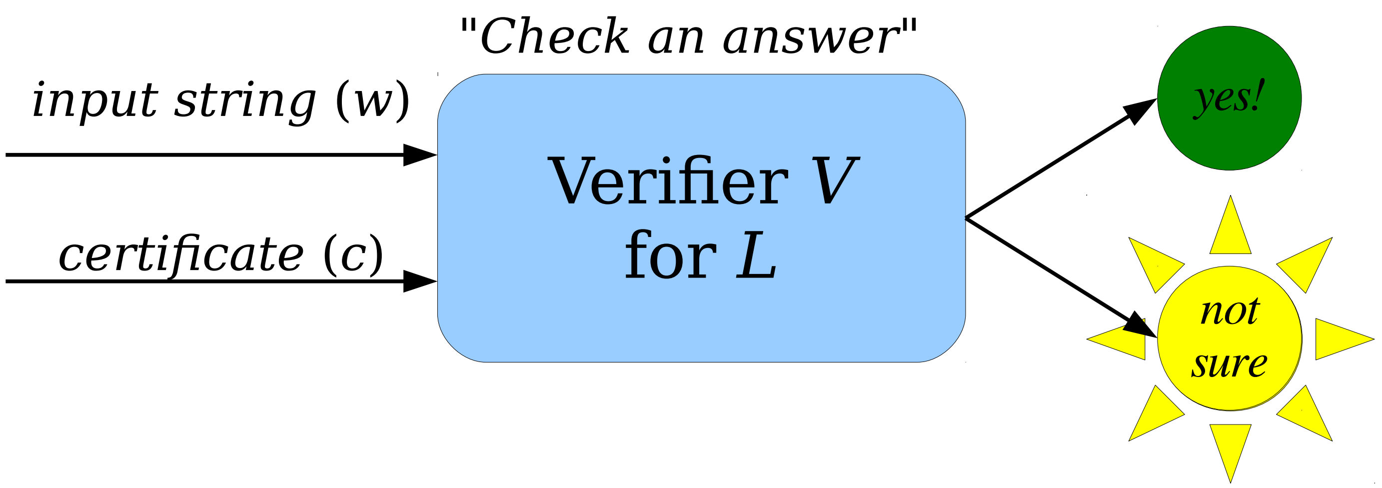
- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in$ **RE**.
- ***Proof goal:*** Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.

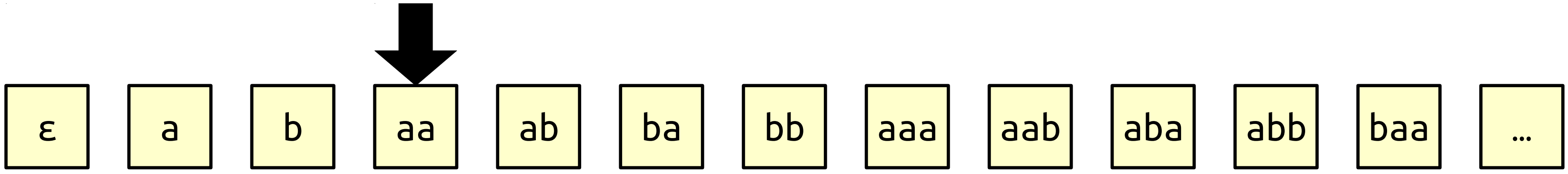


# Verifiers and **RE**

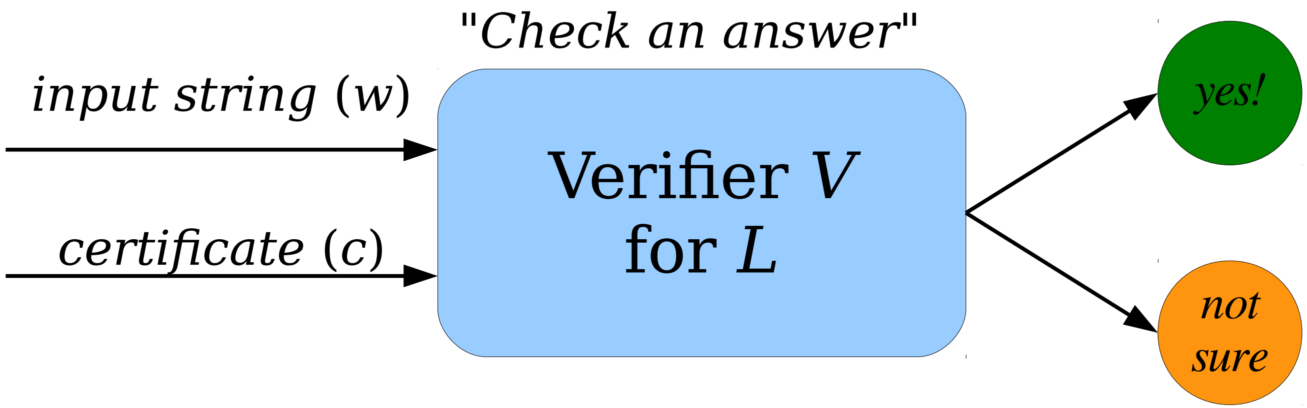
- ***Theorem:*** If there is a verifier *V* for a language *L*, then *L* ∈ **RE**.
- ***Proof goal:*** Given a verifier *V* for a language *L*, find a way to construct a recognizer *M* for *L*.

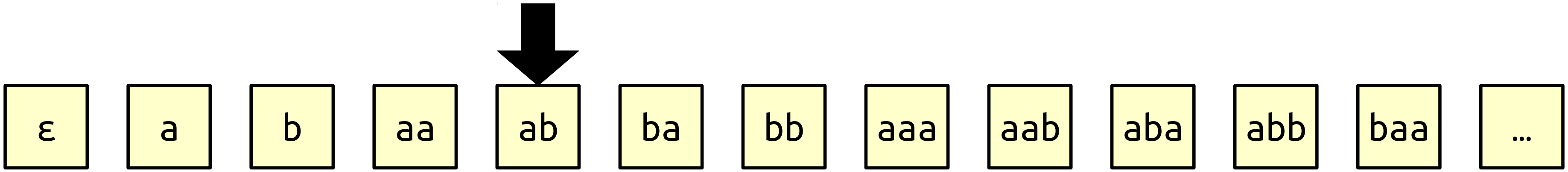


# Verifiers and **RE**

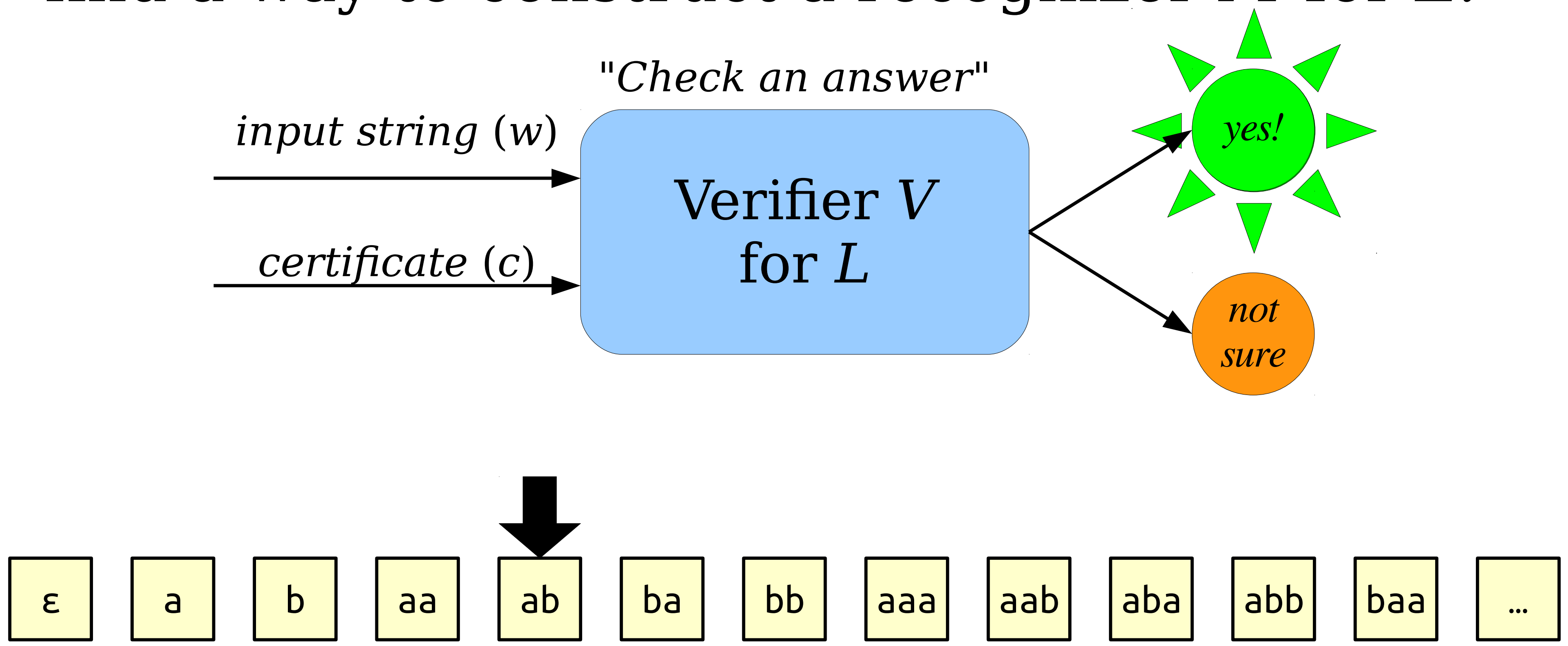
- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in$ **RE**.
- ***Proof goal:*** Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.



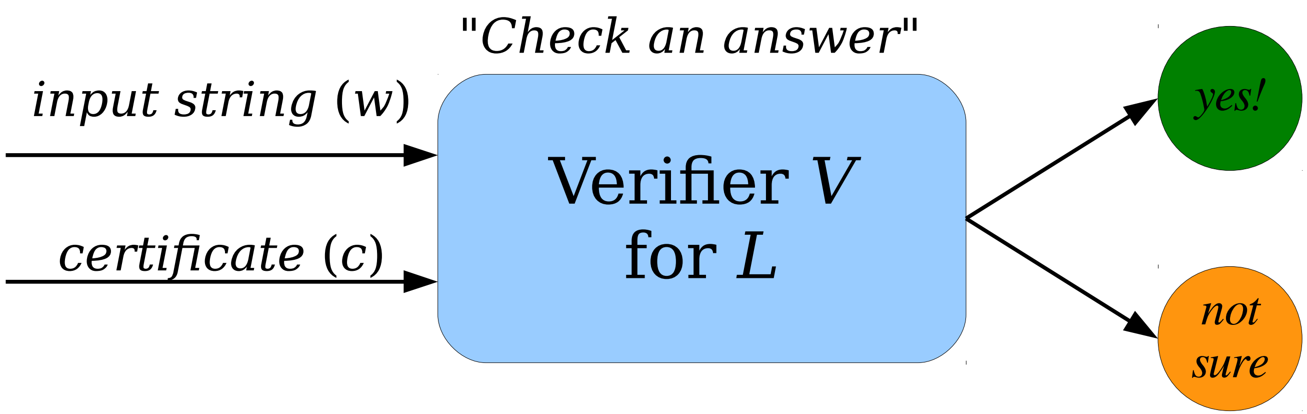
# Verifiers and **RE**

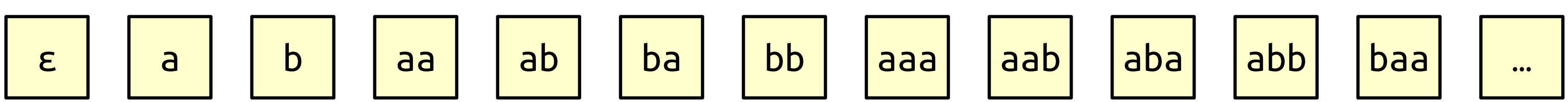
- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in$ **RE**.
- ***Proof goal:*** Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.



# Verifiers and **RE**

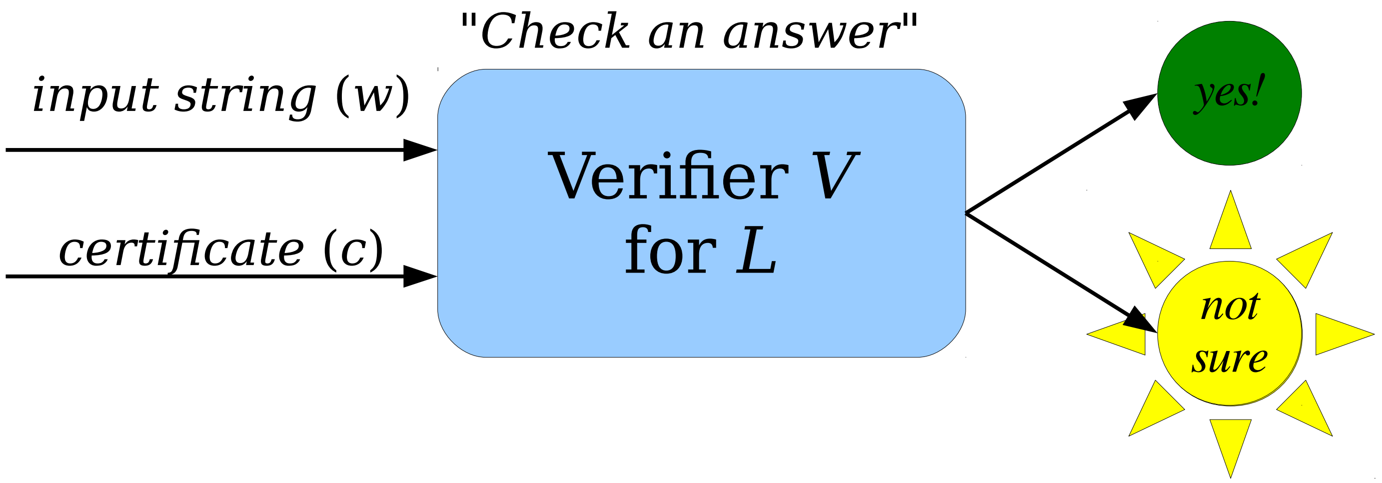
- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in$ **RE**.
- ***Proof goal:*** Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.

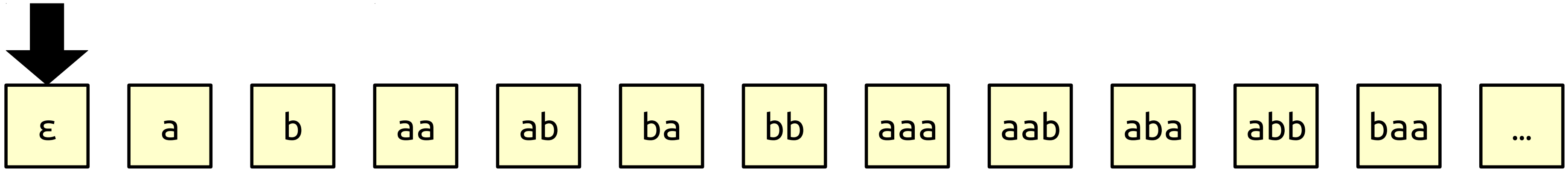


# Verifiers and **RE**

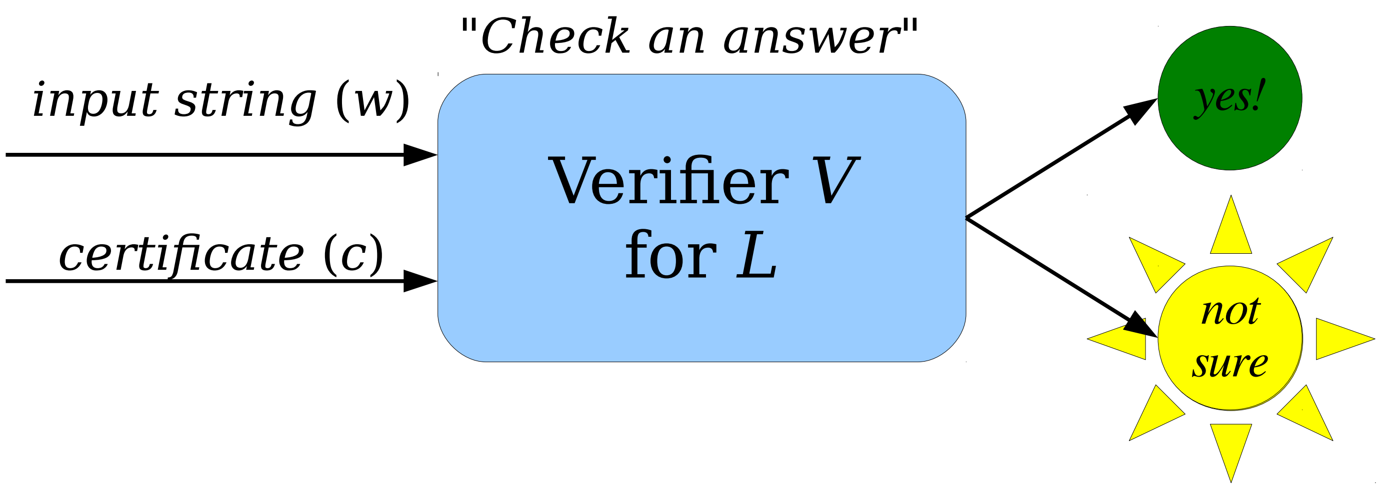
- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in$ **RE**.
- ***Proof goal:*** Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.

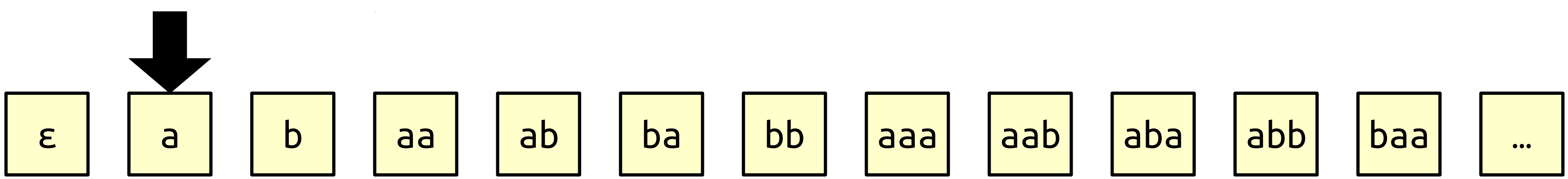


# Verifiers and **RE**

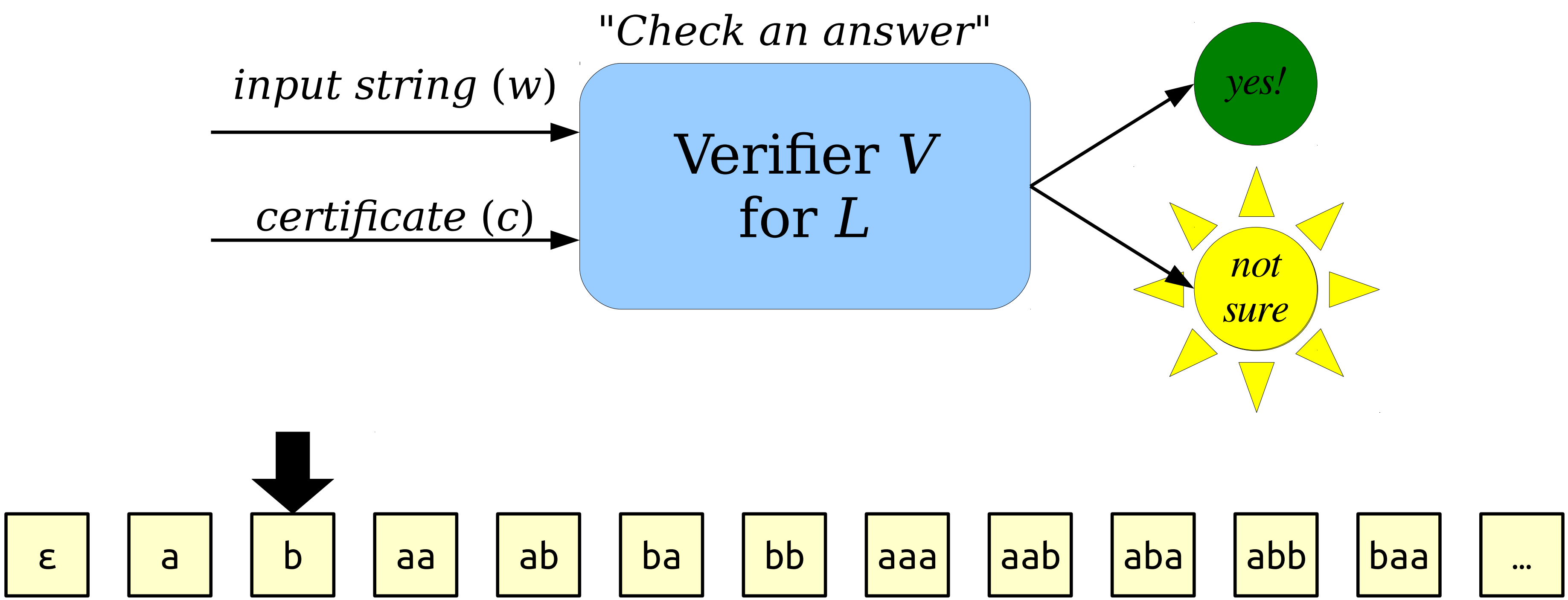
- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in$ **RE**.
- ***Proof goal:*** Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.



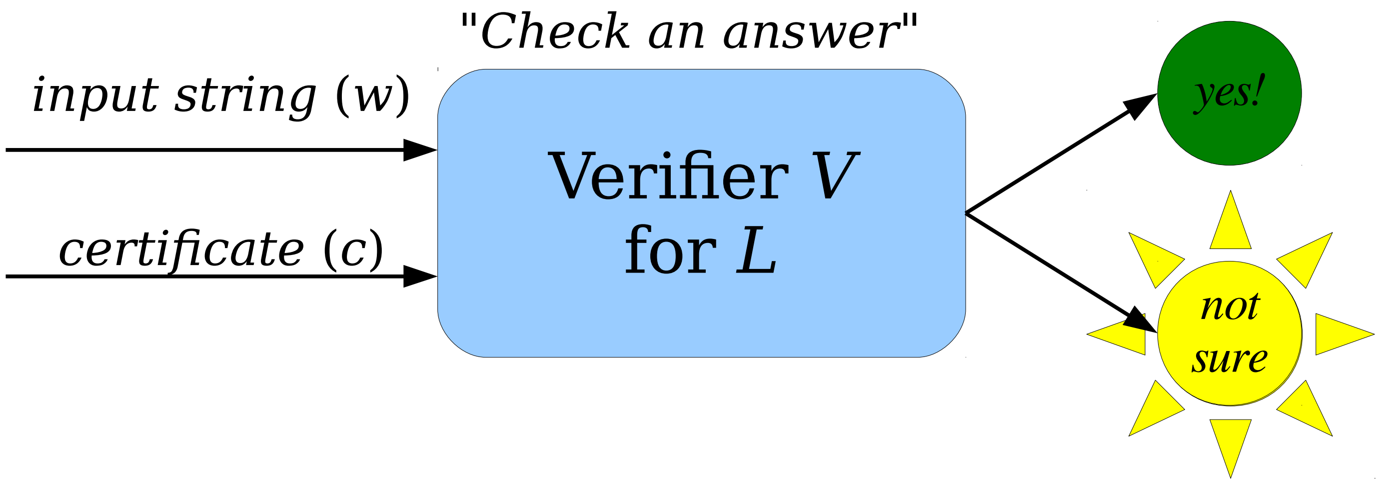
# Verifiers and **RE**

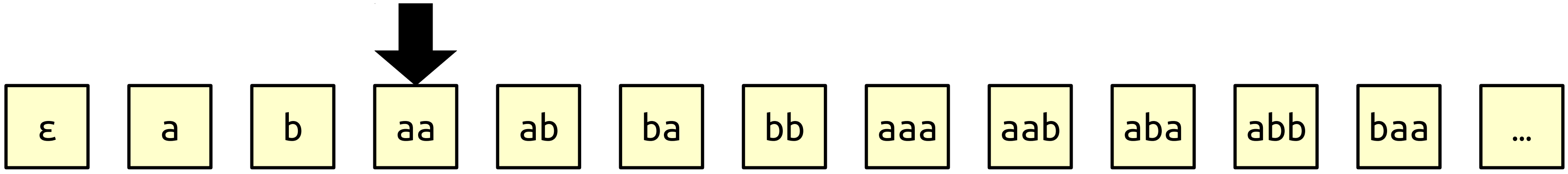
- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in$ **RE**.
- ***Proof goal:*** Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.



# Verifiers and **RE**

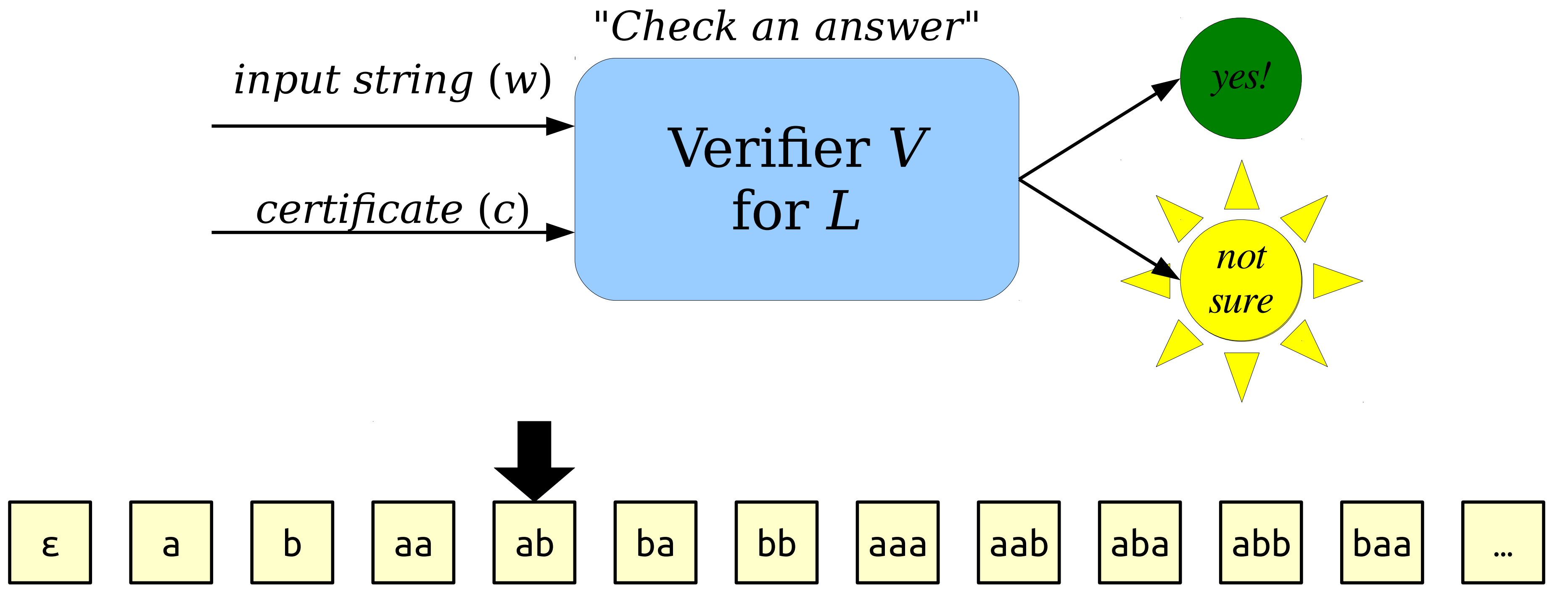
- ***Theorem:*** If there is a verifier *V* for a language *L*, then *L* ∈ **RE**.
- ***Proof goal:*** Given a verifier *V* for a language *L*, find a way to construct a recognizer *M* for *L*.



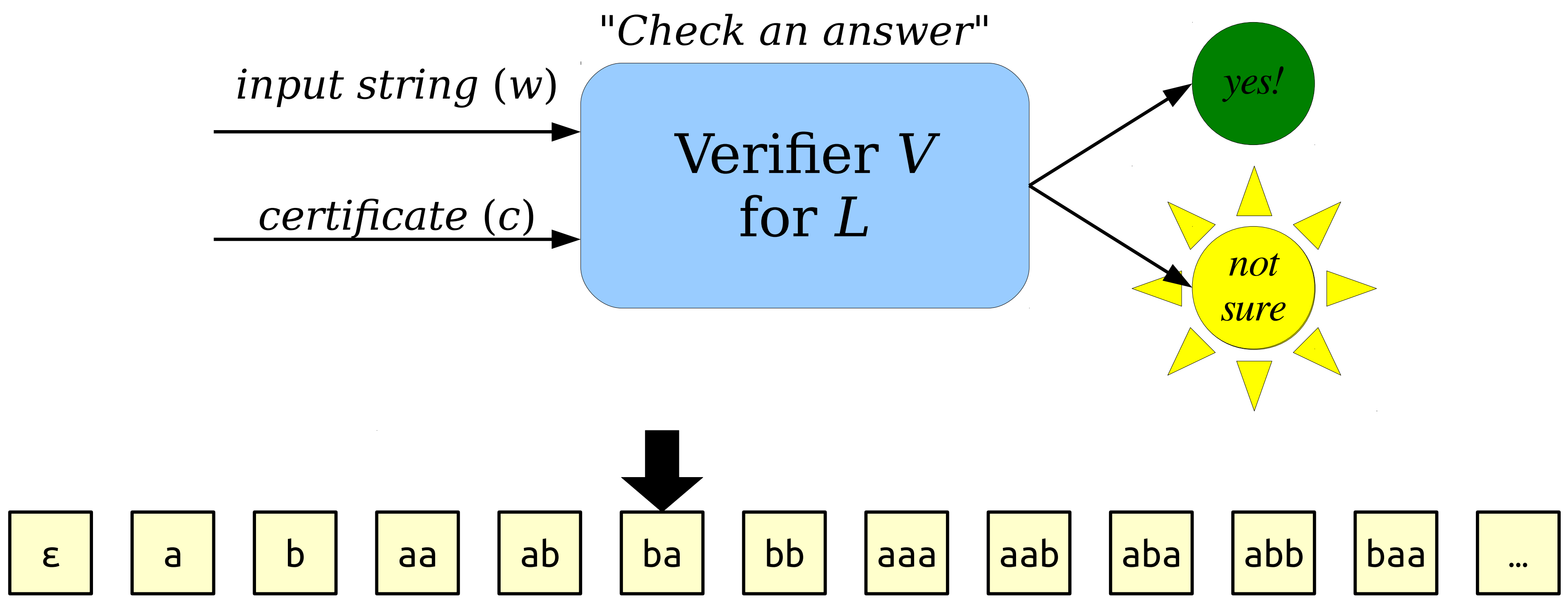
# Verifiers and **RE**

- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in$ **RE**.
- ***Proof goal:*** Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.



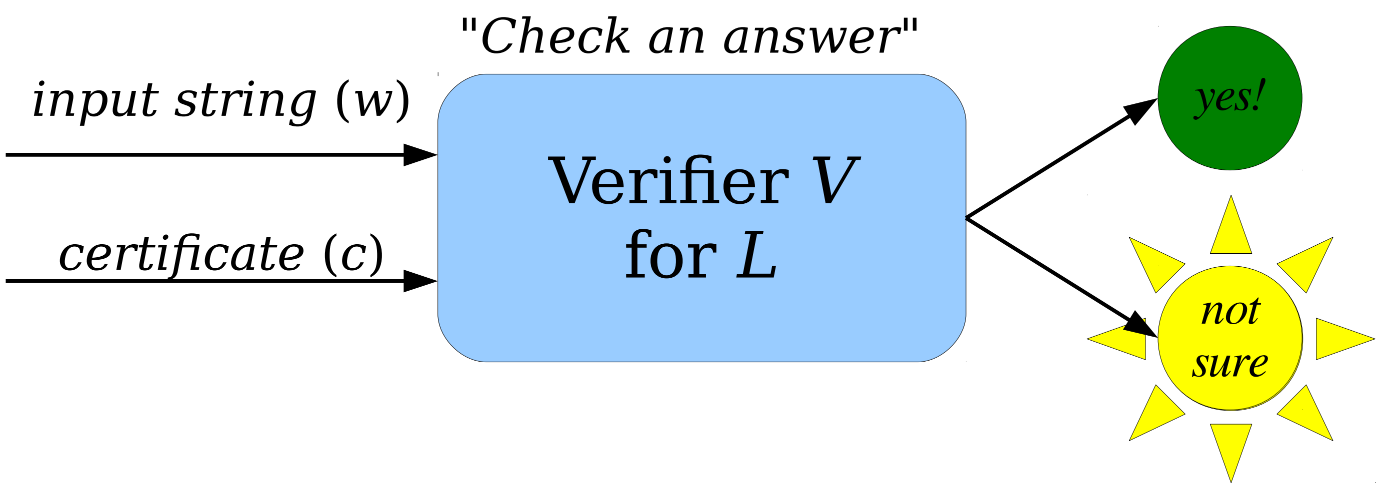
# Verifiers and **RE**

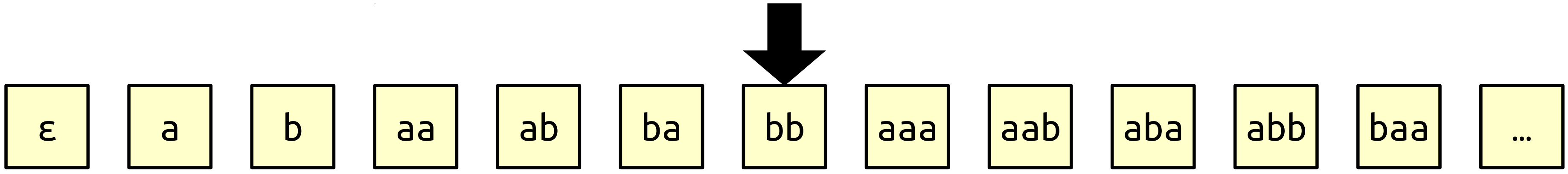
- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in$ **RE**.
- ***Proof goal:*** Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.



# Verifiers and **RE**

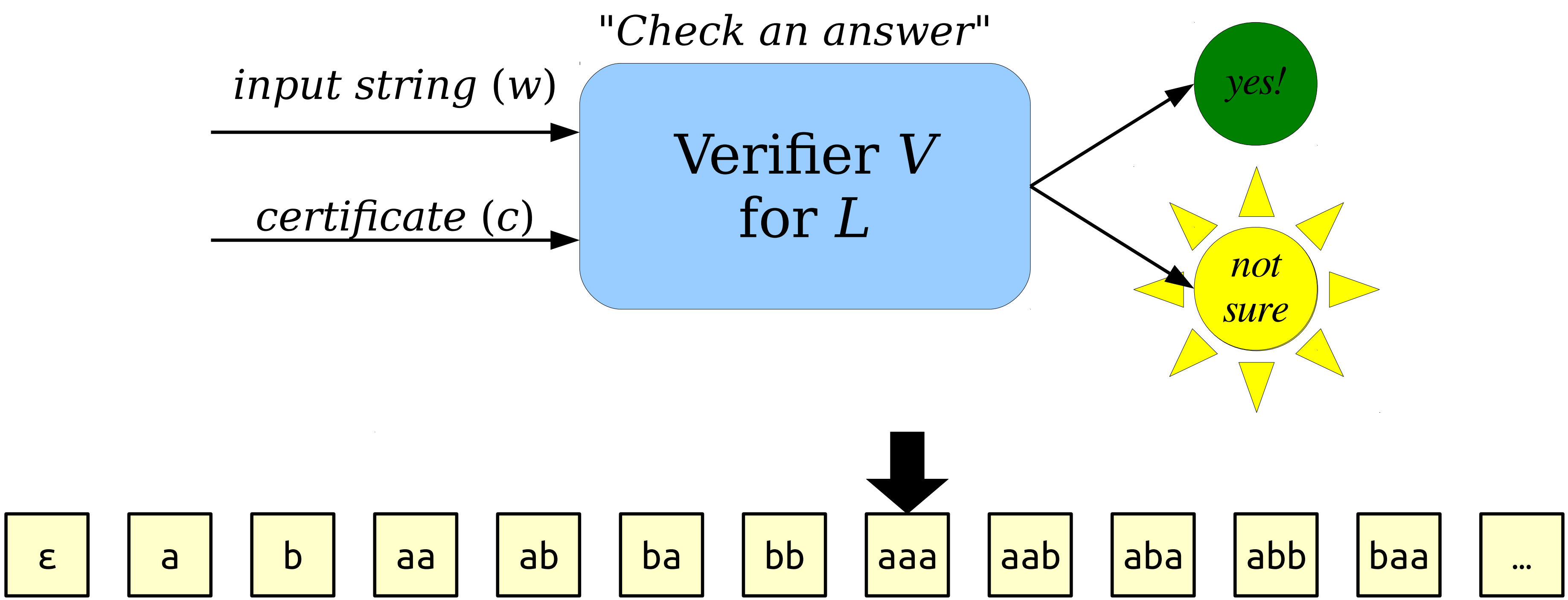
- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in$ **RE**.
- ***Proof goal:*** Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.



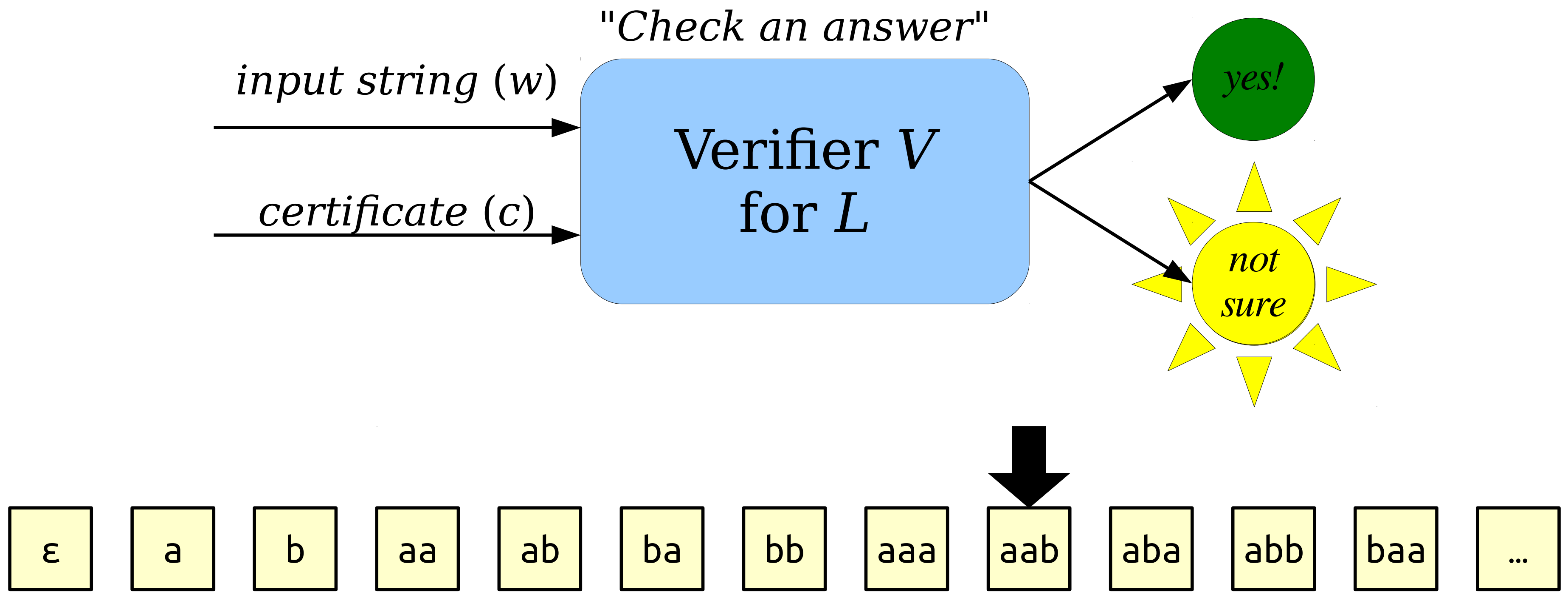
# Verifiers and **RE**

- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in$ **RE**.
- ***Proof goal:*** Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.



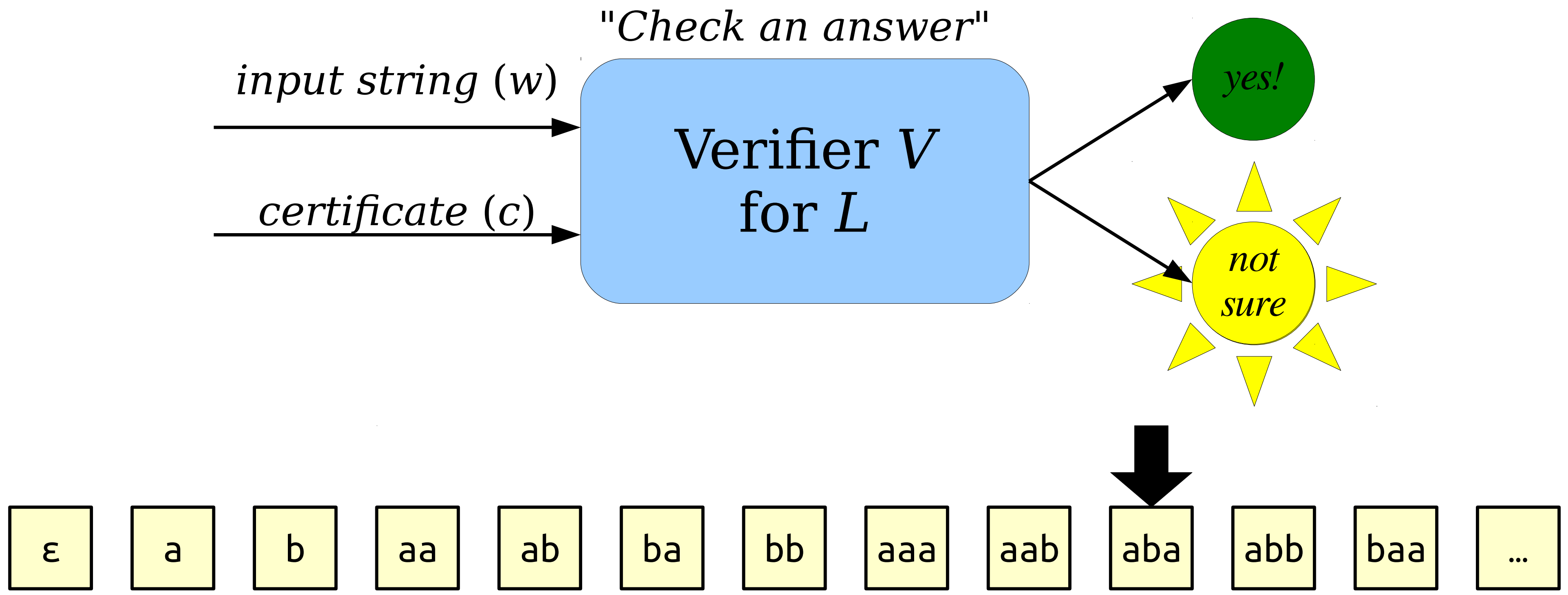
# Verifiers and **RE**

- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in$ **RE**.
- ***Proof goal:*** Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.



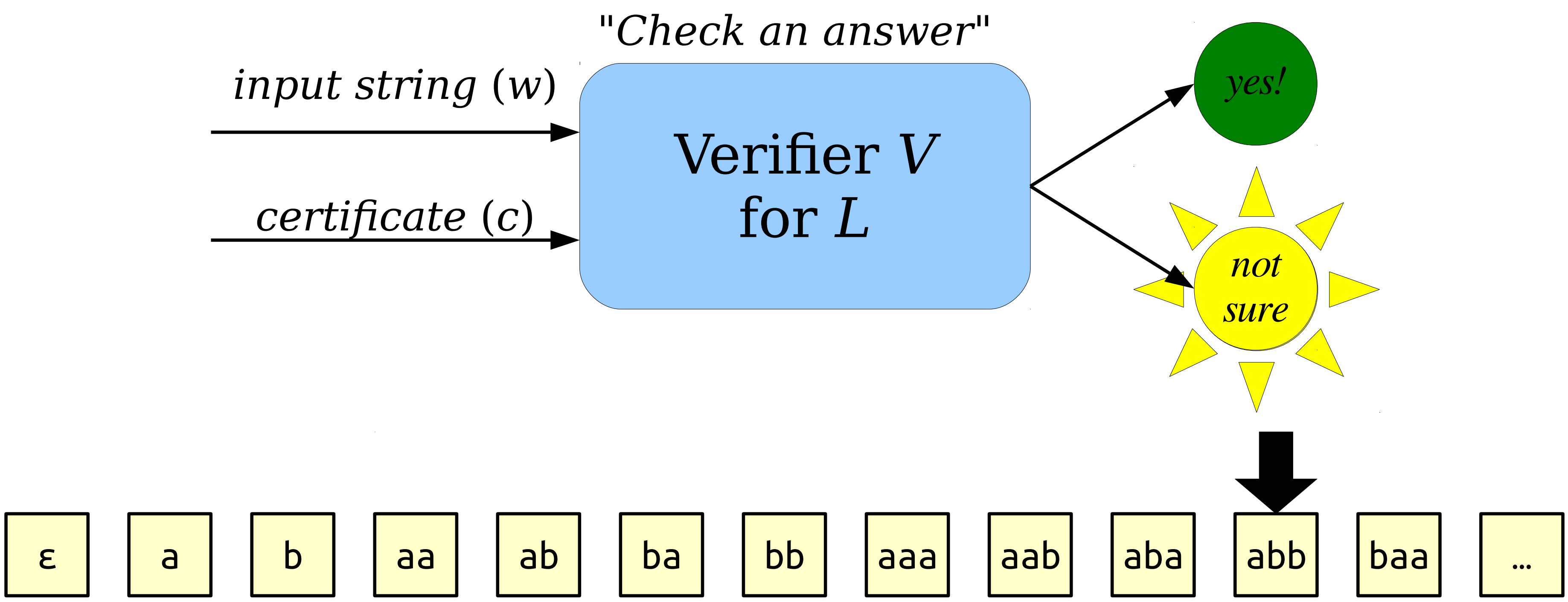
# Verifiers and **RE**

- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in$ **RE**.
- ***Proof goal:*** Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.



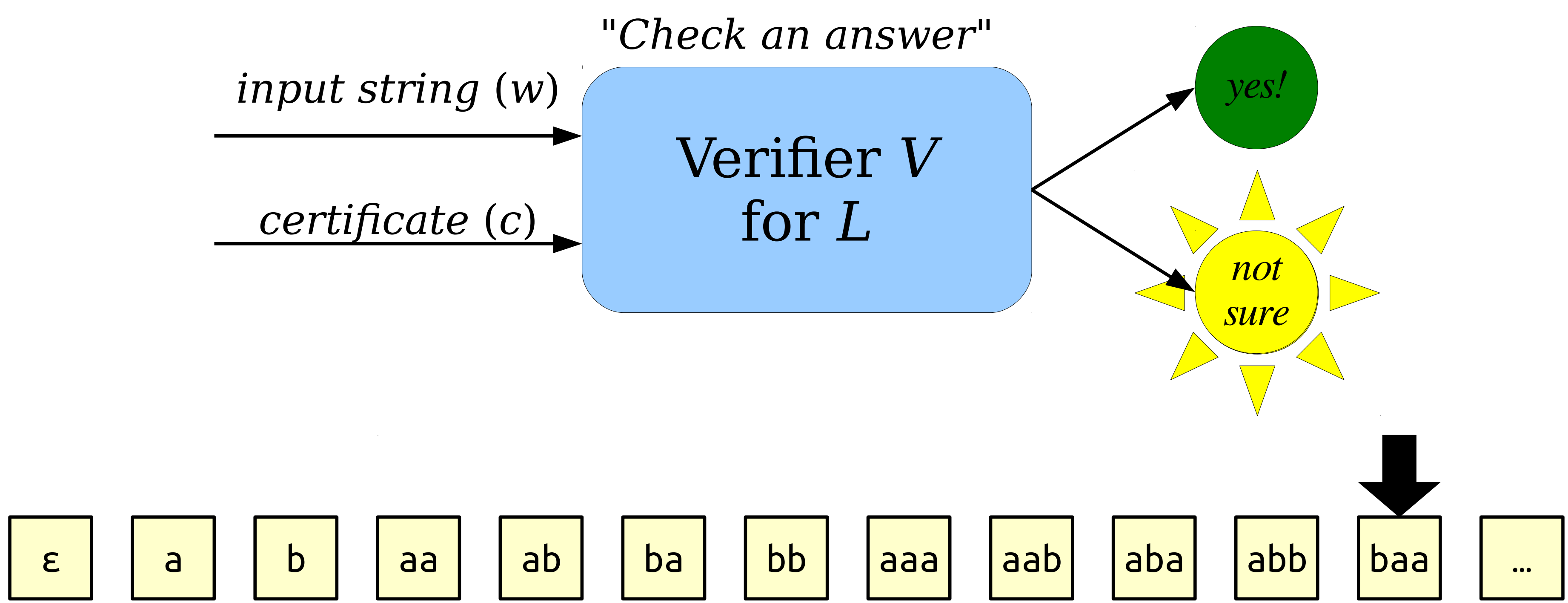
# Verifiers and **RE**

- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in$ **RE**.
- ***Proof goal:*** Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.



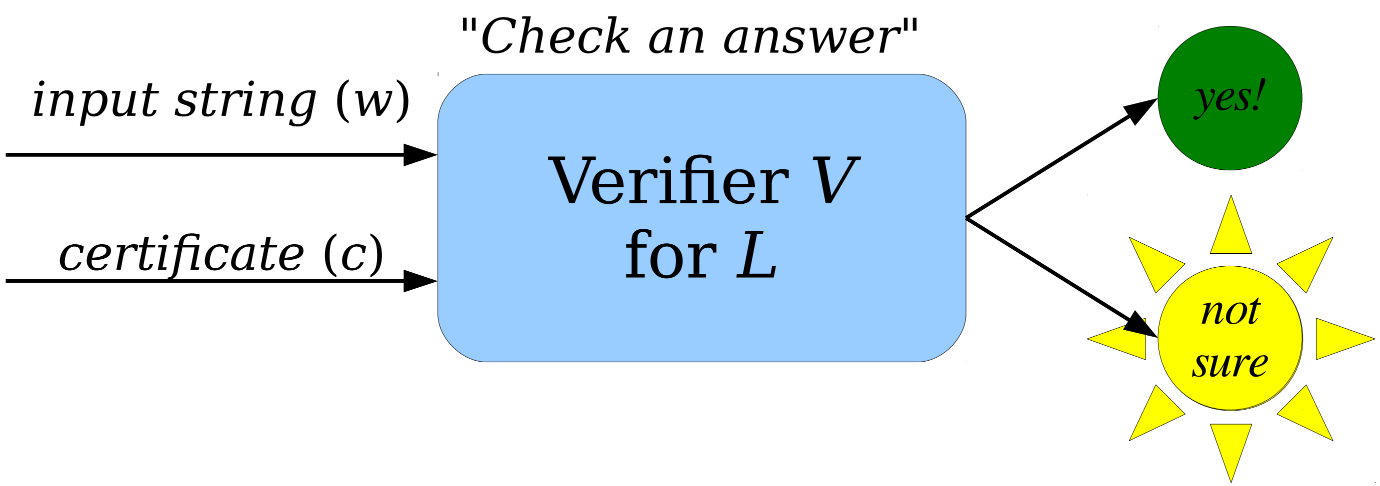
# Verifiers and **RE**

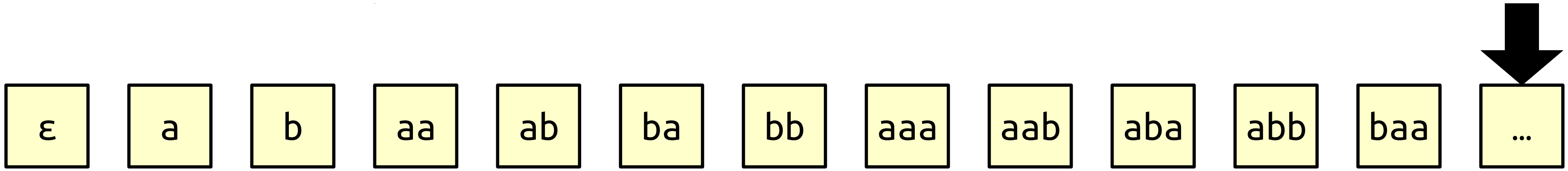
- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in \mathbf{RE}$.
- ***Proof goal:*** Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.



# Verifiers and **RE**

- ***Theorem:*** If there is a verifier $V$ for a language $L$, then $L \in$ **RE**.
- ***Proof goal:*** Given a verifier $V$ for a language $L$, find a way to construct a recognizer $M$ for $L$.



# Verifiers and **RE**

- ***Theorem:*** If $V$ is a verifier for $L$, then $L \in \textbf{RE}$.
- ***Proof sketch:*** Consider the following program:

```
bool isInL(string w) {
    for (each string c) {
        if (V accepts ⟨w, c⟩) return true;
    }
}
```

If $w \in L$, there is some $c \in \Sigma^*$ where $V$ accepts $\langle w, c\rangle$. The function `isInL` tries all possible strings as certificates, so it will eventually find $c$ (or some other working certificate), see $V$ accept $\langle w, c\rangle$, then return true. Conversely, if `isInL(w)` returns true, then there was some string $c$ such that $V$ accepted $\langle w, c\rangle$, so we see that $w \in L$. ■

# Verifiers and **RE**

- ***Theorem:*** If $L \in \mathbf{RE}$, then there is a verifier for $L$.
- ***Proof Goal:*** Beginning with a recognizer $M$ for the language $L$, show how to construct a verifier $V$ for $L$.

# Verifiers and **RE**

- ***Theorem:*** If $L \in$ **RE**, then there is a verifier for $L$.
- ***Proof sketch:*** Let $L$ be a **RE** language and let $M$ be a recognizer for it. Consider this function:

```
bool checkIsInL(string w, int c) {
    TM M = /* hardcoded version of a recognizer for L */;
    set up a simulation of M running on w;
    for (int i = 0; i < c; i++) {
        simulate the next step of M running on W;
    }
    return whether M is in an accepting state;
}
```

Note that `checkIsInL` always halts, since each step takes only finite time to complete. Next, notice that if there is a $c$ where `checkIsInL(w, c)` returns true, then $M$ accepted $w$ after running for $c$ steps, so $w \in L$. Conversely, if $w \in L$, then $M$ accepts $w$ after some number of steps (call that number $c$). Then `checkIsInL(w, c)` will run $M$ on $w$ for $c$ steps, watch $M$ accept $w$, then return true. ■