Unsolvable Problems

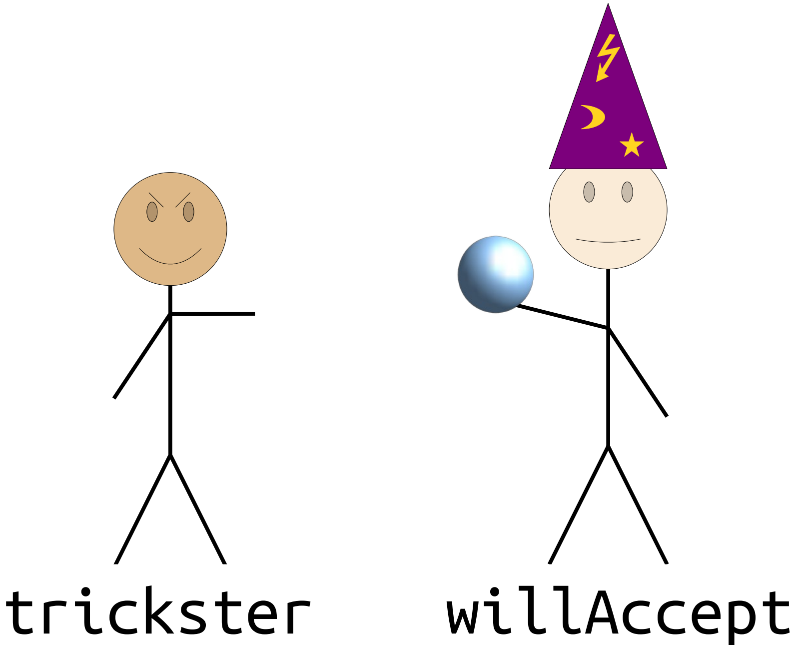
Part Two

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);
}
```



`trickster(input)` returns true

↔

`willAccept(me, input)` returns true

↔

`trickster(input)` returns false

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.

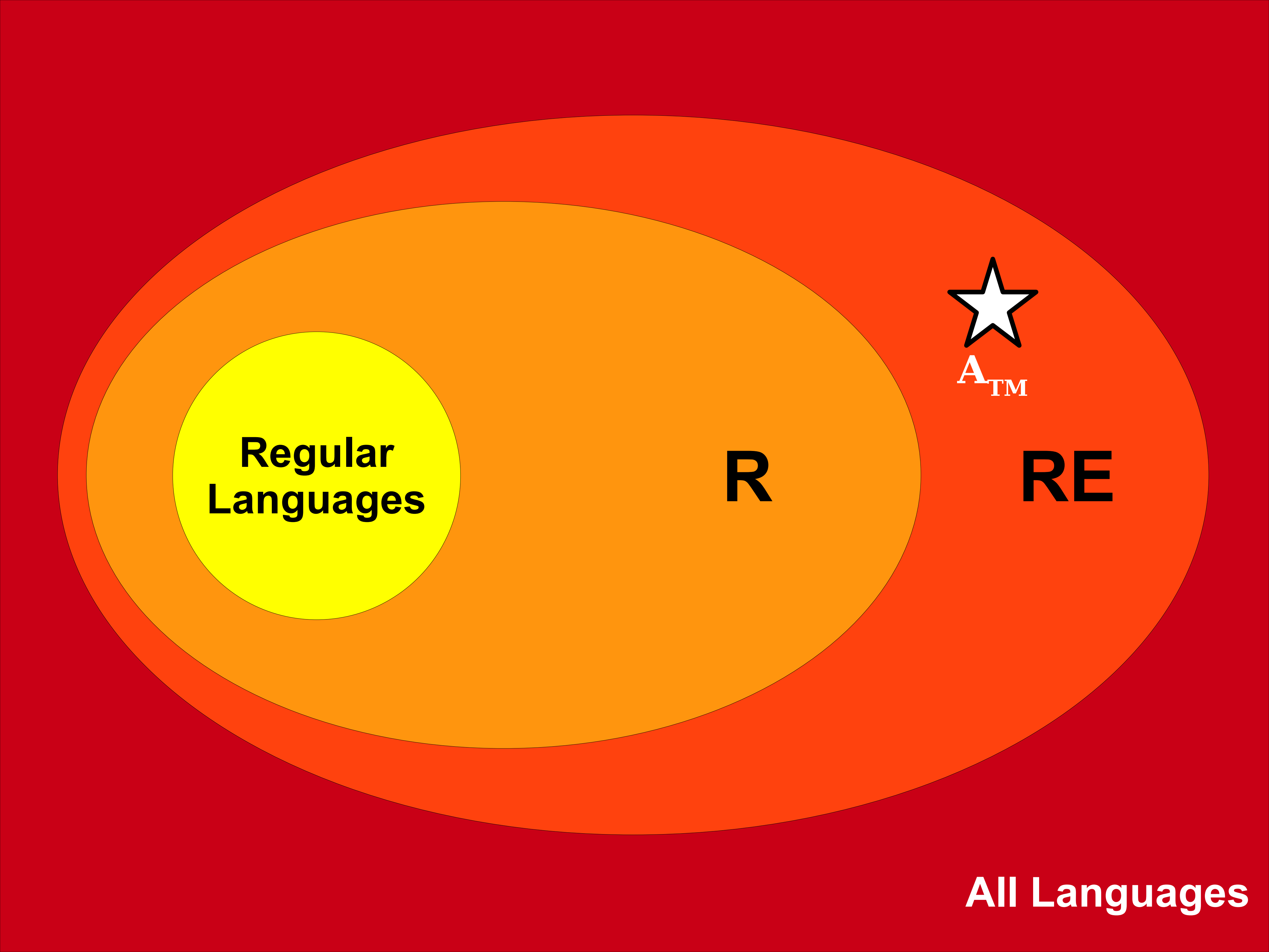
Given how `trickster` is written, we see that

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

This means that

`trickster(input)` returns true if and only if `trickster(input)` returns false.

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



Regular Languages

R

A_{TM}

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*?

- As a formal language, this problem would be expressed as

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

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

HALT ∈ **RE**

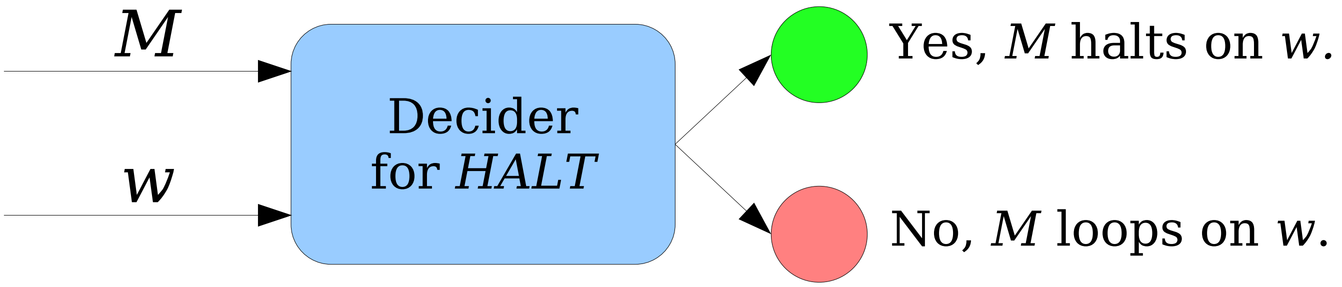
- ***Claim:*** *HALT* ∈ **RE**.
- ***Idea:*** If you were certain that a TM *M* halted on a string *w*, could you convince me of that?
- Yes – just run *M* on *w* and see what happens!

```
bool willHalt(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;
   }
}
```

Theorem: The halting problem is undecidable.

A Decider for *HALT*

- Let’s suppose that, somehow, we managed to build a decider for *HALT* = { ⟨*M*, *w*⟩ | *M* is a TM that halts on *w* }.
- Schematically, that decider would look like this:



M

w

Decider for *HALT*

Yes, *M* halts on *w*.

No, *M* loops on *w*.

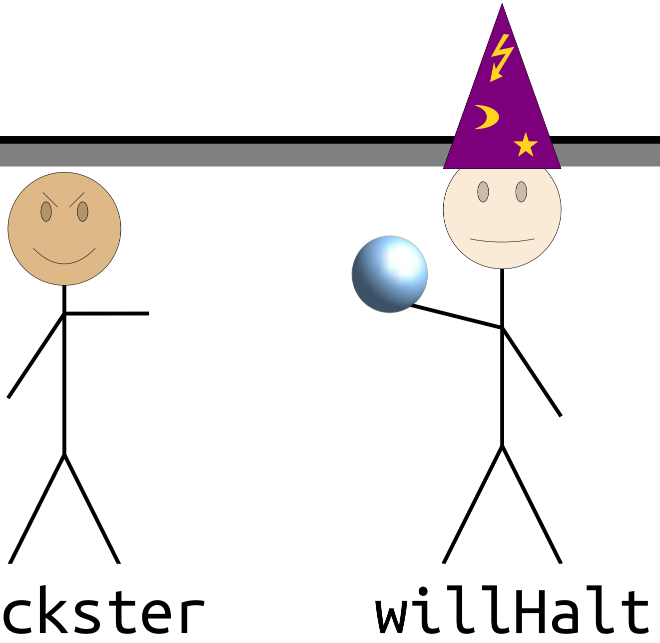
- We could represent this decider in software as a method

```
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) {









}
```

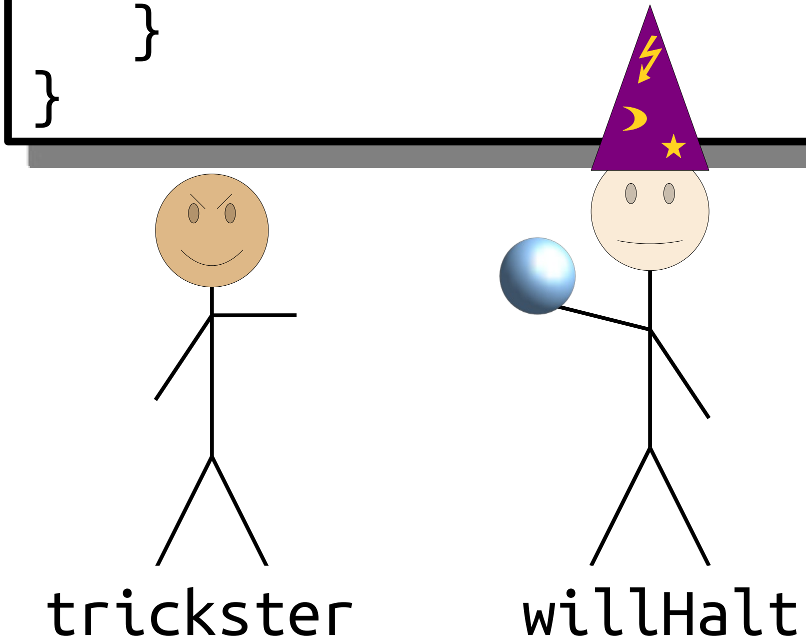


`trickster(input)` halts

↔

`willHalt(me, input)` returns true

```
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;
   }
}
```



`trickster(input)` halts

↔

`willHalt(me, input)` returns true

↔

`trickster(input)` loops

Theorem: *HALT* ∉ **R**.

Proof: By contradiction; assume that *HALT* ∈ **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.

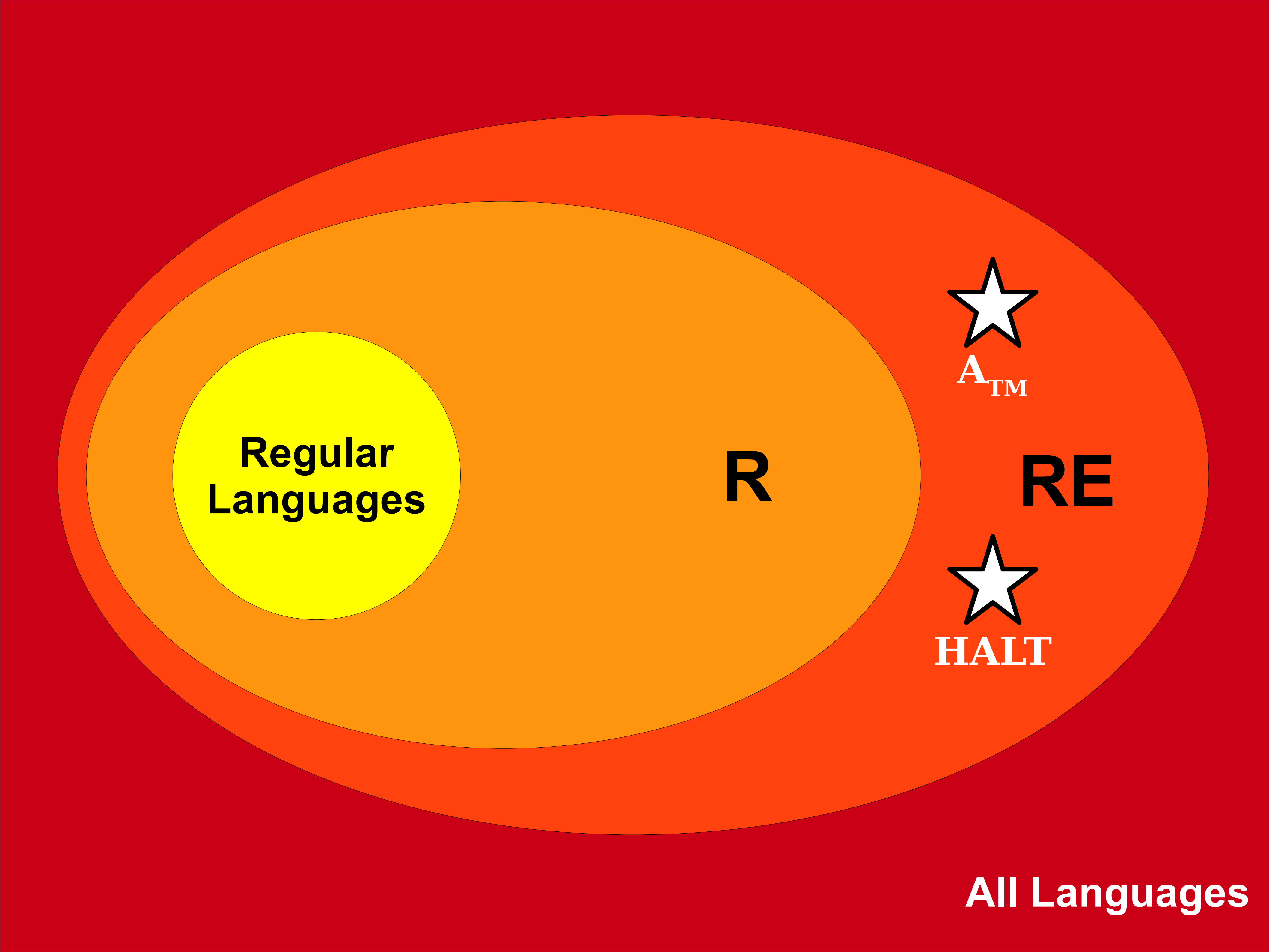
Given how `trickster` is written, we see that

`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

R

A_{TM}

RE

HALT

All Languages

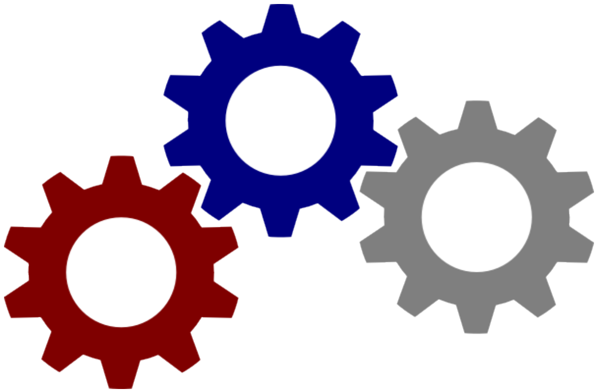
So What?

- 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!

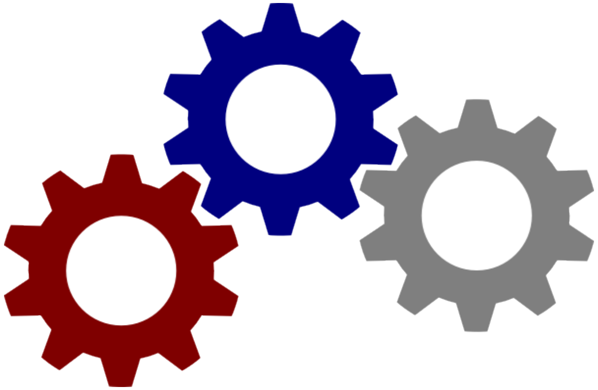
Engineering Problem: Design a diesel engine that doesn’t emit lots of NO_x pollutants.

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

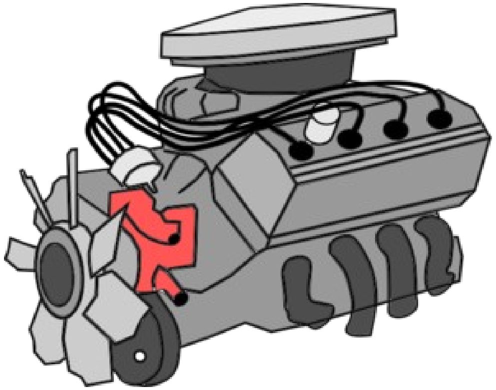


Engineering Prowess!

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

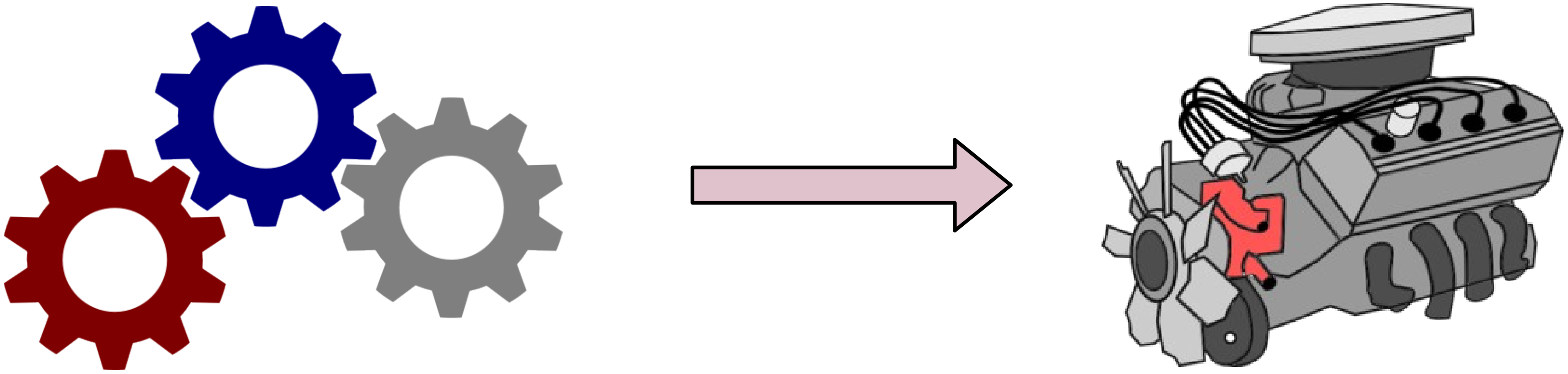


Engineering Prowess!



Awesome Engine!

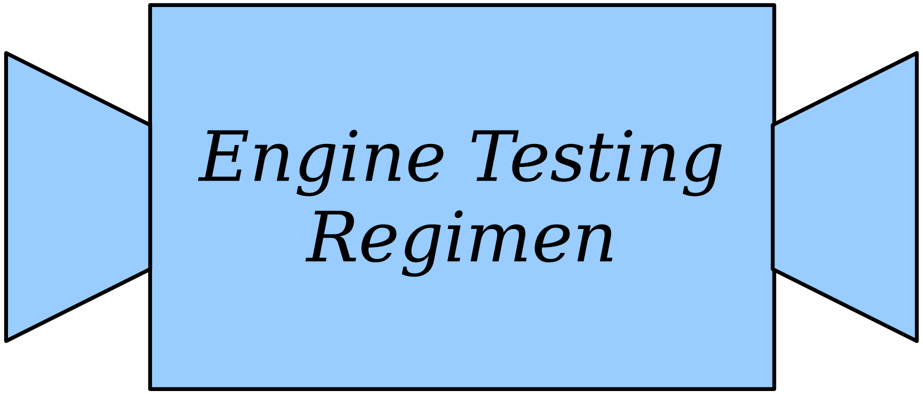
Engineering Problem: Design a diesel engine that doesn't emit lots of NO_x pollutants.



Engineering Prowess!

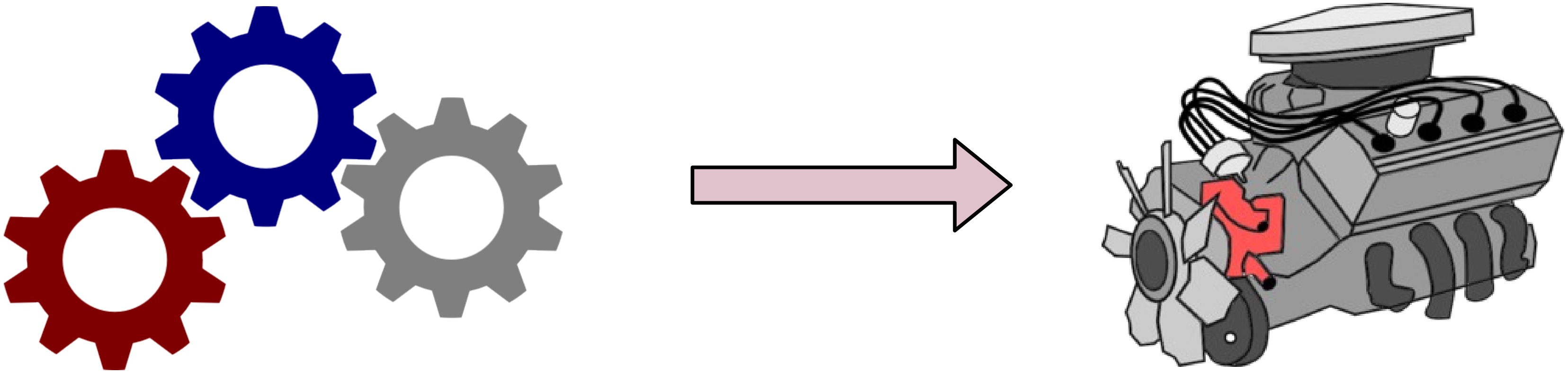
Awesome Engine!

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



Engine Testing Regimen

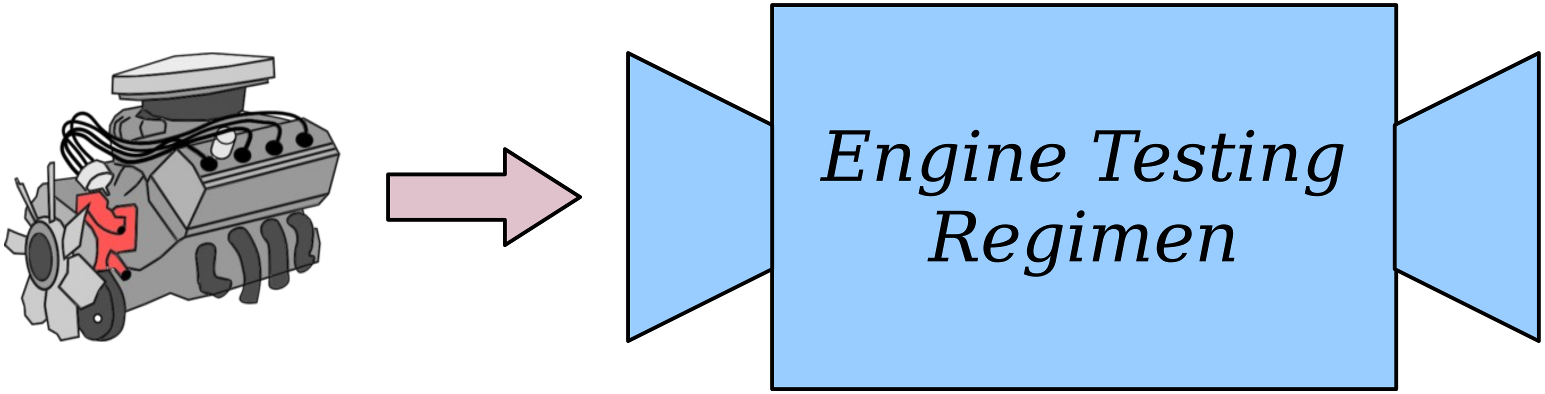
Engineering Problem: Design a diesel engine that doesn't emit lots of NO_x pollutants.



Engineering Prowess!

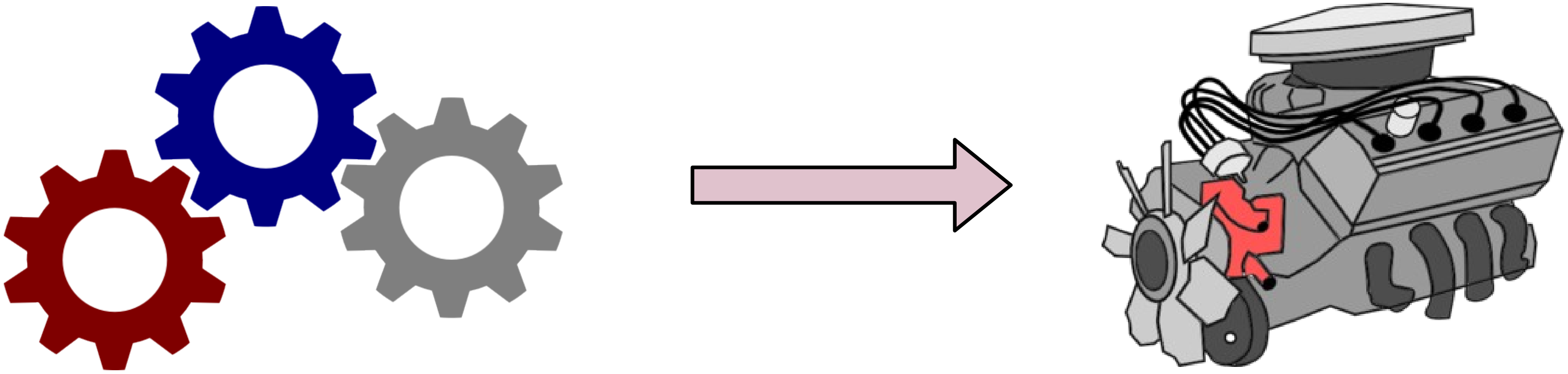
Awesome Engine!

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



Engine Testing Regimen

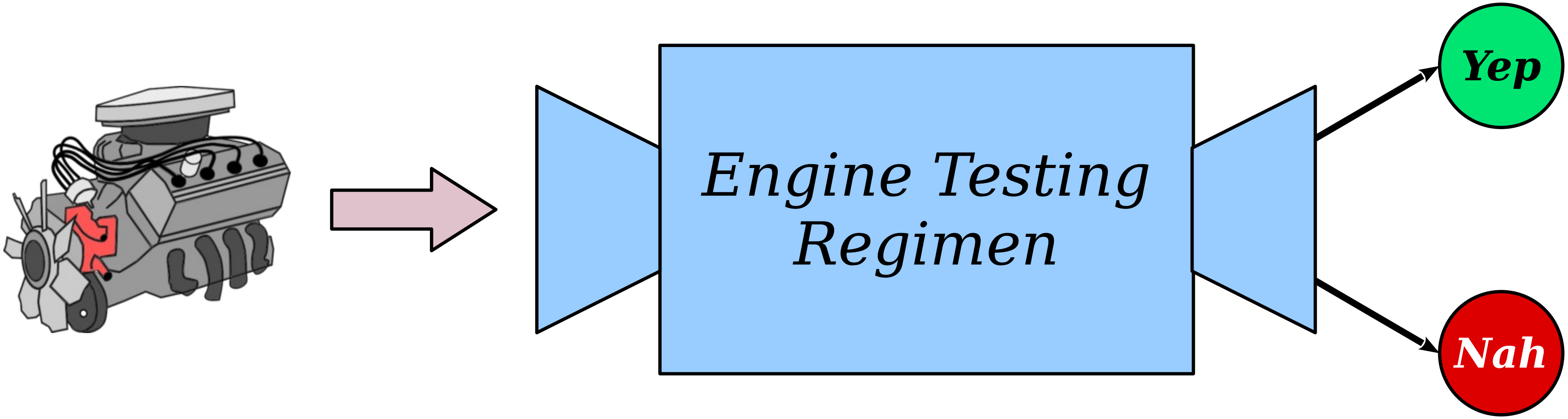
Engineering Problem: Design a diesel engine that doesn't emit lots of NO_x pollutants.



Engineering Prowess!

Awesome Engine!

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



Engine Testing Regimen

Yep

Nah

Fact: Almost all “regulatory problems” about computer programs are undecidable. That is, almost all problems of the form “does this program have [`behavioral property` *X*]” are undecidable.

This can be formalized through a result called ***Rice’s Theorem***; take CS154 for details!

Secure Voting

- Suppose that you want to make a voting machine for use in an election between two parties.
- Let $\Sigma = \{$**`r`**, **`d`**$\}$. A string $w \in \Sigma^*$ corresponds to a series of votes for the candidates.
- Example: **`rrdddrd`** means "two people voted for **`r`**, then three people voted for **`d`**, then one more person voted for **`r`**, then one more person voted for **`d`**."

Secure Voting

- A voting machine is a program that takes as input a string of `r`'s and `d`'s, then reports whether person `r` won the election.
- ***Question:*** Given a TM that someone claims is a secure voting machine, could we automatically check whether it actually is a secure voting machine?

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

```
bool bee(string input) {
   int numRs = countRsIn(input);
   int numDs = countDsIn(input);

   return numRs > numDs;
}
```

A (simple) secure voting machine.

```
bool topaz(string input) {
   return input[0] == 'r';
}
```

A (simple) insecure voting machine.

```
bool anna(string input) {
   int numRs = countRsIn(input);
   int numDs = countDsIn(input);

   if (numRs = numDs) {
      return false;
   } else if (numRs < numDs) {
      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 numRs = countRsIn(input);
   int numDs = countDsIn(input);

   return numRs > numDs;
}
```

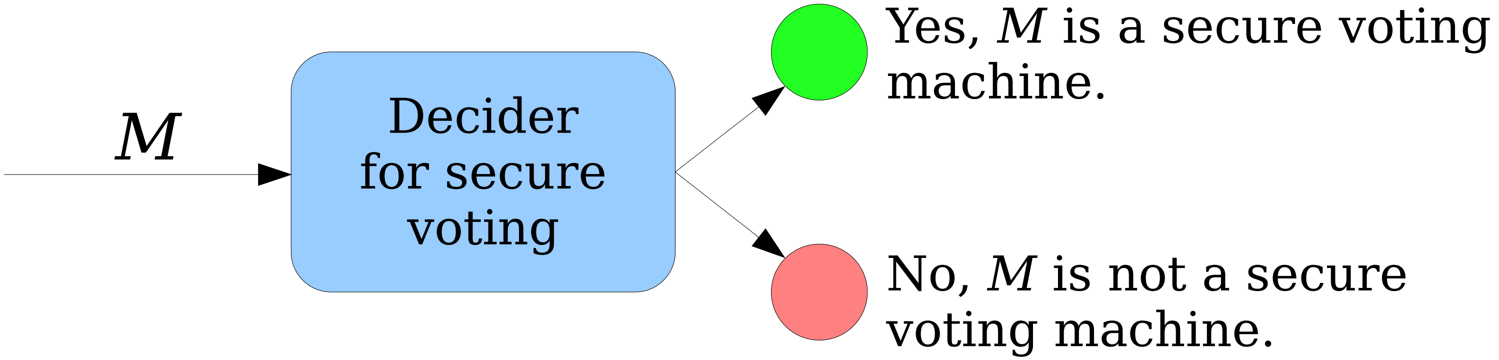
No one knows!

Secure Voting

- A voting machine is a program that takes as input a string of **`r`**'s and **`d`**'s, then reports whether person **`r`** won the election.
- ***Question:*** Given a TM that someone claims is a secure voting machine, could we automatically check whether it actually is a secure voting machine?

A Decider for Secure Voting

- Let's suppose that, somehow, we managed to build a decider for the secure voting problem.
- Schematically, that decider would look like this:

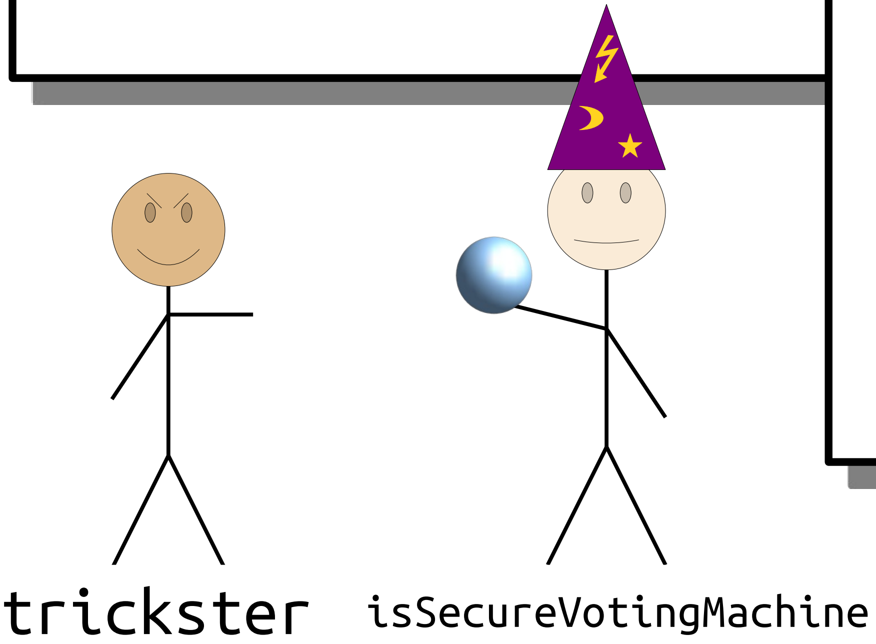


- We could represent this decider in software as a method

```
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 r’s than d’s.
}

bool trickster(string input) {






}
```

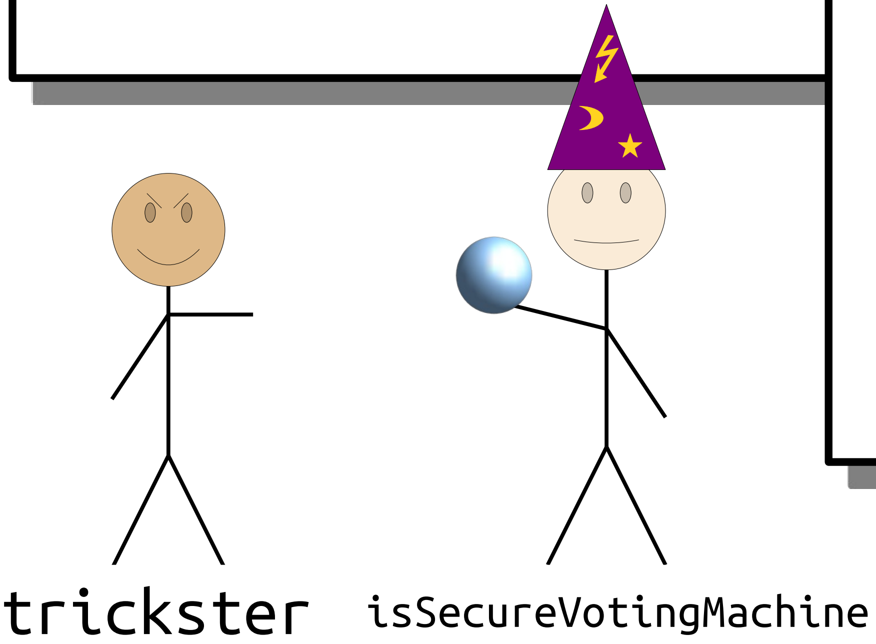


`trickster` is a secure voting machine

↔

`isSecureVotingMachine(me)` returns true

```
bool isSecureVotingMachine(string function) {
   // Returns whether function accepts only
   // strings with more r’s than d’s.
}

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

   if (isSecureVotingMachine(me)) {
       return countRsIn(input) <= countDsIn(input);
   } else {
       return countRsIn(input) > countDsIn(input);
   }
}
```



`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 **`r`**'s than **`d`**'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 r's as d's */;
    } else {
        return /* if input has more r's than d'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 general algorithm* that we can follow to determine whether a program is a secure voting machine. In other words, any general algorithm to check voting machines will always be wrong on at least one input.
- So what can we do?
  - Design algorithms that work in *some*, but not *all* cases. (This is often done in practice.)
  - Fall back on human verification of voting machines. (We do that too.)
  - Carry a healthy degree of skepticism about electronic voting machines. (Then again, did we even need the theoretical result for this?)

Time-Out for Announcements!

Please evaluate this course in Axess.

Your comments really make a difference.

Problem Sets

- PS6 solution released. We are aiming to finish grading by Wednesday noon.
- Problem Set Seven is due on Wednesday at 2:30PM.

Final Exam

- The final exam will be this Saturday from 7-10PM PST at Hewlett 201 (this lecture hall)
- The exam is open-book, open-note, open-internet, and closed-other-humans.
  - Turing Machines will **not** be covered on this exam.
  - Expect the other topics to scale accordingly.
- ***You can do this.*** Best of luck on the exam!

Back to CS103!

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

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

- We've now seen how to use self-reference as a tool for showing undecidability (finding languages not in **R**).
- We still have not broken out of **RE** yet, though.
- To do so, we will need to build up a better intuition for the class **RE**.

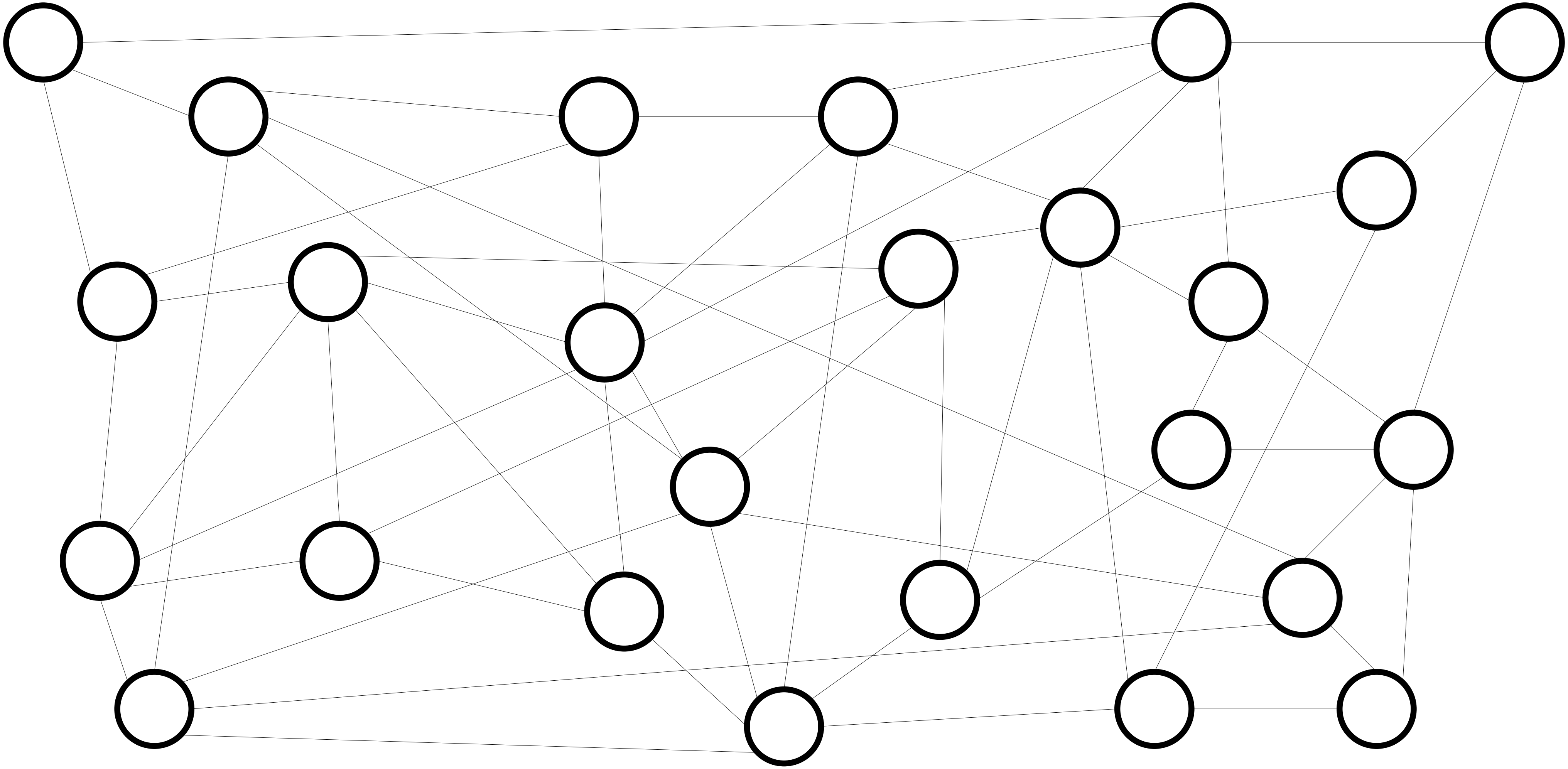
What exactly is the class **RE**?

RE, Formally

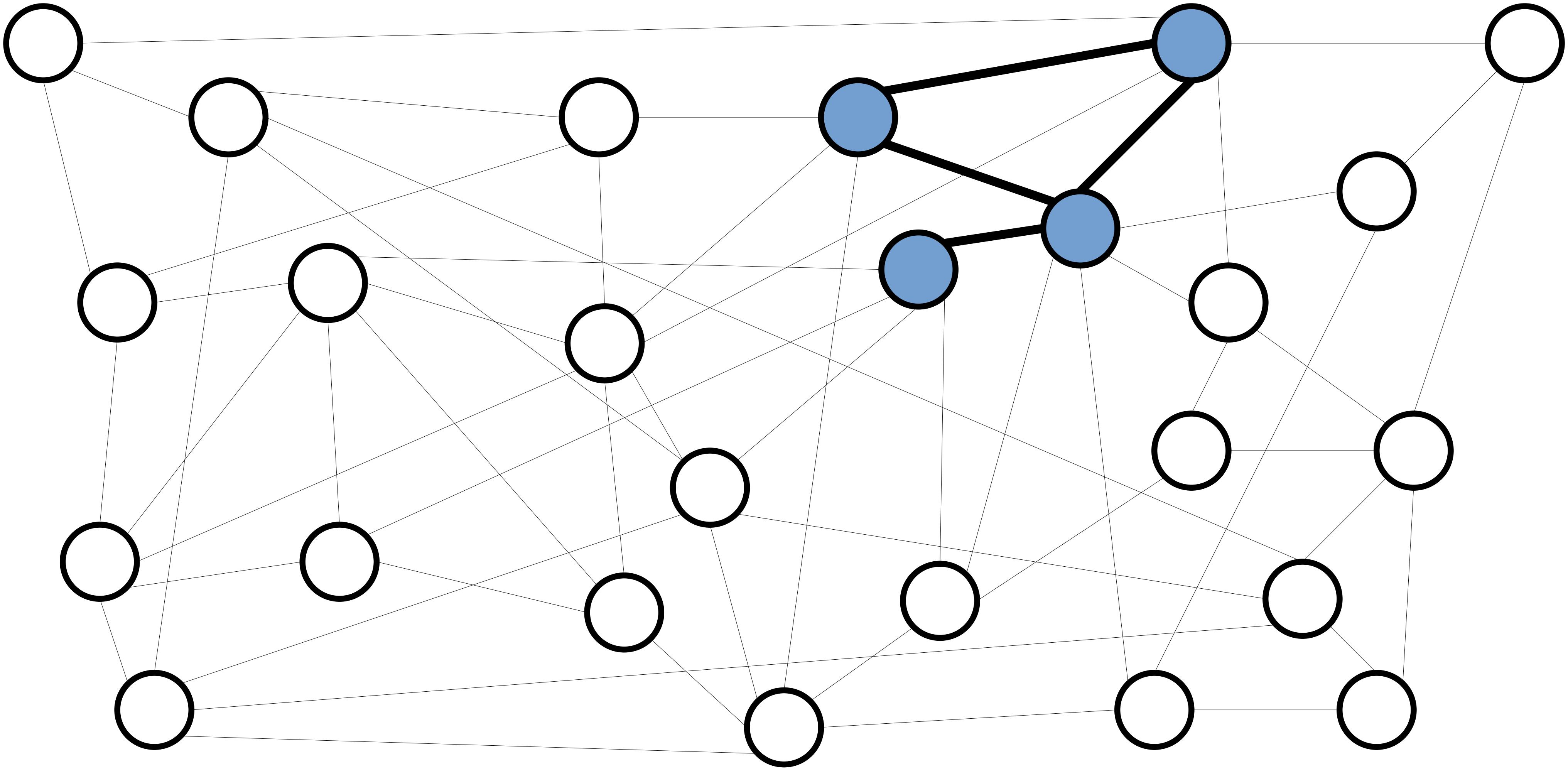
- Recall that the class **RE** is the class of all recognizable languages:

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

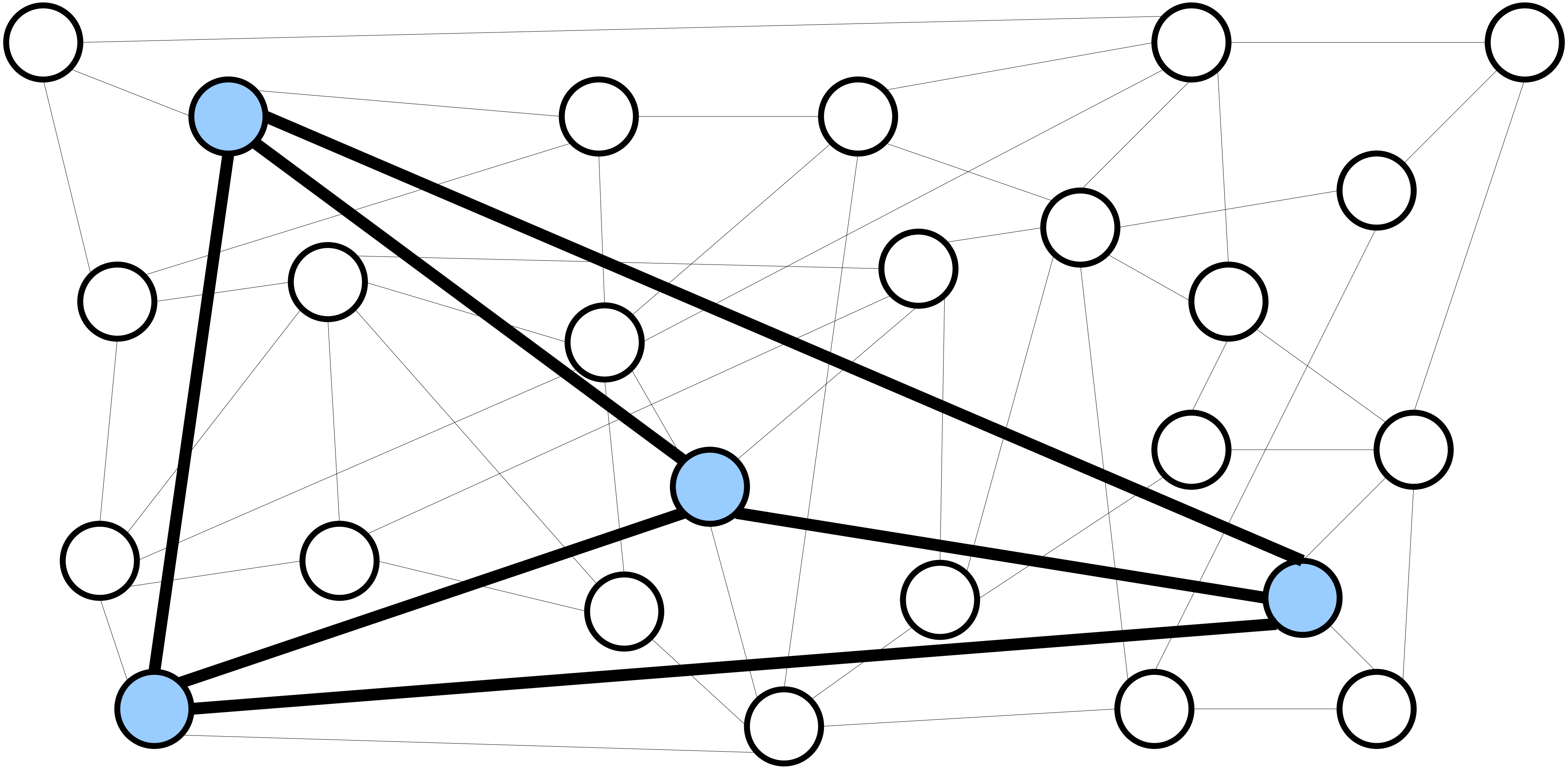
- Since **R** ≠ **RE**, there is no general way to "solve" problems in the class **RE**, if by "solve" you mean "make a computer program that can always tell you the correct answer."
- So what exactly *are* the sorts of languages in **RE**?



Does this graph contain four mutually adjacent nodes?



Does this graph contain four mutually adjacent nodes?



Does this graph contain four mutually adjacent nodes?

Key Intuition:

A language L is in **RE** if, for any string w, if you are *convinced* that $w \in L$, there is some way you could prove that to someone else.

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

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?

Verifiers

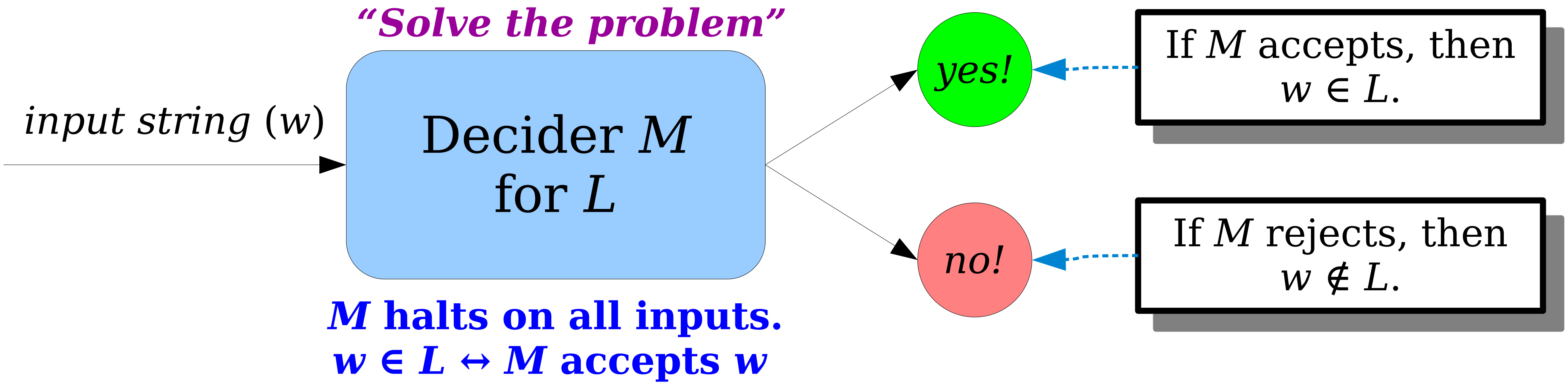
- A ***verifier*** for a language *L* is a TM *V* with the following two properties:

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



"Solve the problem"

input string (w)

Decider M for L

yes!

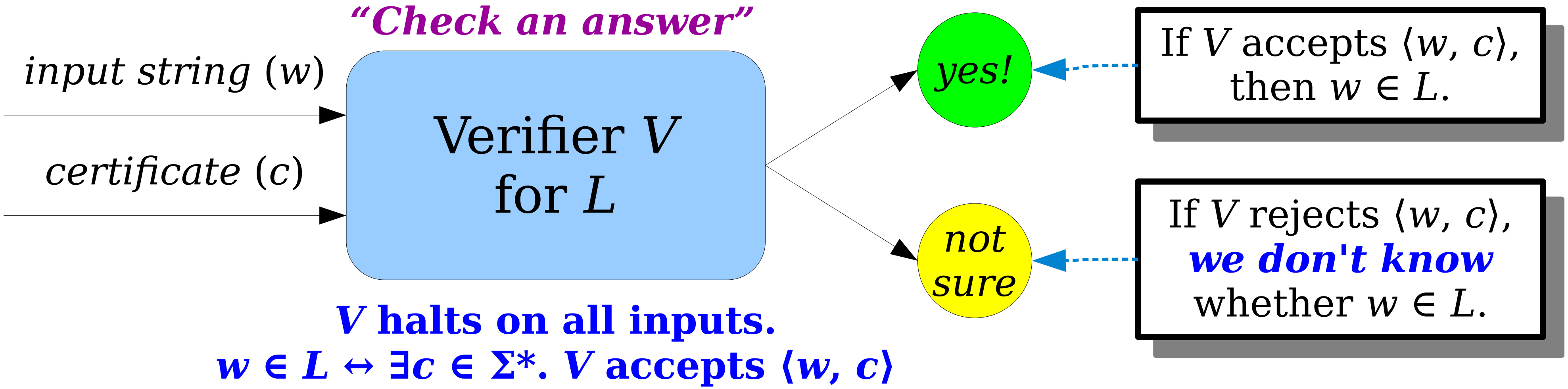
no!

If M accepts, then $w \in L$.

If M rejects, then $w \notin L$.

M halts on all inputs.

$w \in L \leftrightarrow M$ accepts w



"Check an answer"

input string (w)

certificate (c)

Verifier V for L

yes!

not sure

If V accepts $\langle w, c \rangle$, then $w \in L$.

If V rejects $\langle w, c \rangle$, ***we don't know*** whether $w \in L$.

V halts on all inputs.

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

Verifiers

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

V halts on all inputs.

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

- 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:
  - Notice that 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:
  - Notice that V isn't a decider for L and isn't a recognizer for L.
  - The job of V is just to check certificates, not to 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:
  - Although this formal definition works with a string c, remember that c can be an encoding of some other object.
  - In practice, c will likely just be “some other auxiliary data that helps you out.”

A Very Nifty Verifier

- Consider A_{TM}:

  $A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$.

- This is a ***canonical*** example of an undecidable language. There's no way, in general, to tell whether a TM *M* will accept a string *w*.
- Although this language is undecidable, it's an **RE** language, and it's possible to build a verifier for it!

A Very Nifty Verifier

- Consider A_{TM}:

  $A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}.$

- We know that U_{TM} is a recognizer for A_{TM}. It is also a *verifier* for A_{TM}?
- No, for two reasons:
  - U_{TM} doesn't always halt. *(Do you see why?)*
  - U_{TM} takes as input a TM M and a string w. A verifier also needs a certificate.

A Very Nifty Verifier

- Consider A_{TM}:

  $A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$.

- A verifier for A_{TM} would take as input
  - A TM M,
  - a string w, and
  - a certificate c.
- The certificate c should be some evidence that suggests that M accepts w.
- What could our certificate be?

Some Verifiers

- Consider A_{TM}:

$$A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}.$$

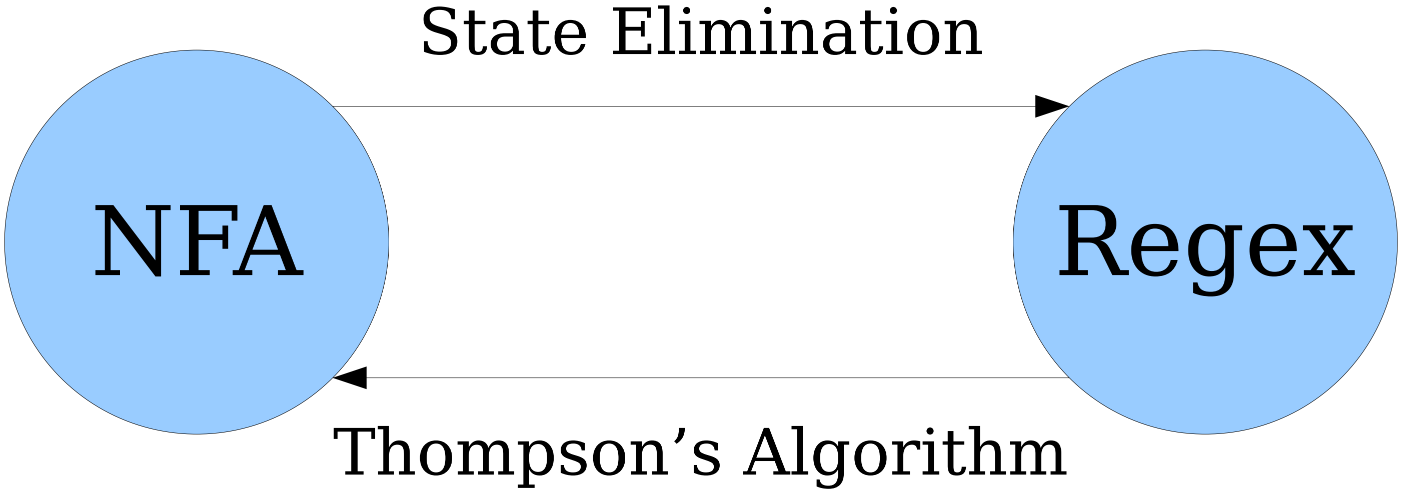
```
bool checkWillAccept(TM M, string w, int c) {
    set up a simulation of M running on w;
    for (int i = 0; i < c; i++) {
       simulate the next step of M running on w;
    }
    return whether M is in an accepting state;
}
```

- Do you see why M accepts w if and only if there is a c such that `checkWillAccept(M, w, c)` returns true?
- Do you see why `checkWillAccept` always halts?

What languages are verifiable?

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

Where We've Been



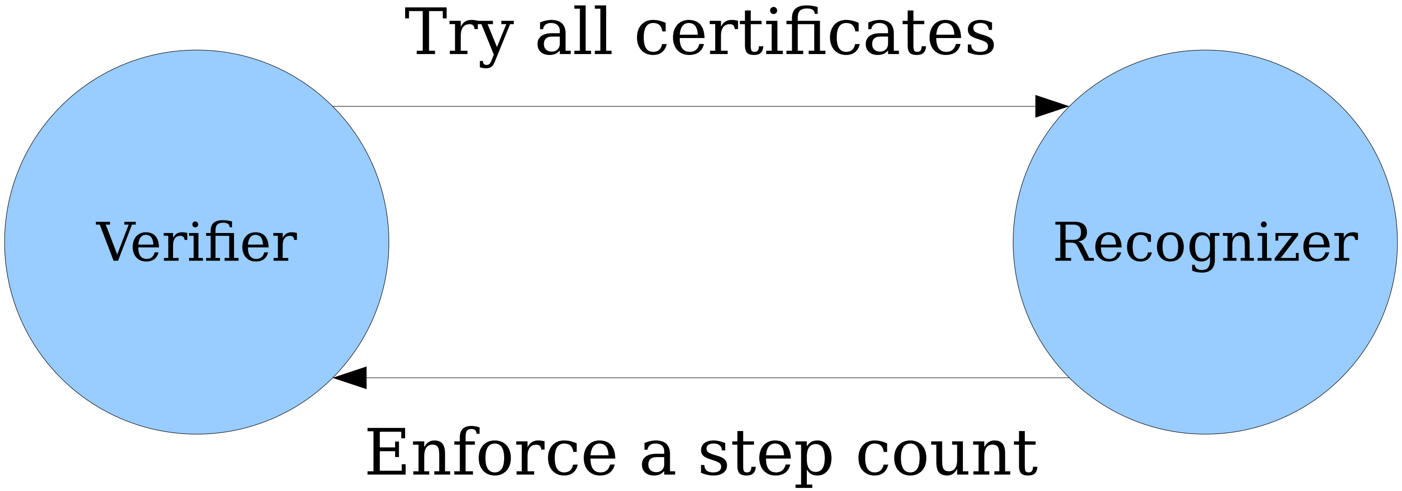
State Elimination

NFA

Regex

Thompson's Algorithm

Where We're Going



Try all certificates

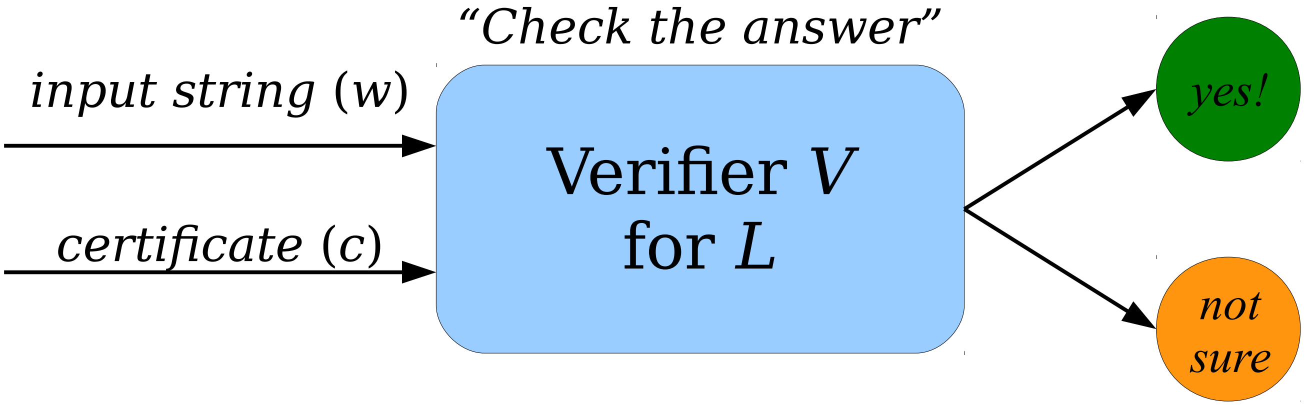
Verifier

Recognizer

Enforce a step count

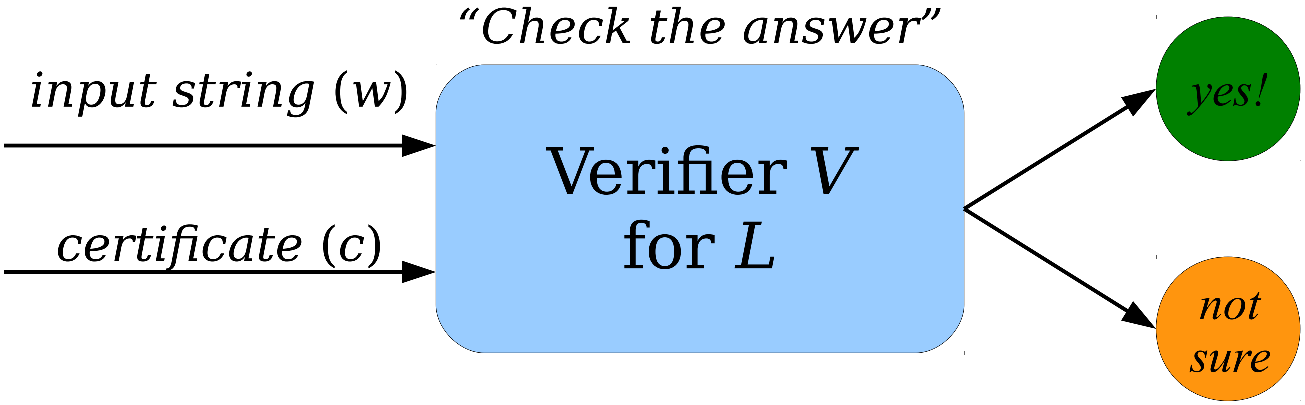
Verifiers and **RE**

- ***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 $L \in \textbf{RE}$.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.



"Check the answer"

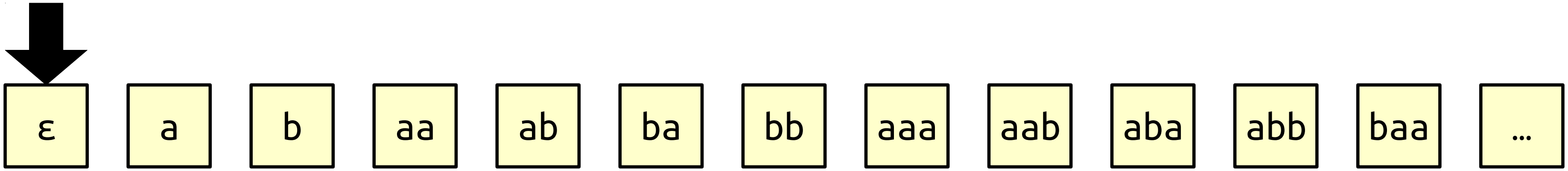
input string (w)

certificate (c)

Verifier V for L

yes!

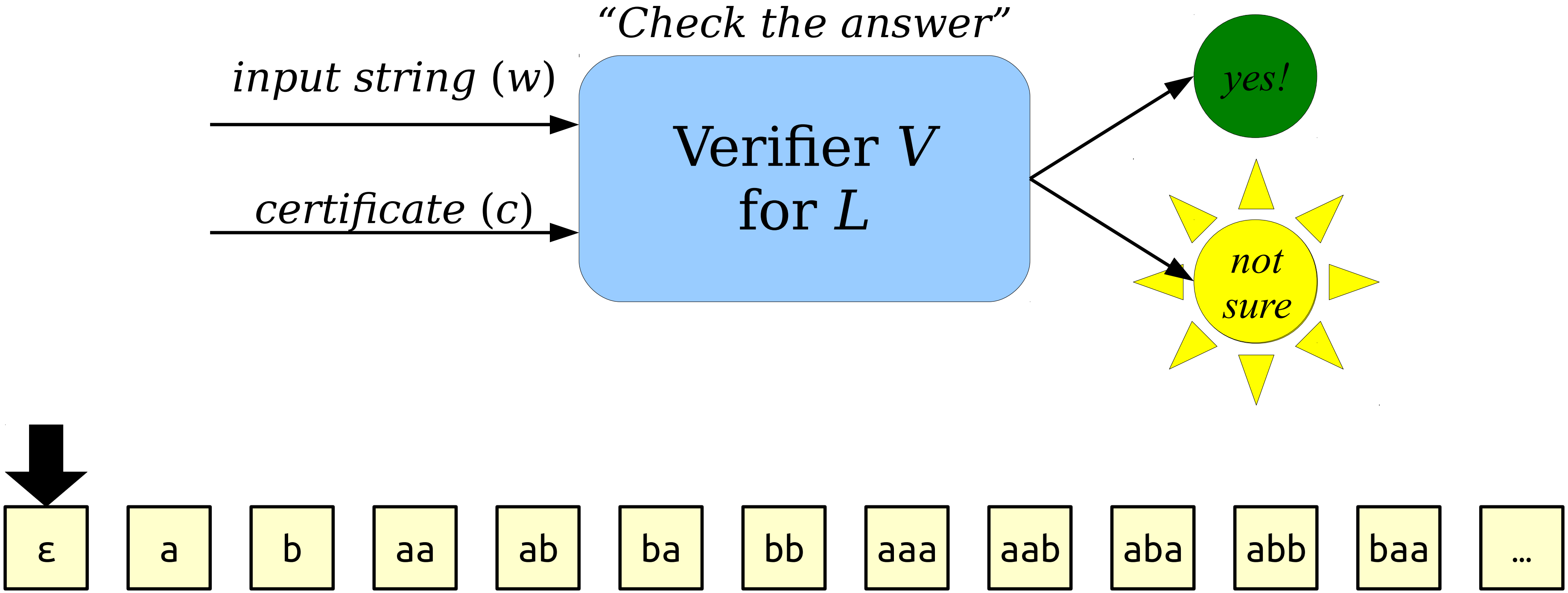
not sure



ε a b aa ab ba bb aaa aab aba abb baa …

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.



"Check the answer"

input string (w)

certificate (c)

Verifier V
for L

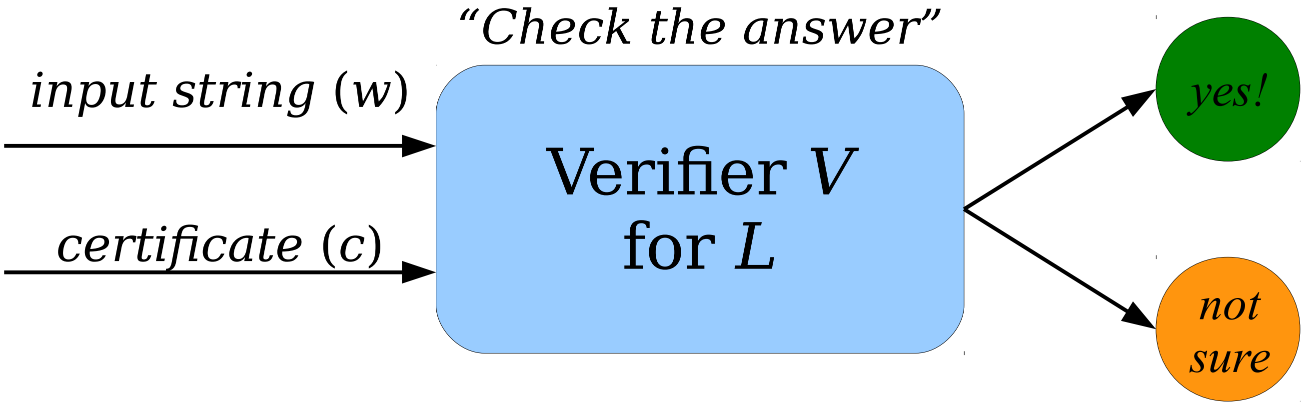
yes!

not sure

ε a b aa ab ba bb aaa aab aba abb baa …

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.



"Check the answer"

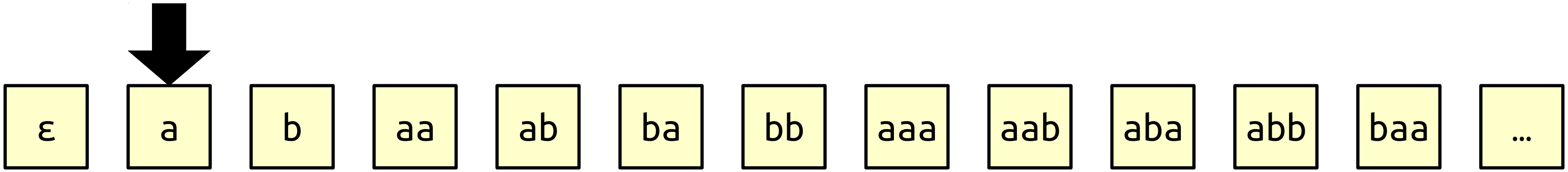
input string (w)

certificate (c)

Verifier V
for L

yes!

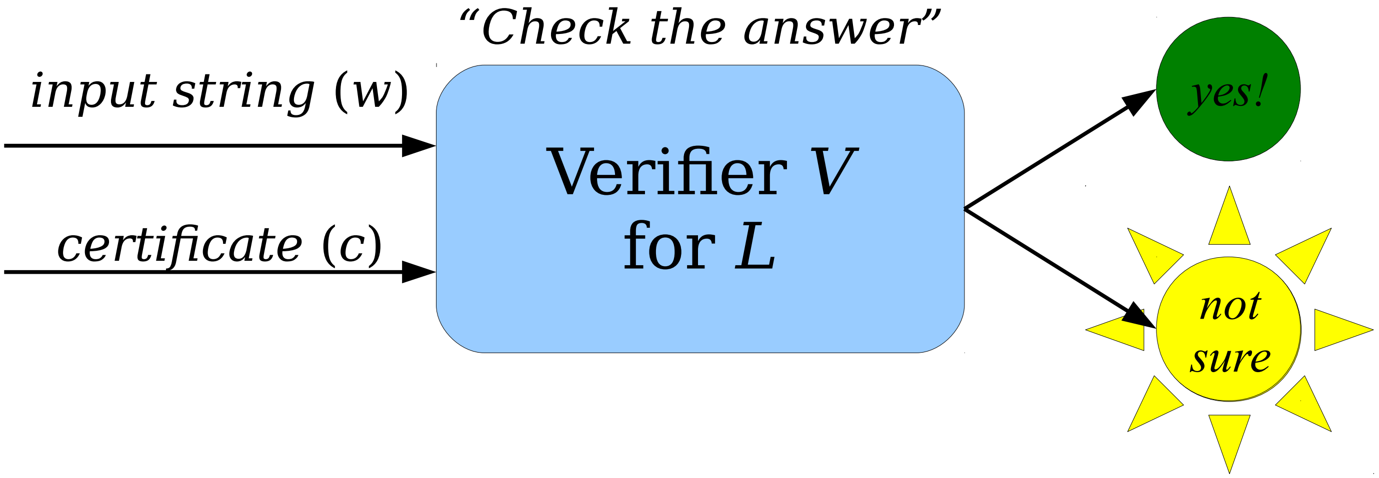
not sure

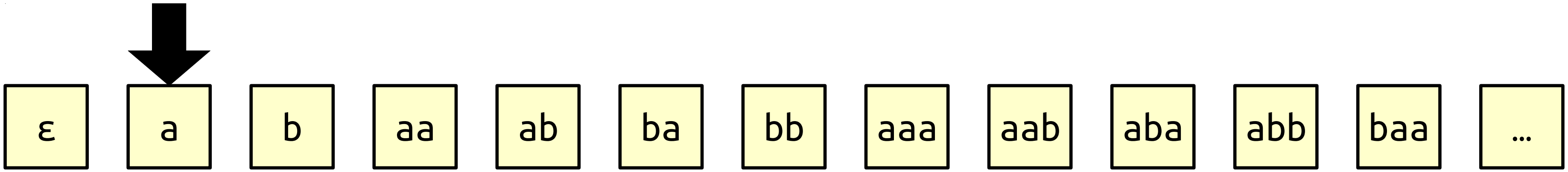


ε a b aa ab ba bb aaa aab aba abb baa …

Verifiers and **RE**

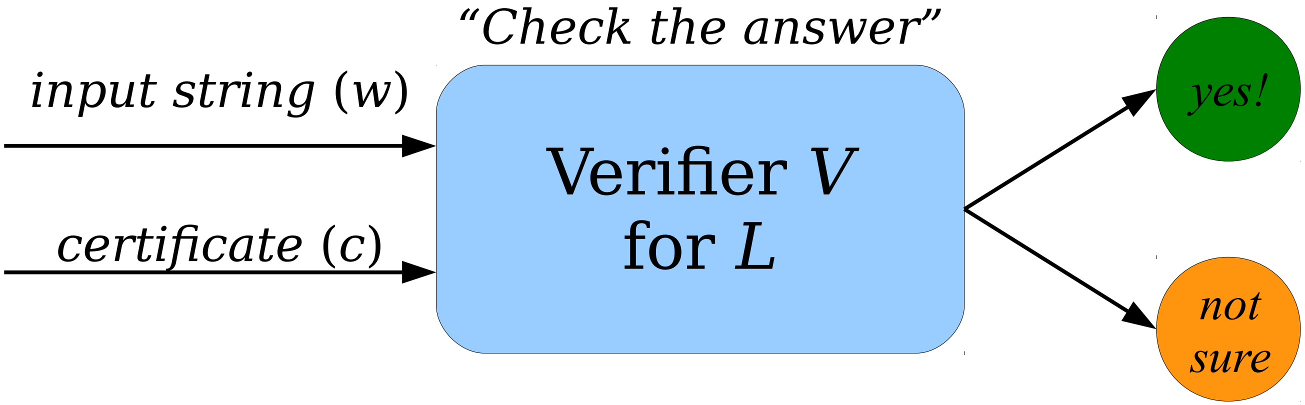
- ***Theorem:*** If there is a verifier V for a language L, then $L \in$ **RE**.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.





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.



"Check the answer"

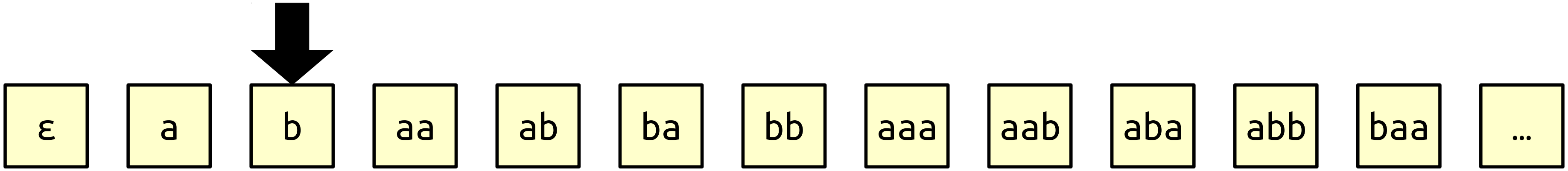
input string (w)

certificate (c)

Verifier V for L

yes!

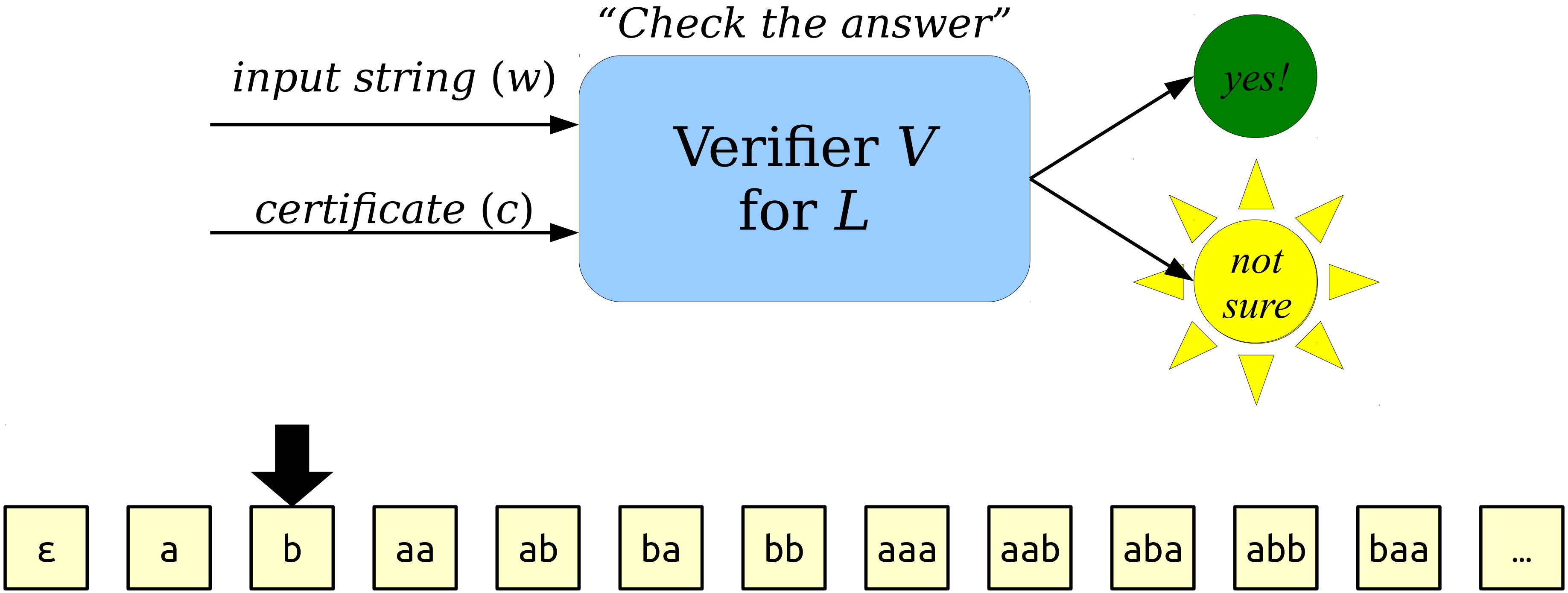
not sure



ε | a | b | aa | ab | ba | bb | aaa | aab | aba | abb | baa | …

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.



“Check the answer”

input string (w)

certificate (c)

Verifier V for L

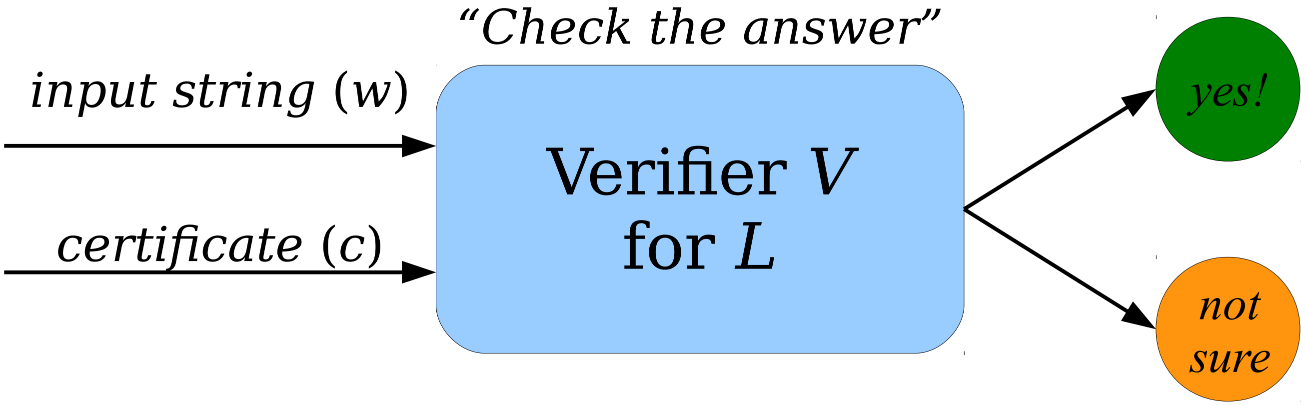
yes!

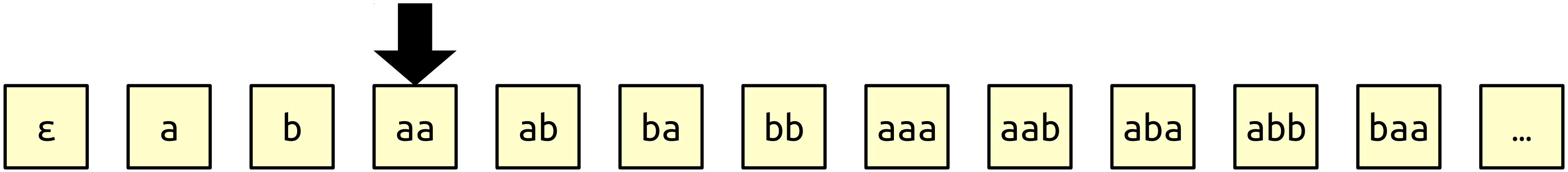
not sure

ε a b aa ab ba bb aaa aab aba abb baa …

Verifiers and **RE**

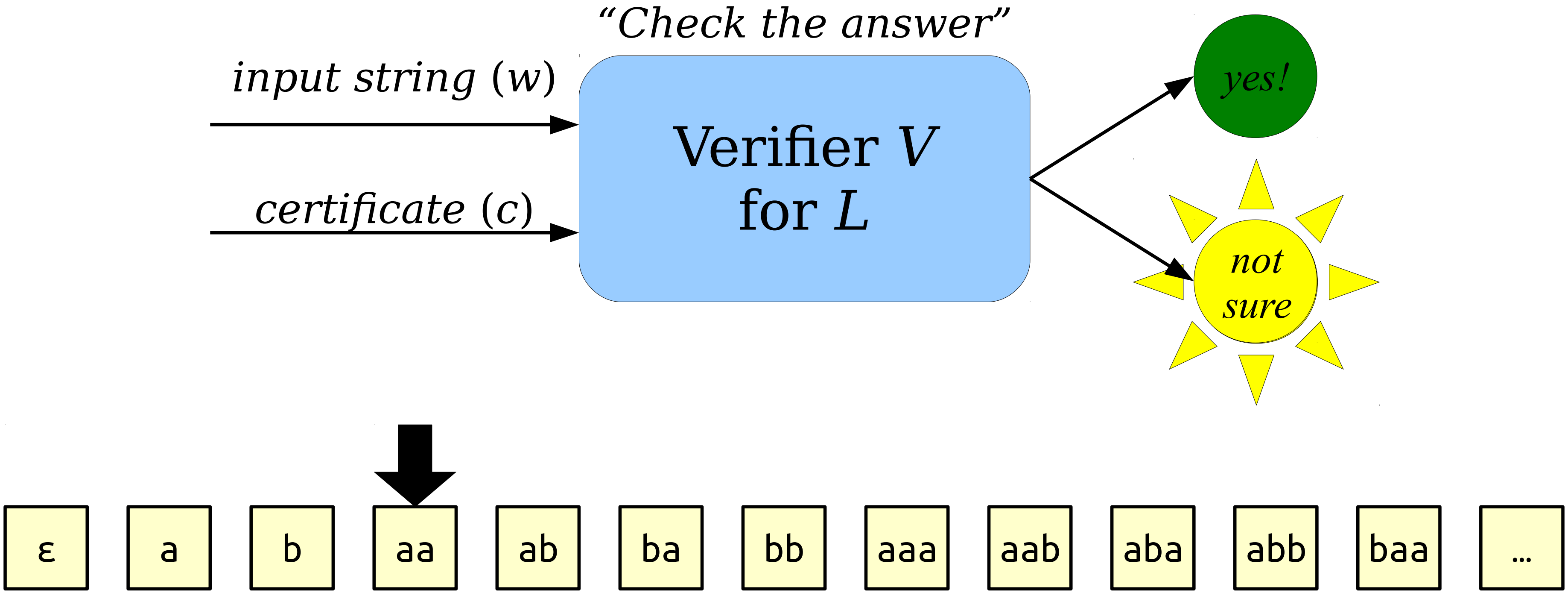
- ***Theorem:*** If there is a verifier *V* for a language *L*, then *L* ∈ **RE**.
- ***Proof goal:*** Given a verifier *V* for a language *L*, find a way to construct a recognizer *M* for *L*.





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.



“Check the answer”

input string (w)

certificate (c)

Verifier V for L

yes!

not sure

ε

a

b

aa

ab

ba

bb

aaa

aab

aba

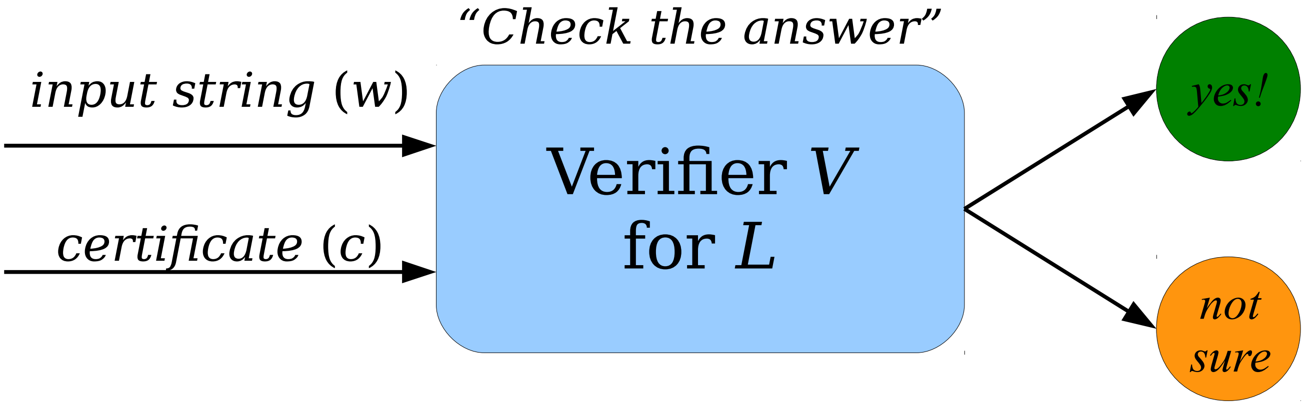
abb

baa

…

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*.



"Check the answer"

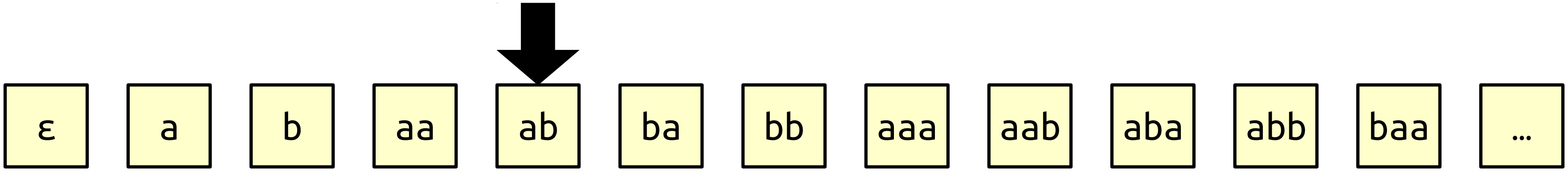
input string (*w*)

certificate (*c*)

Verifier *V*
for *L*

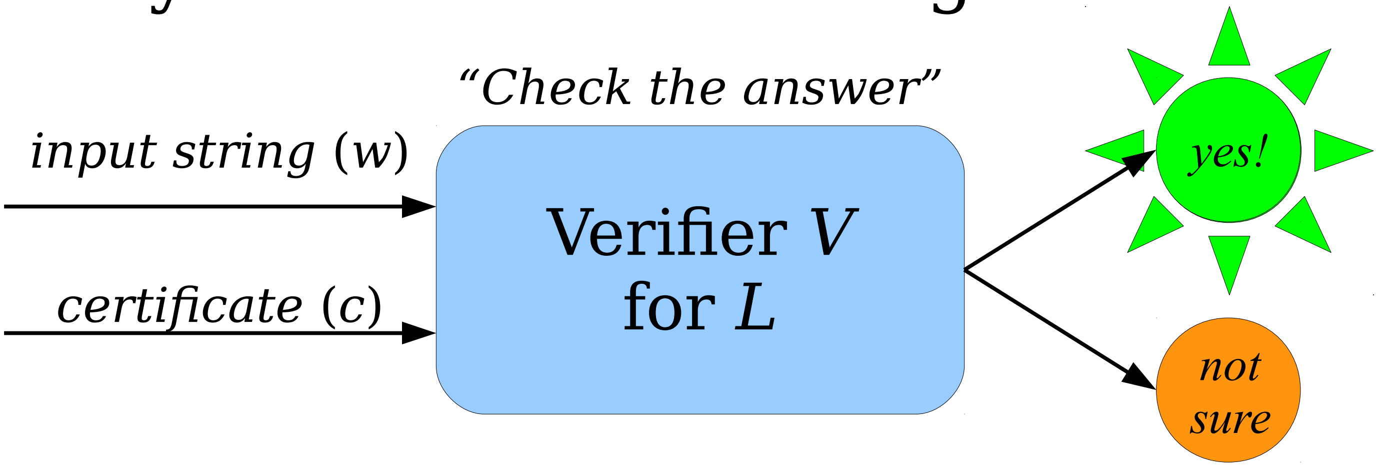
yes!

not sure



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.



"Check the answer"

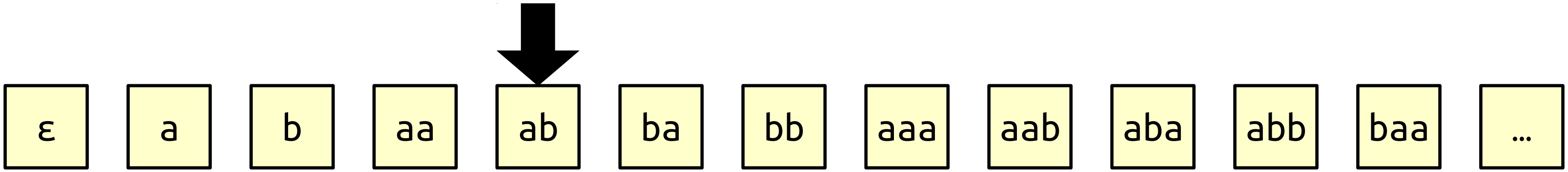
input string (w)

certificate (c)

Verifier V
for L

yes!

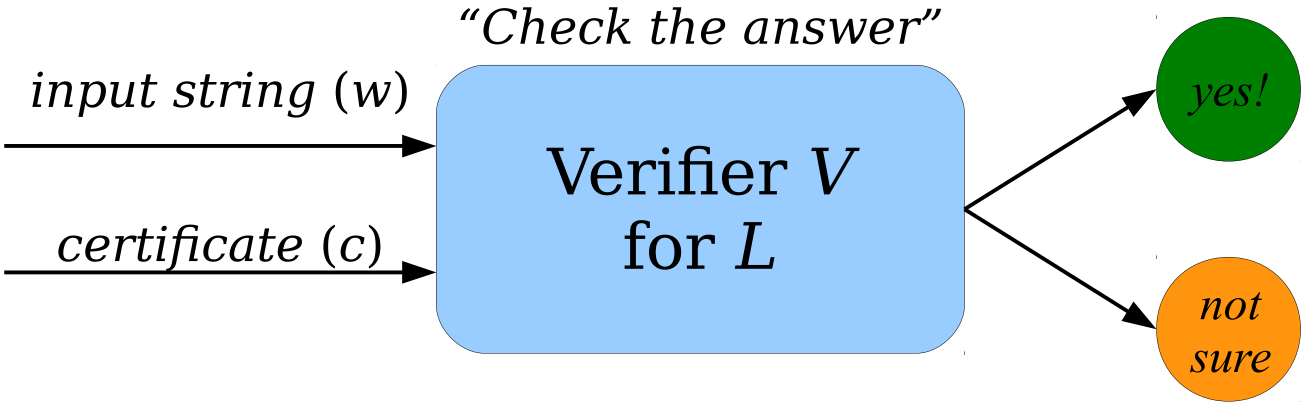
not sure



ε a b aa ab ba bb aaa aab aba abb baa …

Verifiers and **RE**

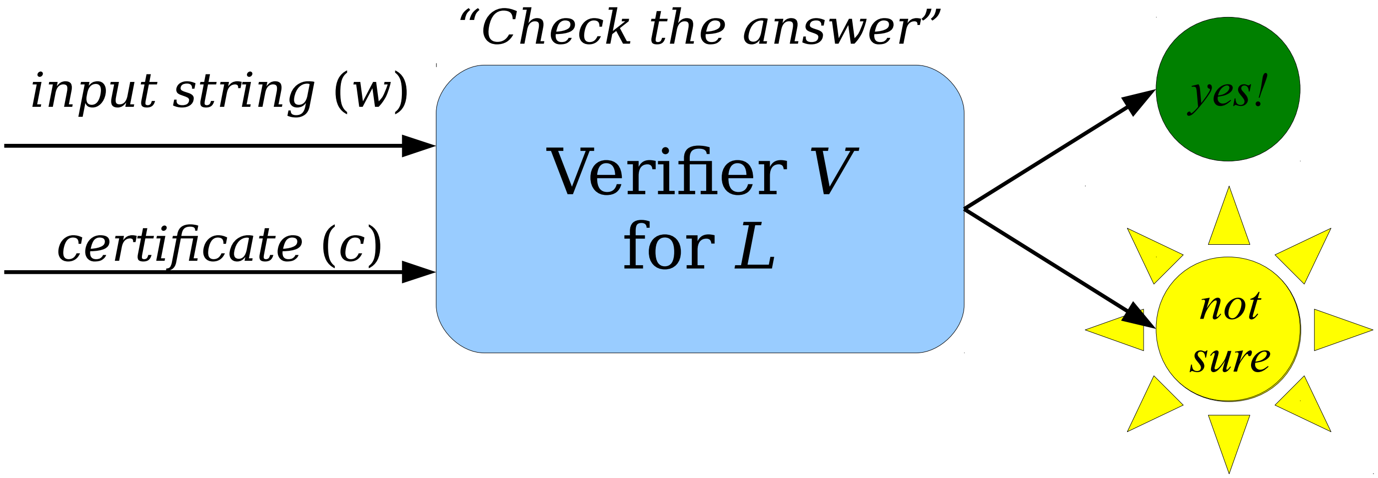
- ***Theorem:*** If there is a verifier V for a language L, then $L \in$ **RE**.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.



ε	a	b	aa	ab	ba	bb	aaa	aab	aba	abb	baa	…

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.



"Check the answer"

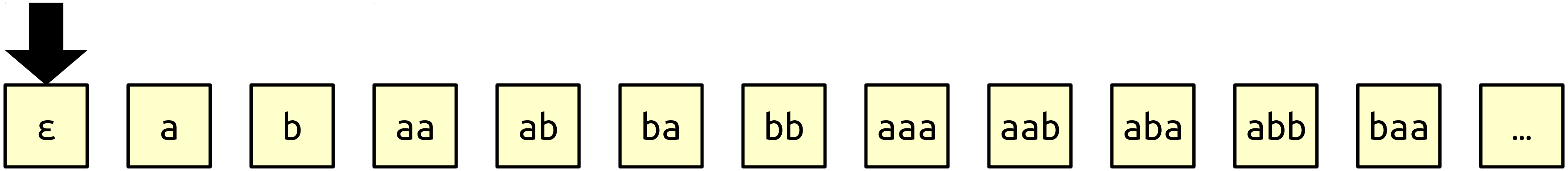
input string (w)

certificate (c)

Verifier V
for L

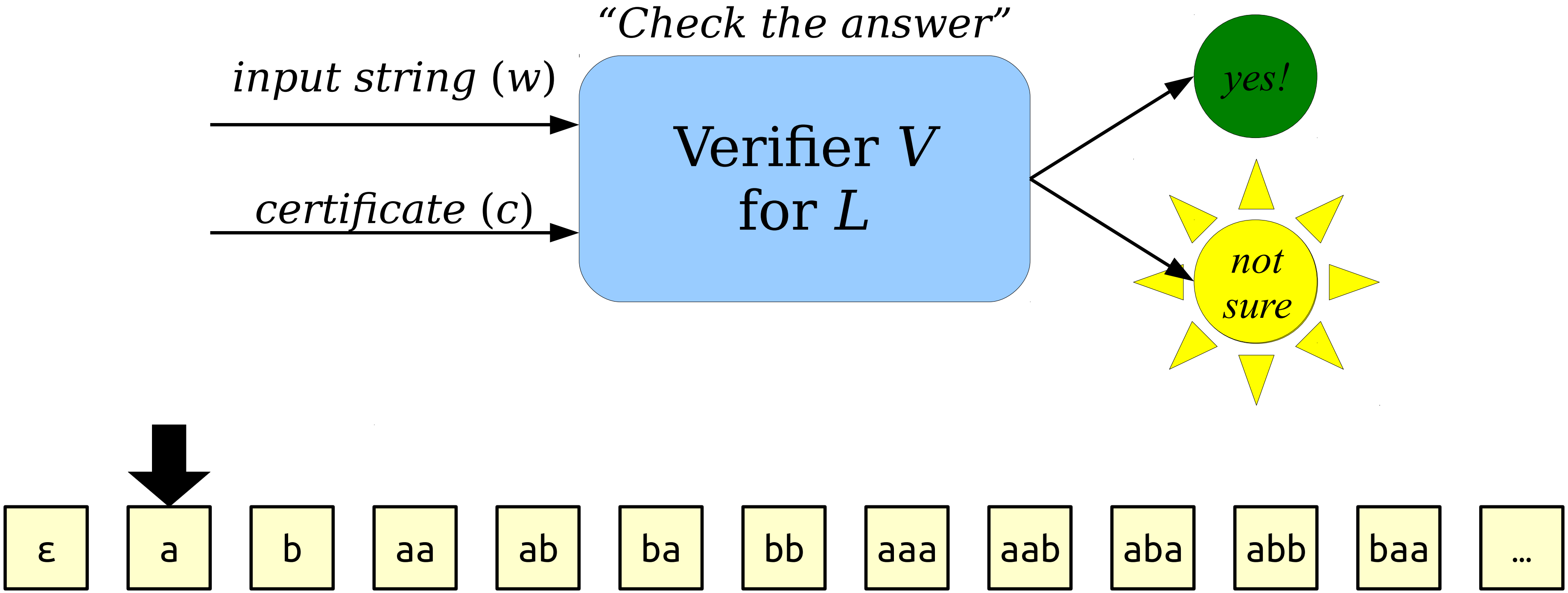
yes!

not sure



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*.



"Check the answer"

input string (*w*)

certificate (*c*)

Verifier *V*
for *L*

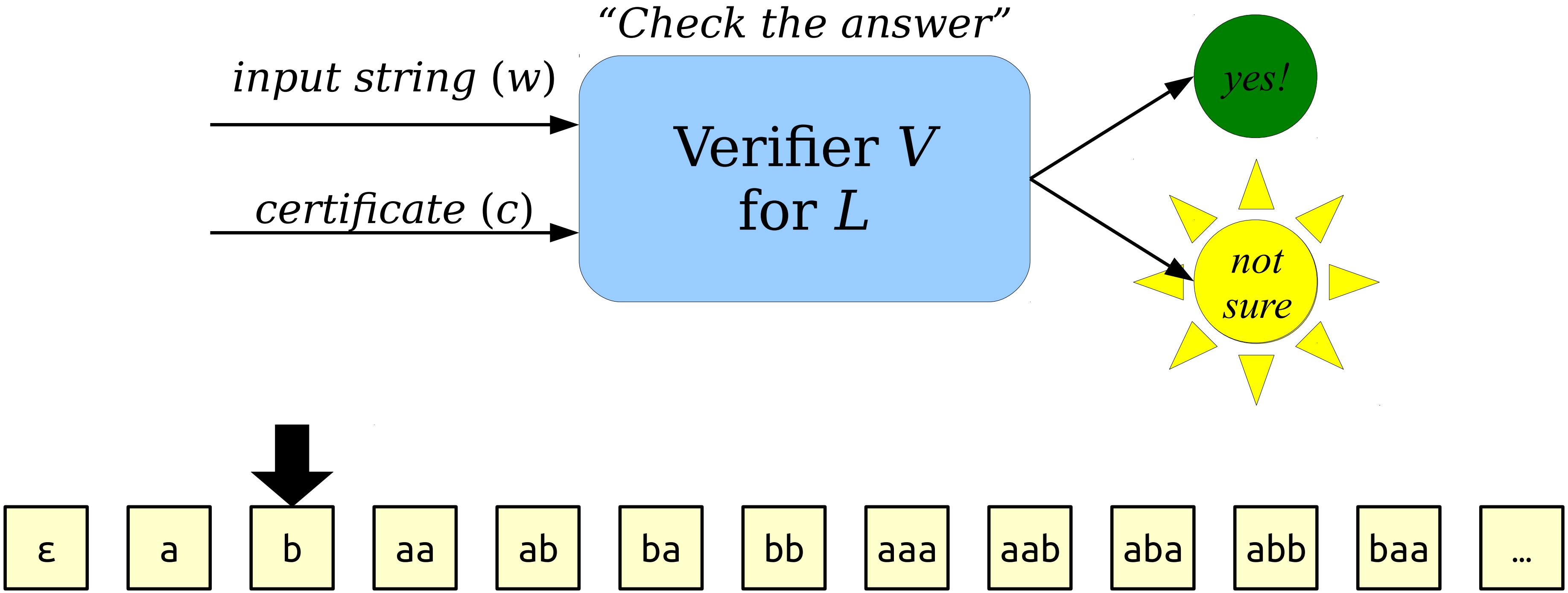
yes!

not sure

ε a b aa ab ba bb aaa aab aba abb baa …

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.



"Check the answer"

input string (w)

certificate (c)

Verifier V for L

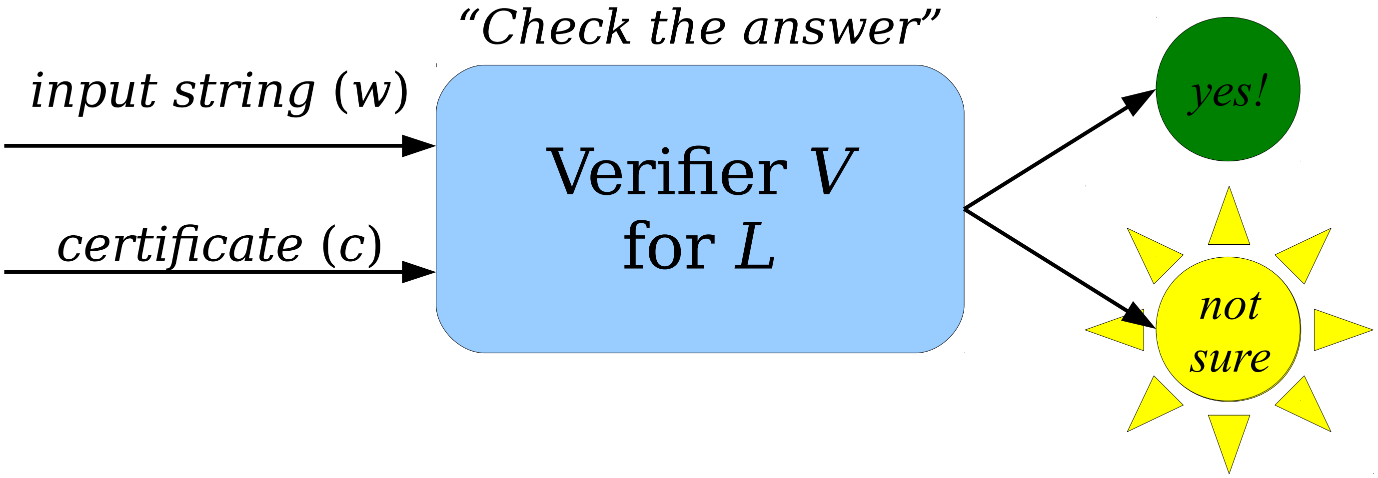
yes!

not sure

ε a b aa ab ba bb aaa aab aba abb baa …

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*.



"Check the answer"

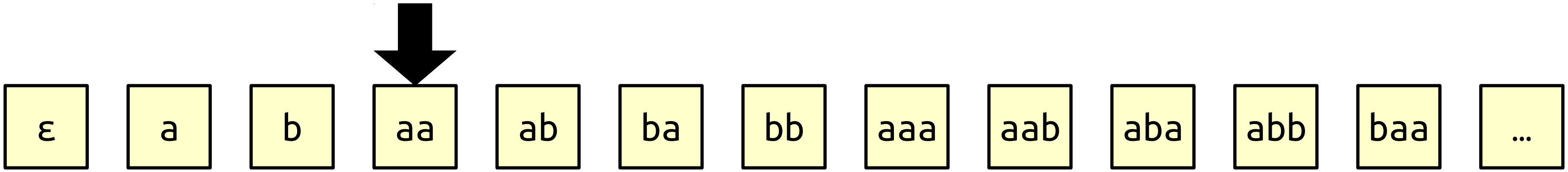
input string (*w*)

certificate (*c*)

Verifier *V*
for *L*

yes!

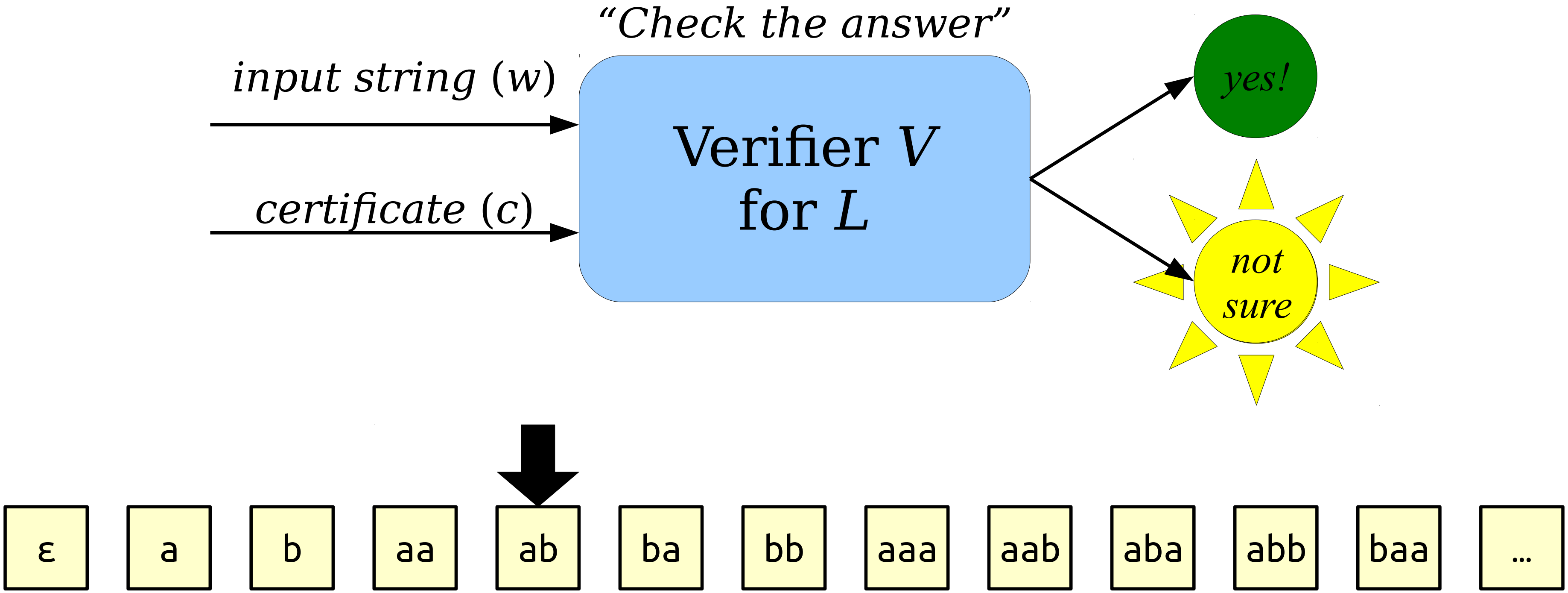
not sure



ε a b aa ab ba bb aaa aab aba abb baa …

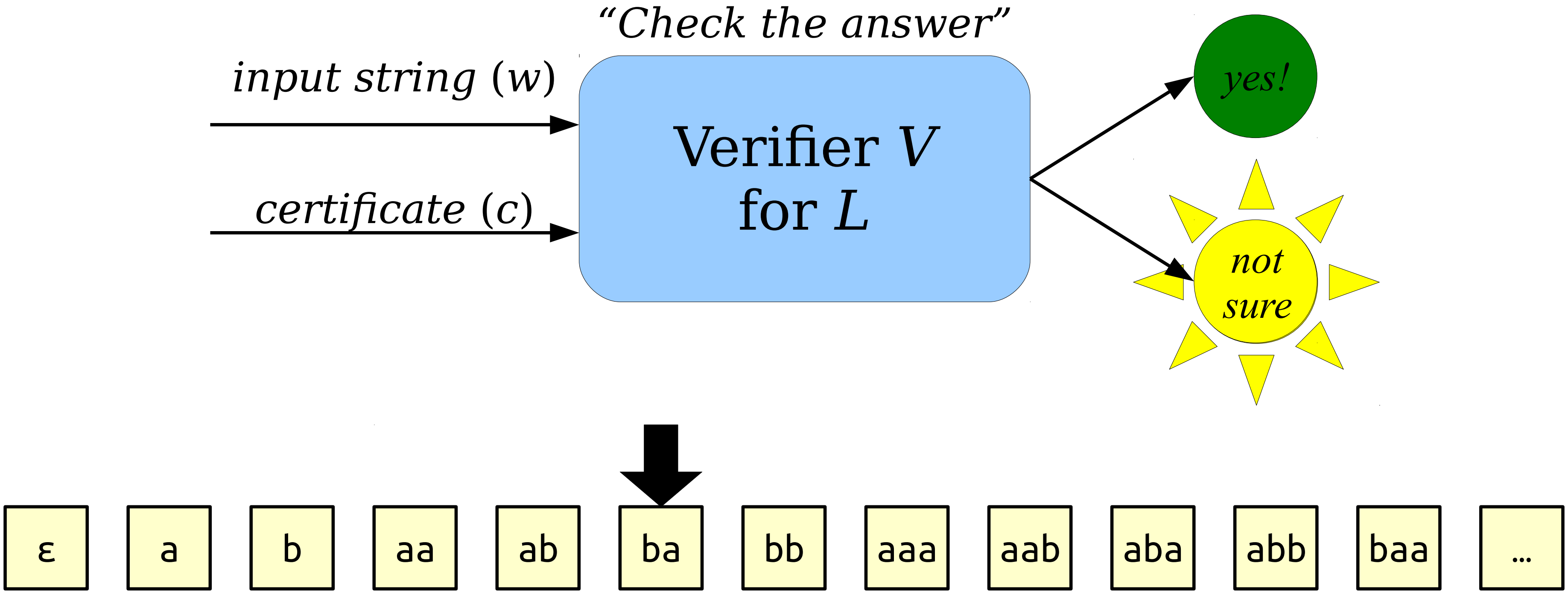
Verifiers and **RE**

- ***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*.



"Check the answer"

input string (*w*)

certificate (*c*)

Verifier *V*
for *L*

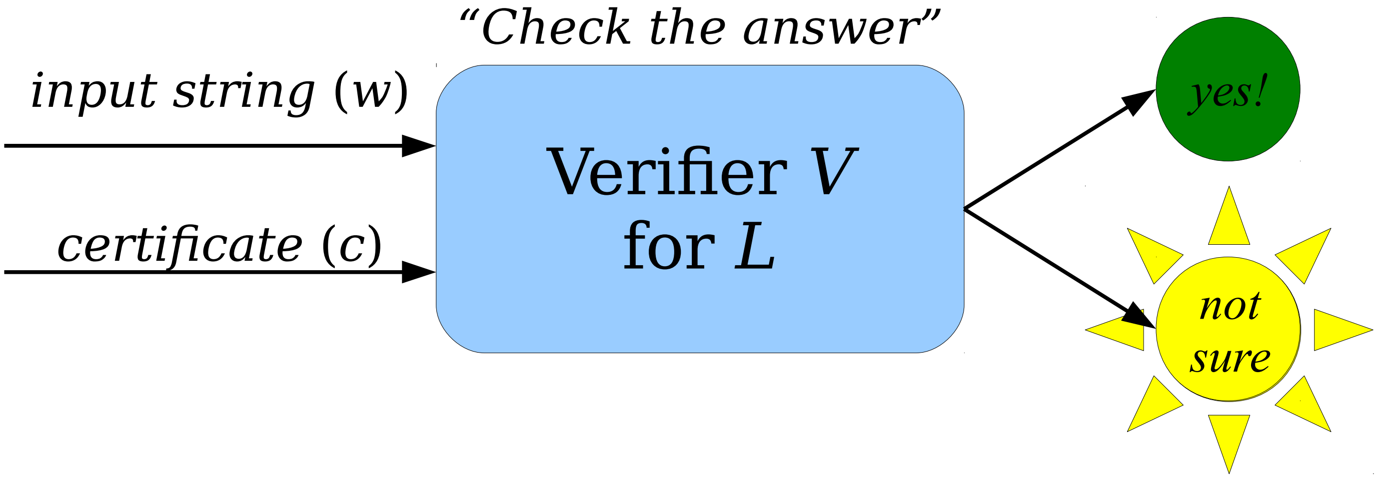
yes!

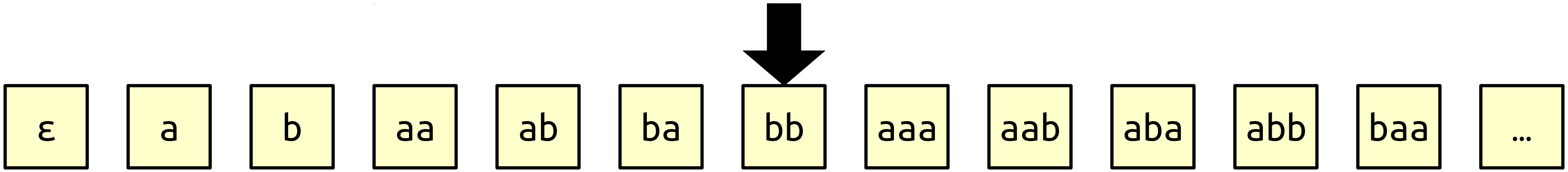
not sure

ε a b aa ab ba bb aaa aab aba abb baa …

Verifiers and **RE**

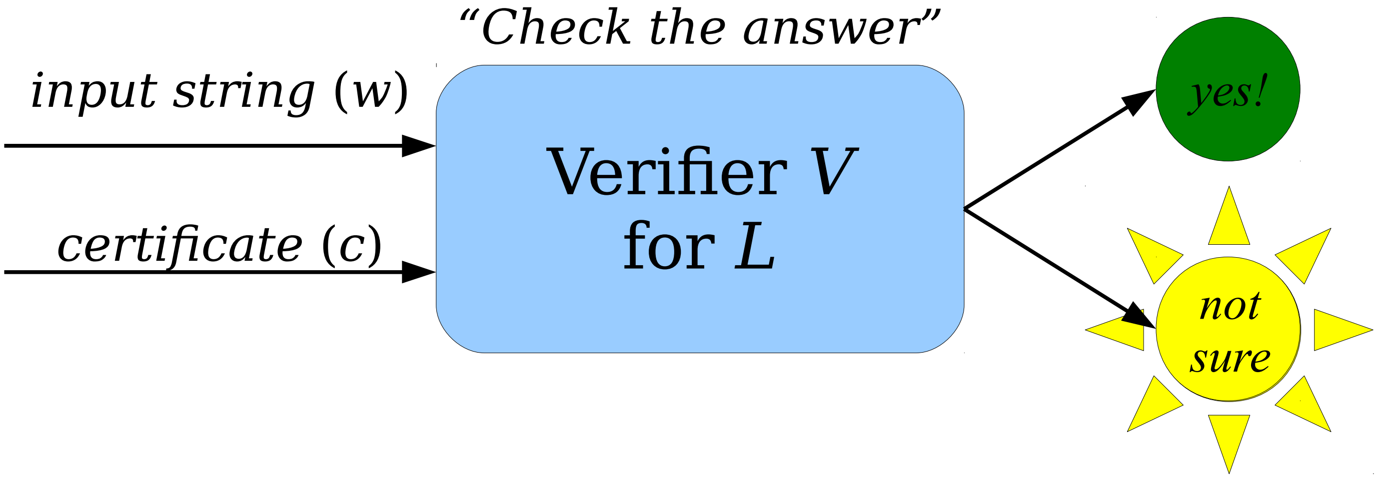
- ***Theorem:*** If there is a verifier V for a language L, then $L \in$ **RE**.
- ***Proof goal:*** Given a verifier V for a language L, find a way to construct a recognizer M for L.





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.



"Check the answer"

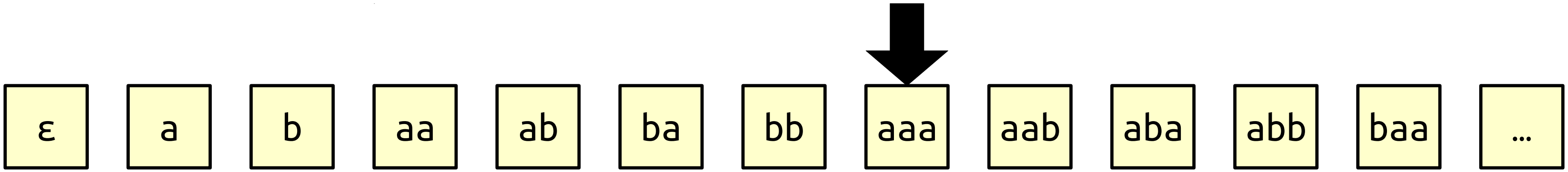
input string (w)

certificate (c)

Verifier V
for L

yes!

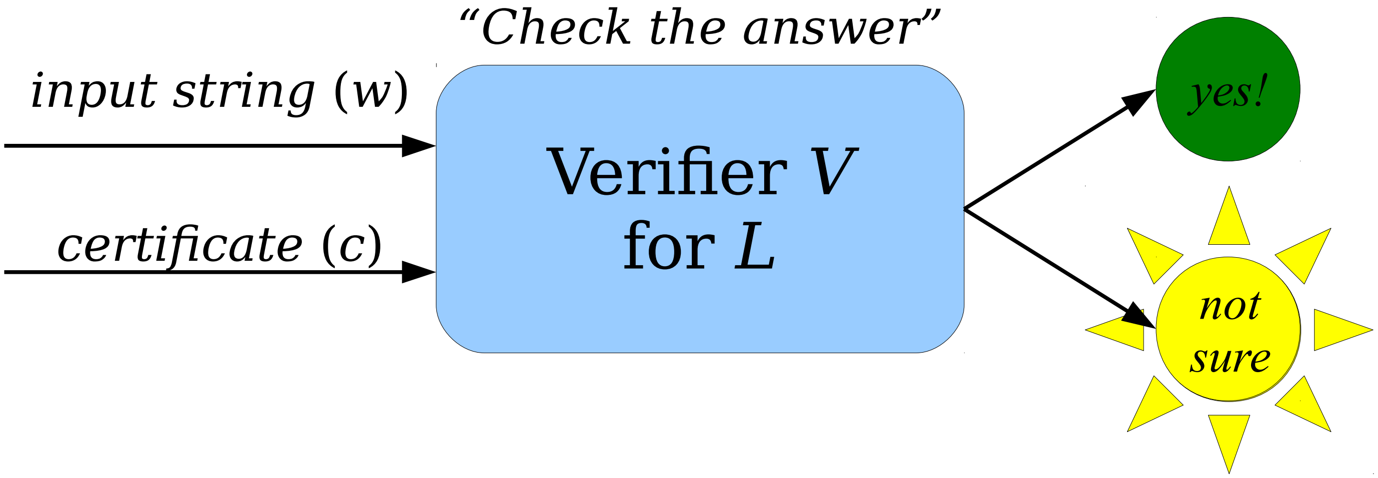
not sure



ε a b aa ab ba bb aaa aab aba abb baa …

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*.



"Check the answer"

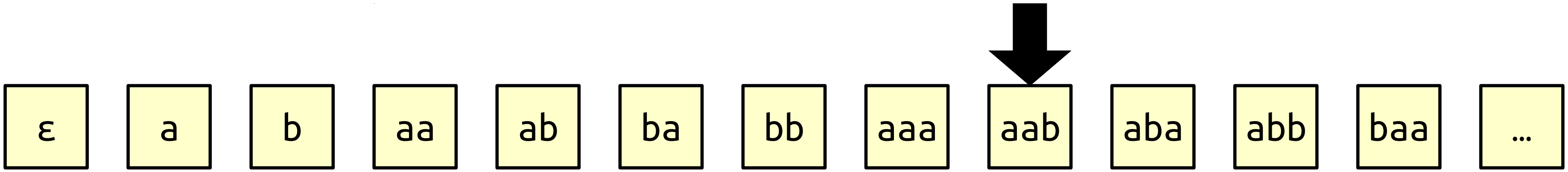
input string (*w*)

certificate (*c*)

Verifier *V* for *L*

yes!

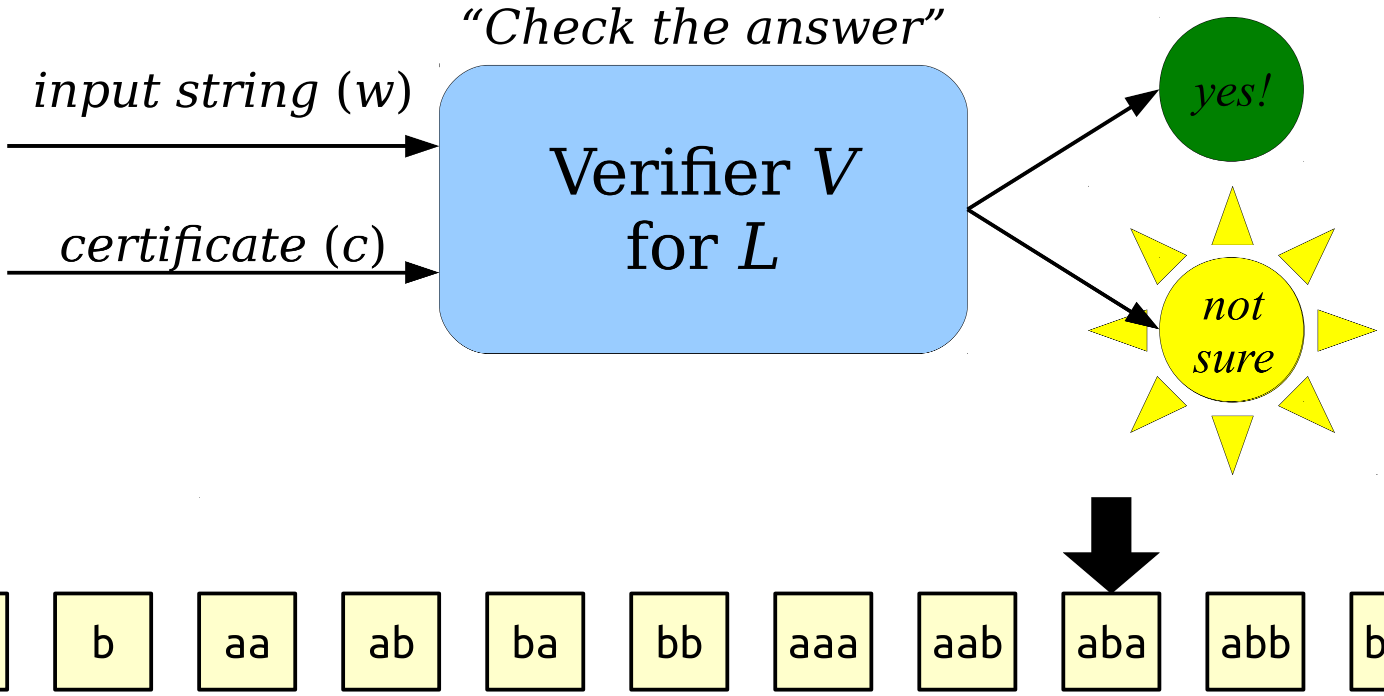
not sure



ε a b aa ab ba bb aaa aab aba abb baa …

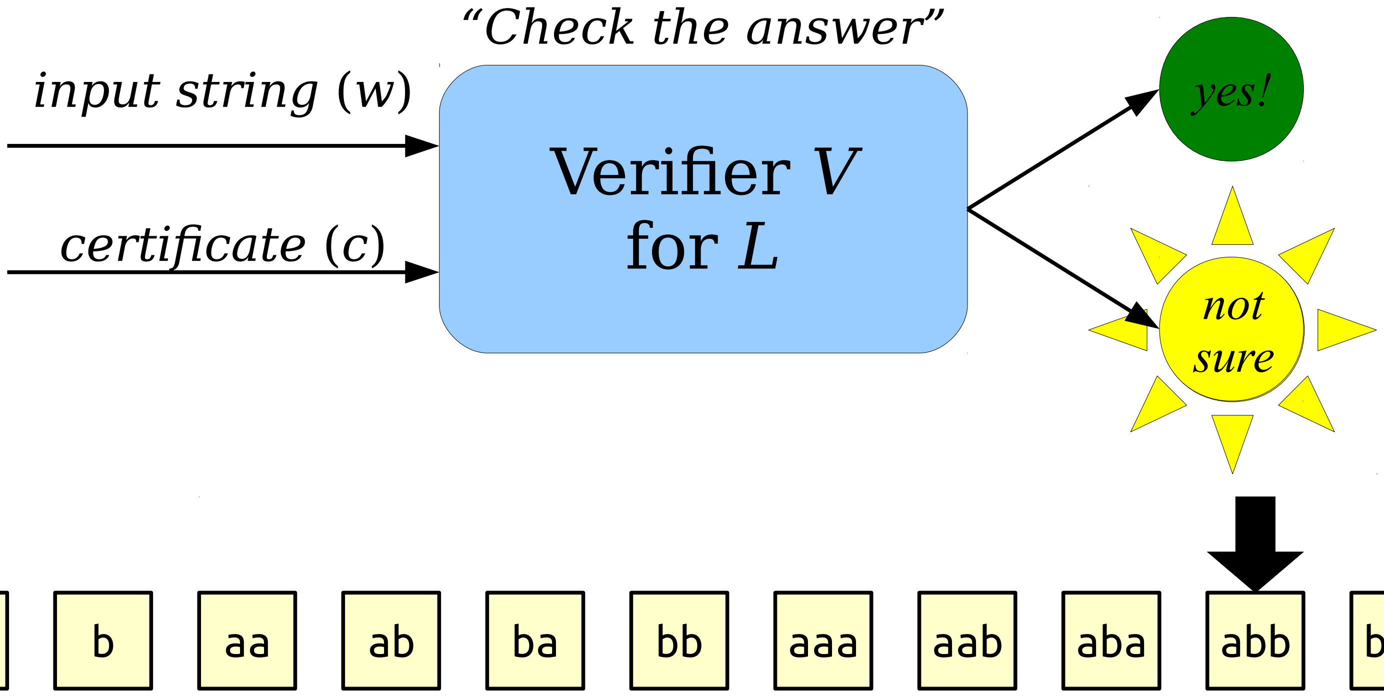
Verifiers and **RE**

- ***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.



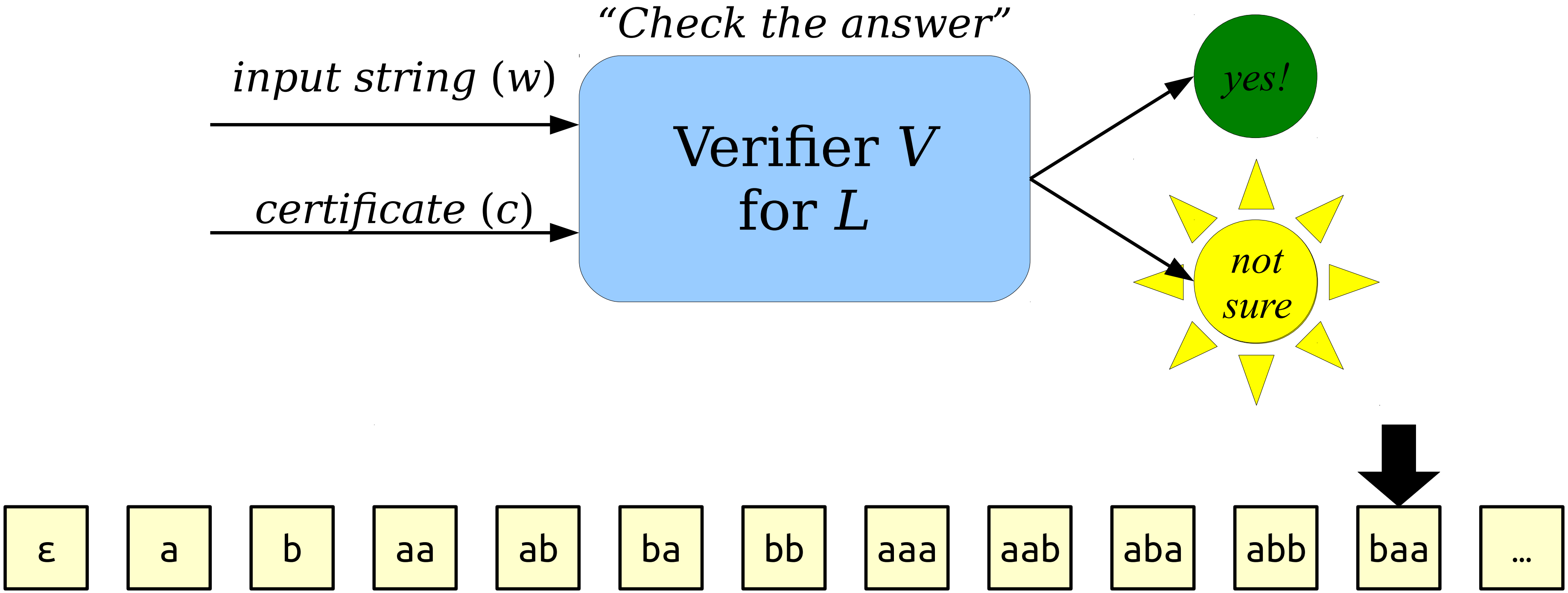
Verifiers and **RE**

- ***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.



"Check the answer"

input string (w)

certificate (c)

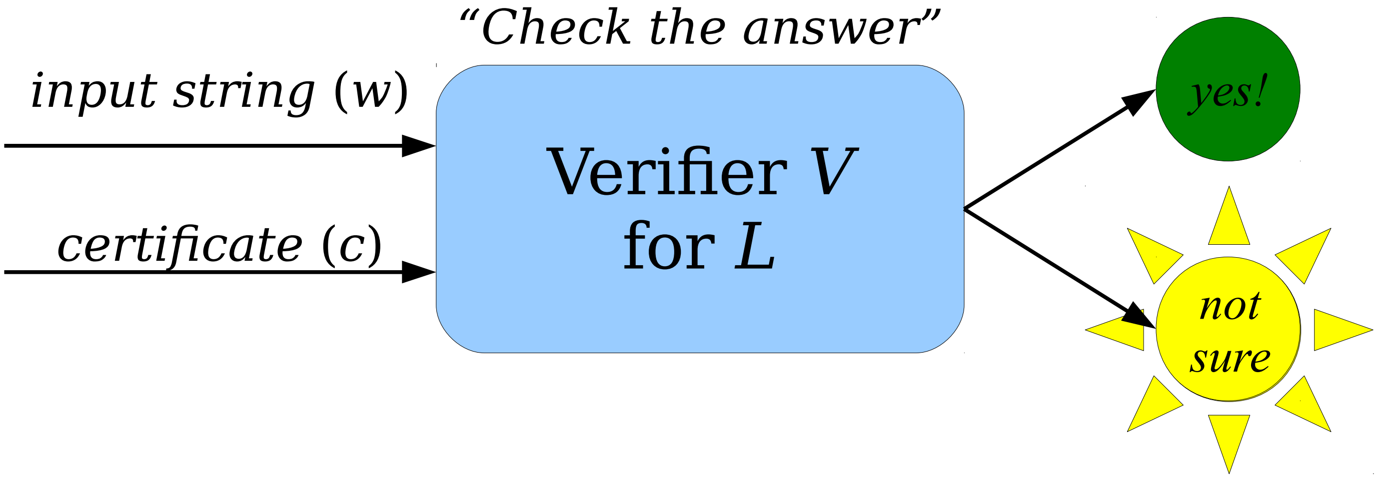
Verifier V for L

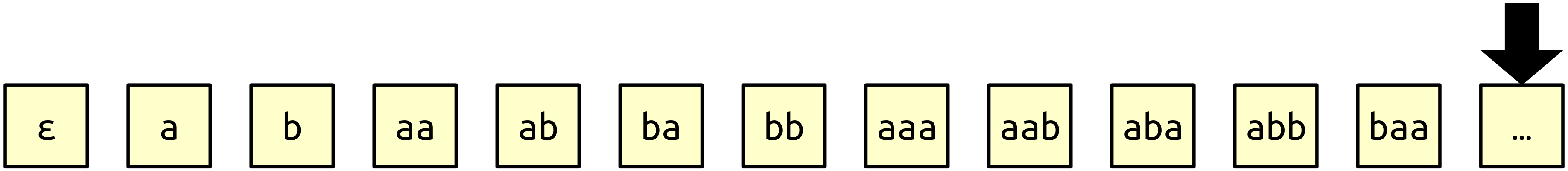
yes!

not sure

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 $L \in$ **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 $L \in$ **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.

We have a recognizer for a language.
We want to turn it into a verifier.
Where did we see this before?

Some Verifie

Observation: This trick of enforcing a step count limits how long M can run for!

Consider A_{TM}:

$$A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}.$$

```
bool checkWillAccept(TM M, string w, int c) {
    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;
}
```

Do you see why M accepts w iff there is some c such that `checkWillAccept(M, w, c)` returns true?

Do you see why `checkWillAccept` always halts?

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. ■

RE and Proofs

- Verifiers and recognizers give two different perspectives on the “proof” intuition for **RE**.
- Verifiers are explicitly built to check proofs that strings are in the language.
  - If you know that some string w belongs to the language and you have the proof of it, you can convince someone else that $w \in L$.
- You can think of a recognizer as a device that “searches” for a proof that $w \in L$.
  - If it finds it, great!
  - If not, it might loop forever.

RE and Proofs

- If the **RE** languages represent languages where membership can be proven, what does a non-**RE** language look like?
- Intuitively, a language is *not* in **RE** if there is no general way to prove that a given string $w \in L$ actually belongs to L.
- In other words, even if you knew that a string was in the language, you may never be able to convince anyone of it!

Finding Non-**RE** Languages

Finding Non-**RE** Languages

- Right now, we know that non-**RE** languages exist, but we have no idea what they look like.
- How might we find one?

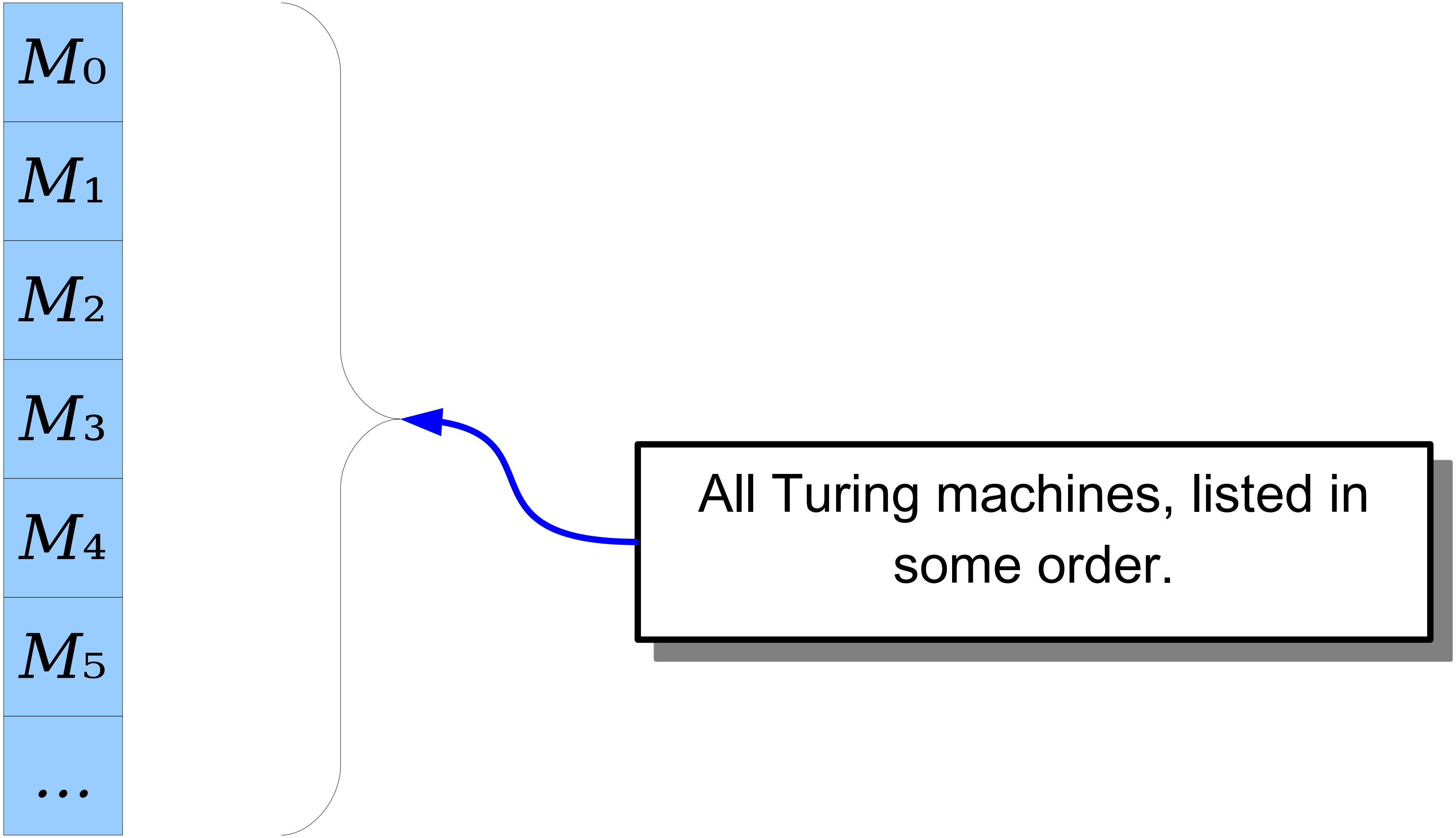
Recognizers and Recognizability

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

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

- This above description applies to all strings, including strings that, by pure coincidence, happen to be encodings of TMs.
- 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
$\dots$



M_0

M_1

M_2

M_3

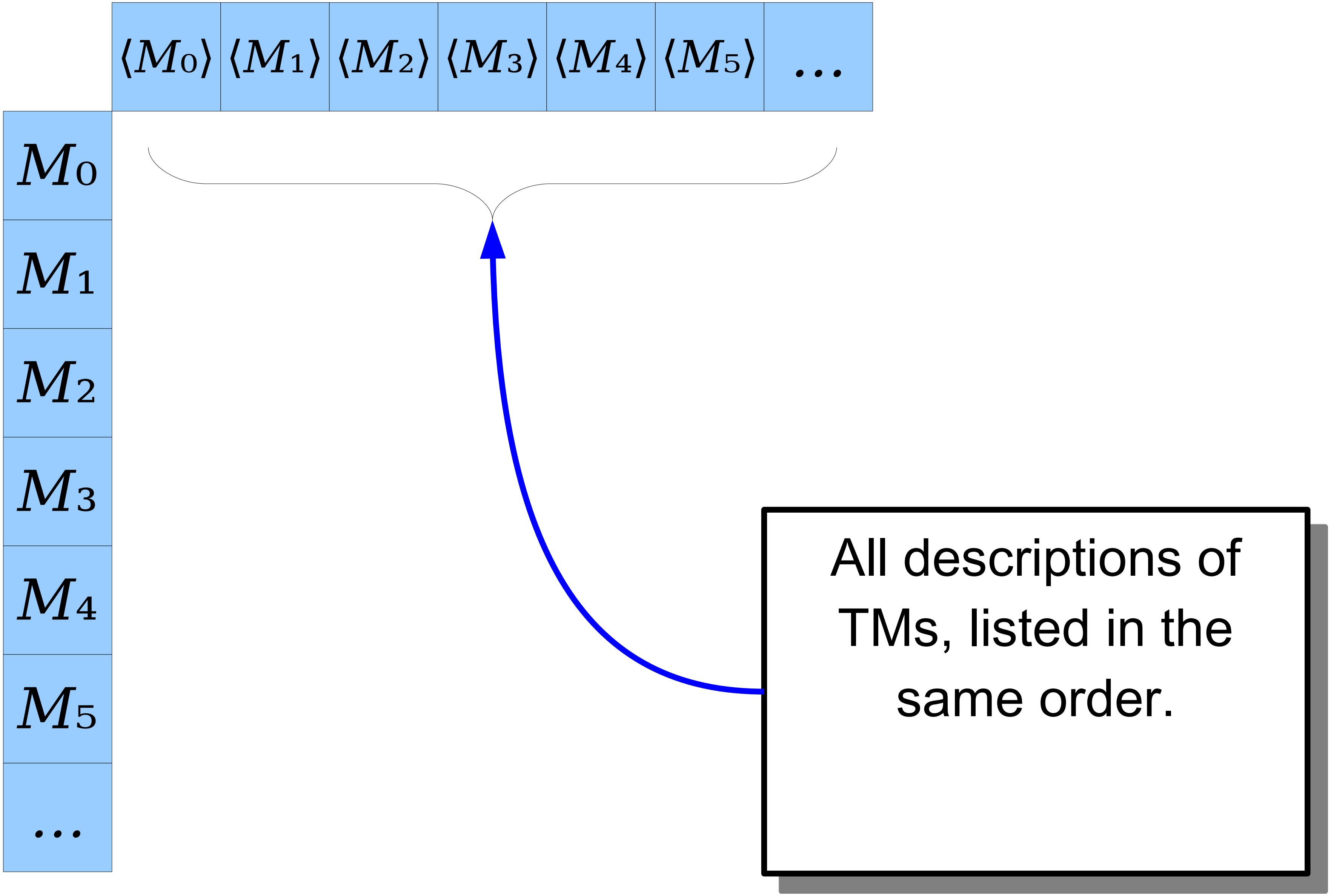
M_4

M_5

$\dots$

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							
$\dots$							



	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0							
M_1							
M_2							
M_3							
M_4							
M_5							
…							

All descriptions of TMs, listed in the same order.

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

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

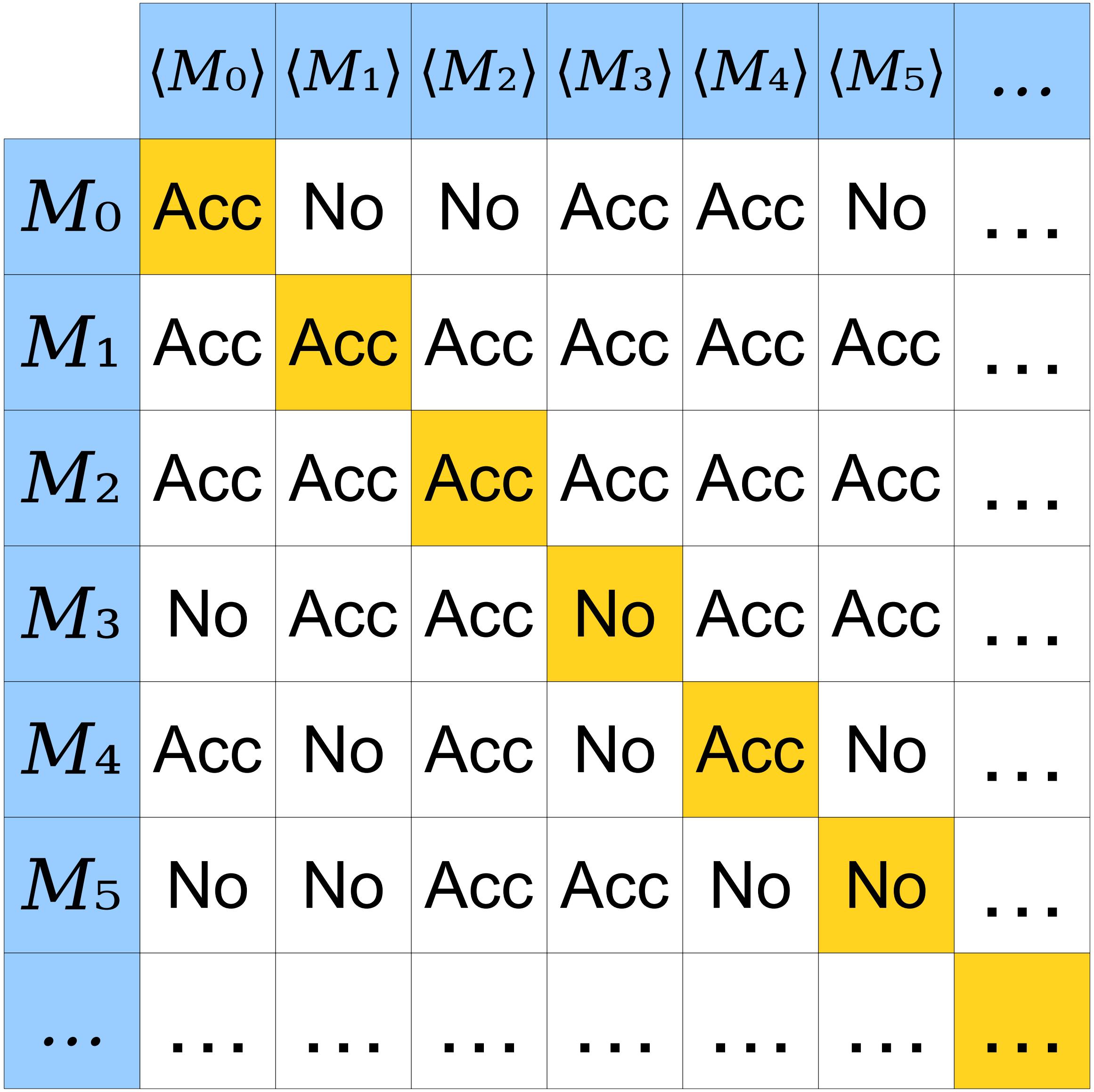
	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	$\ldots$
M_0	Acc	No	No	Acc	Acc	No	$\ldots$
M_1	Acc	Acc	Acc	Acc	Acc	Acc	$\ldots$
M_2	Acc	Acc	Acc	Acc	Acc	Acc	$\ldots$
M_3							
M_4							
M_5							
$\ldots$							

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
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							
…							

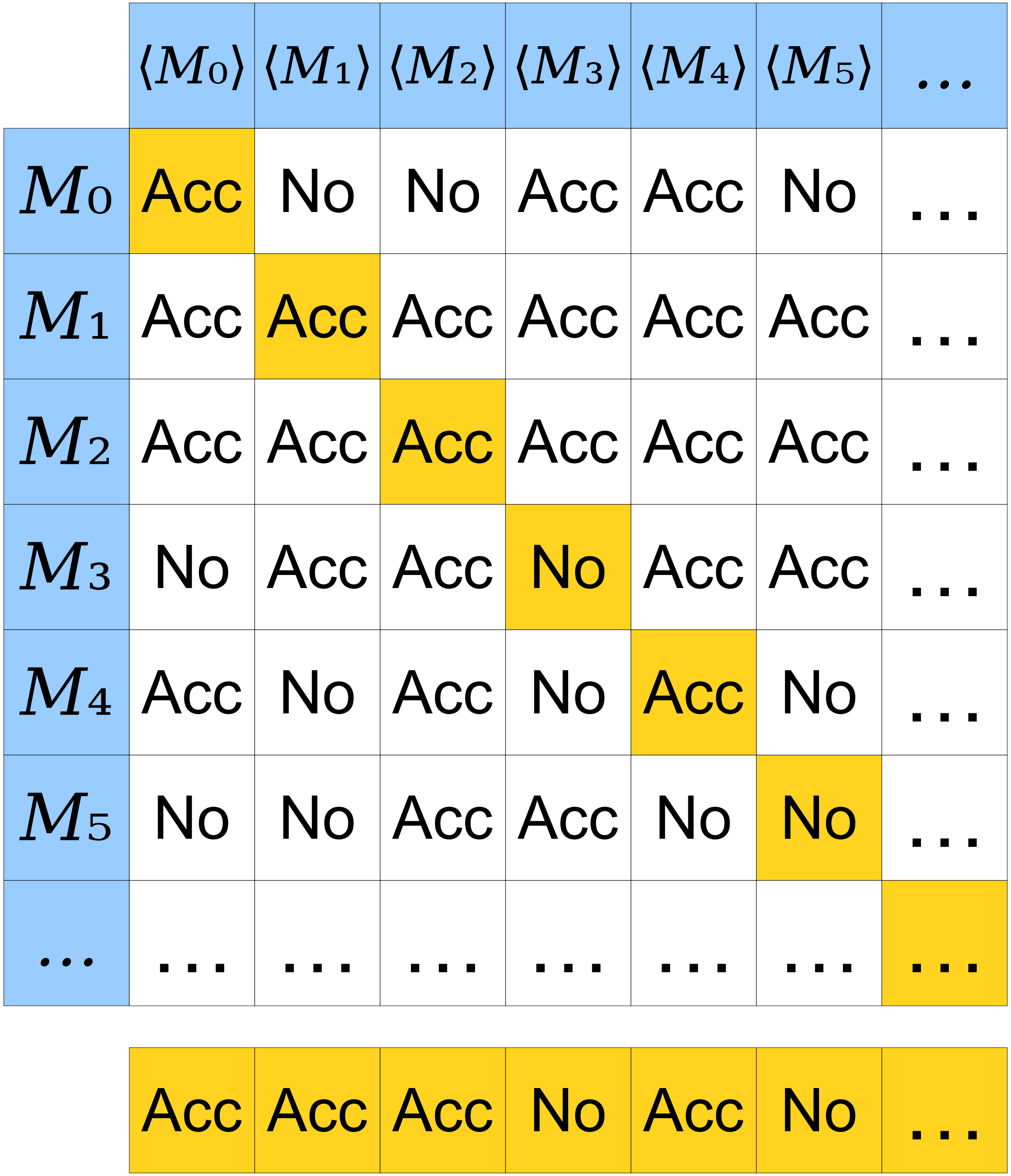
	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5							
…							

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
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	…
…							

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
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	…
…	…	…	…	…	…	…	…

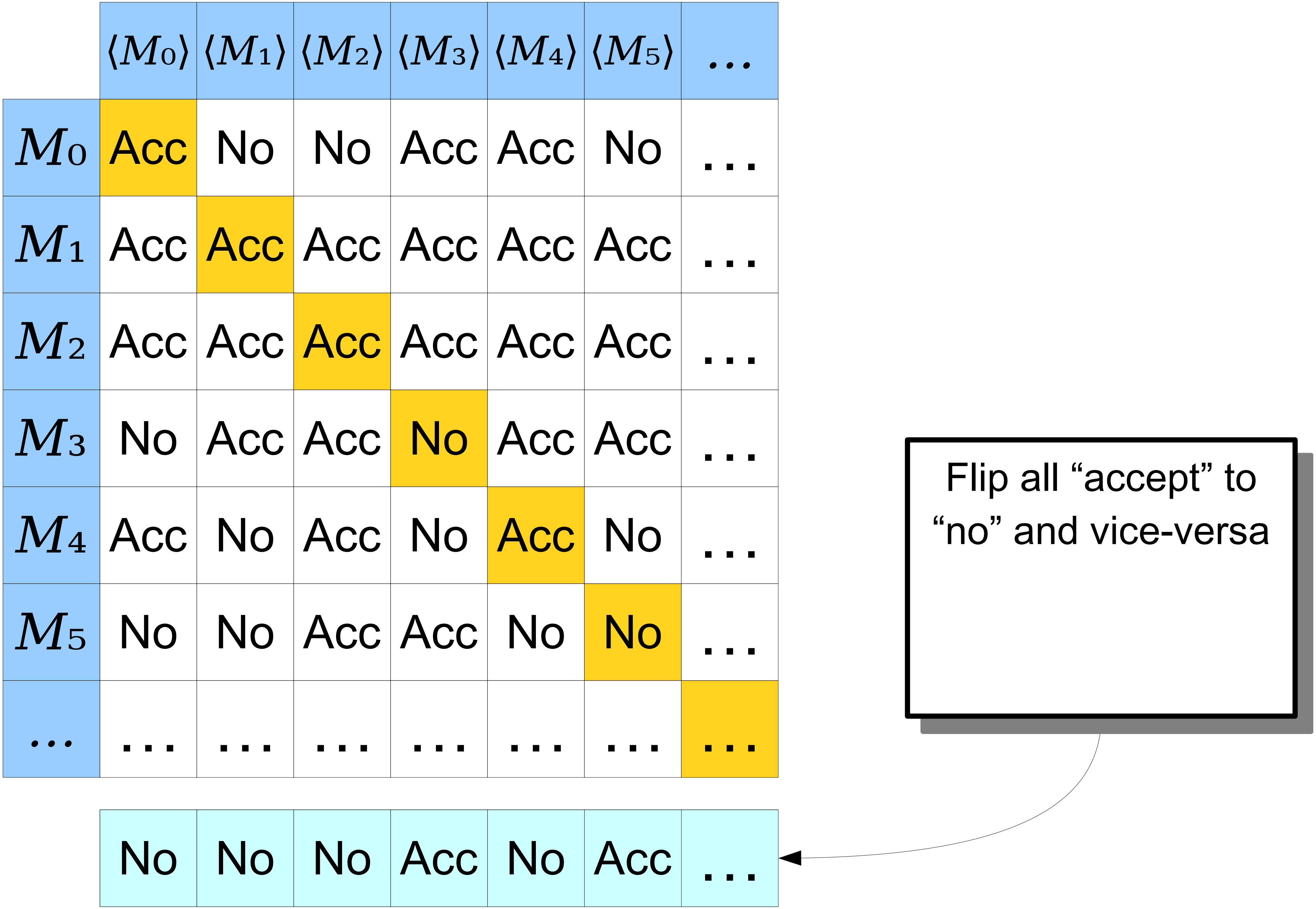


	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	$\ldots$
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc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



	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	$\dots$
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	...
$\dots$	...	...	...	...	...	...	...

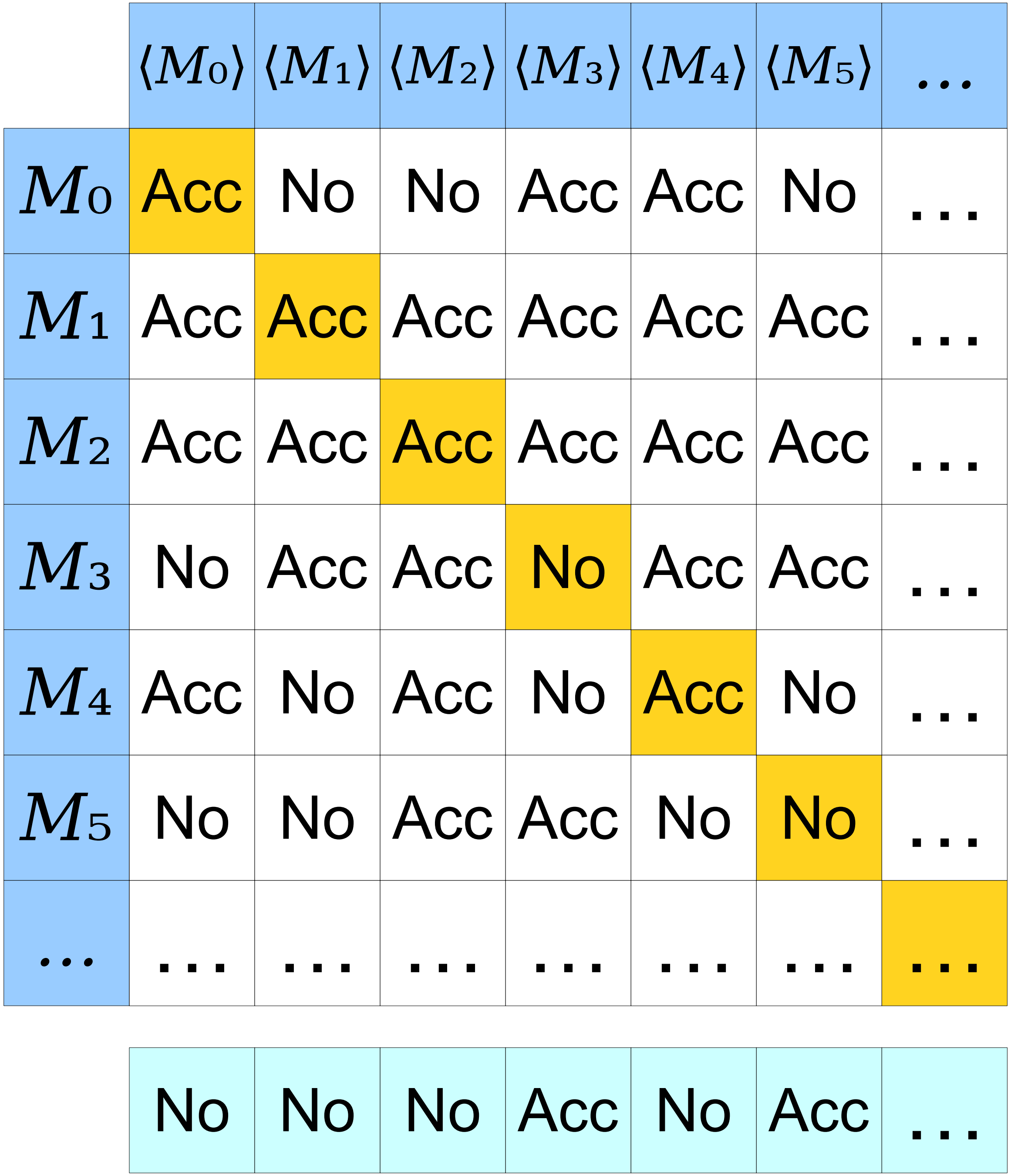
Acc	Acc	Acc	No	Acc	No	...



	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
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	…
…	…	…	…	…	…	…	…

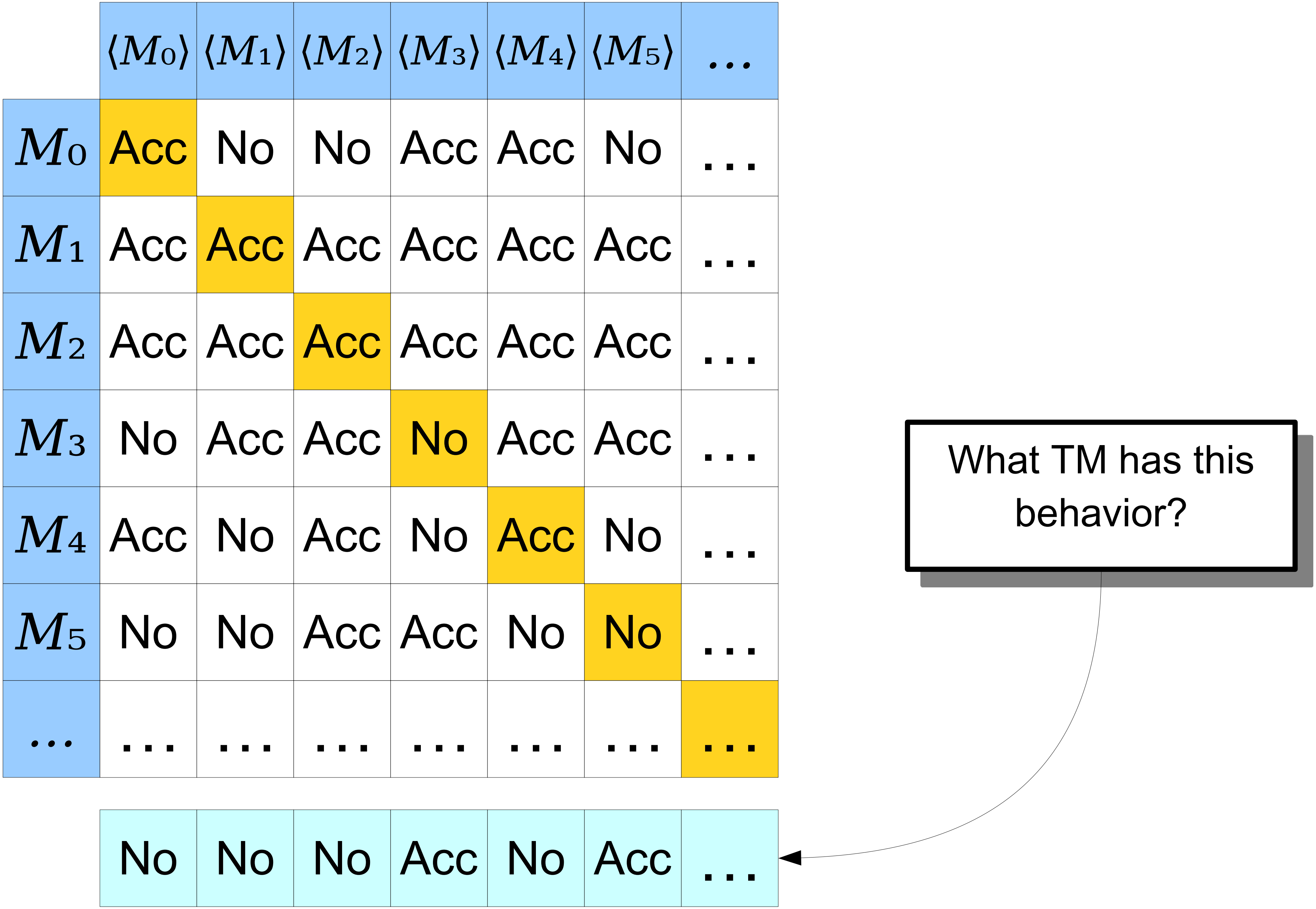
No	No	No	Acc	No	Acc	…

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	…
$\dots$	…	…	…	…	…	…	…

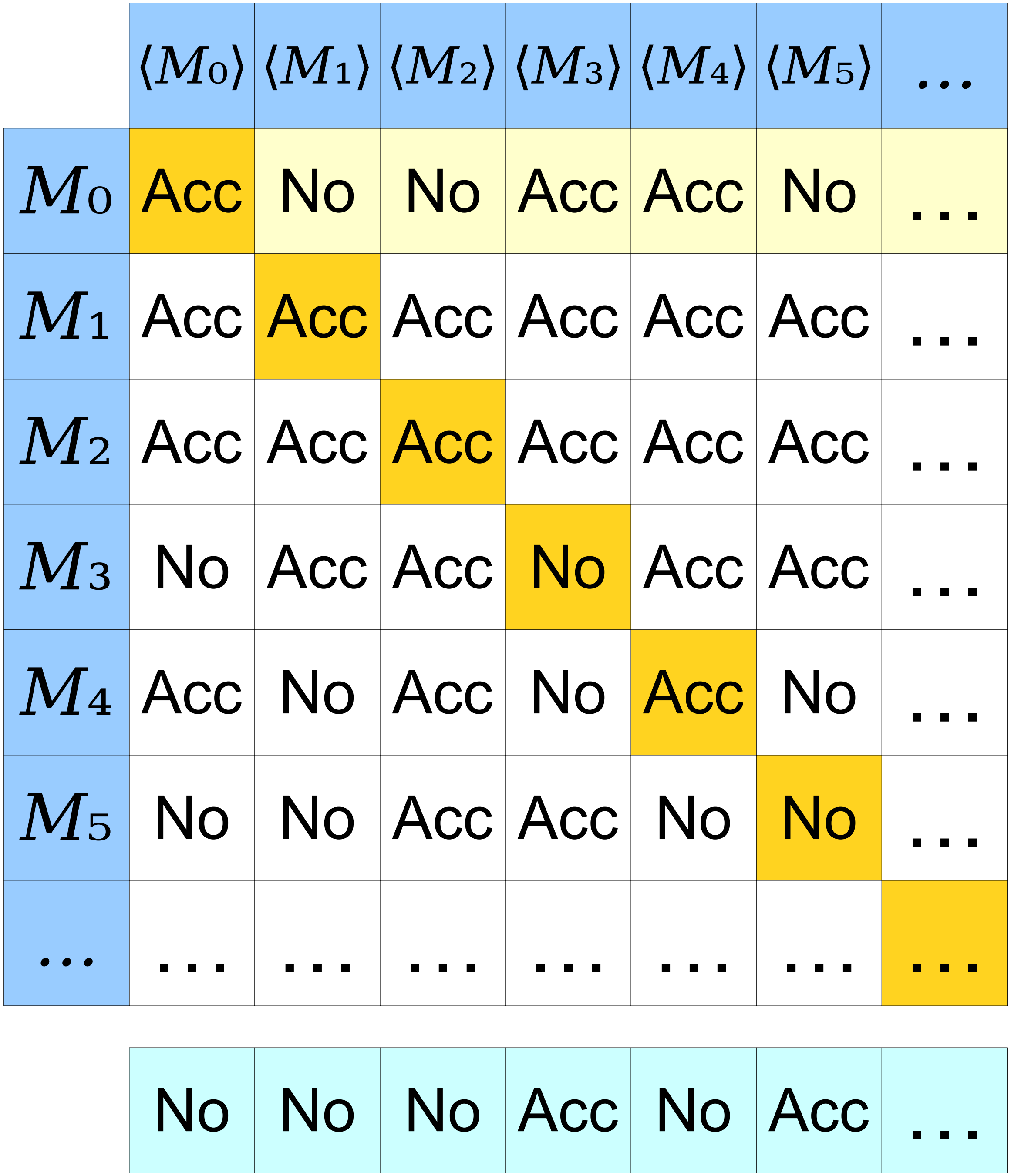
No	No	No	Acc	No	Acc	…



	$\langle M_0\rangle$	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	$\langle M_5\rangle$	…
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	…

What TM has this behavior?

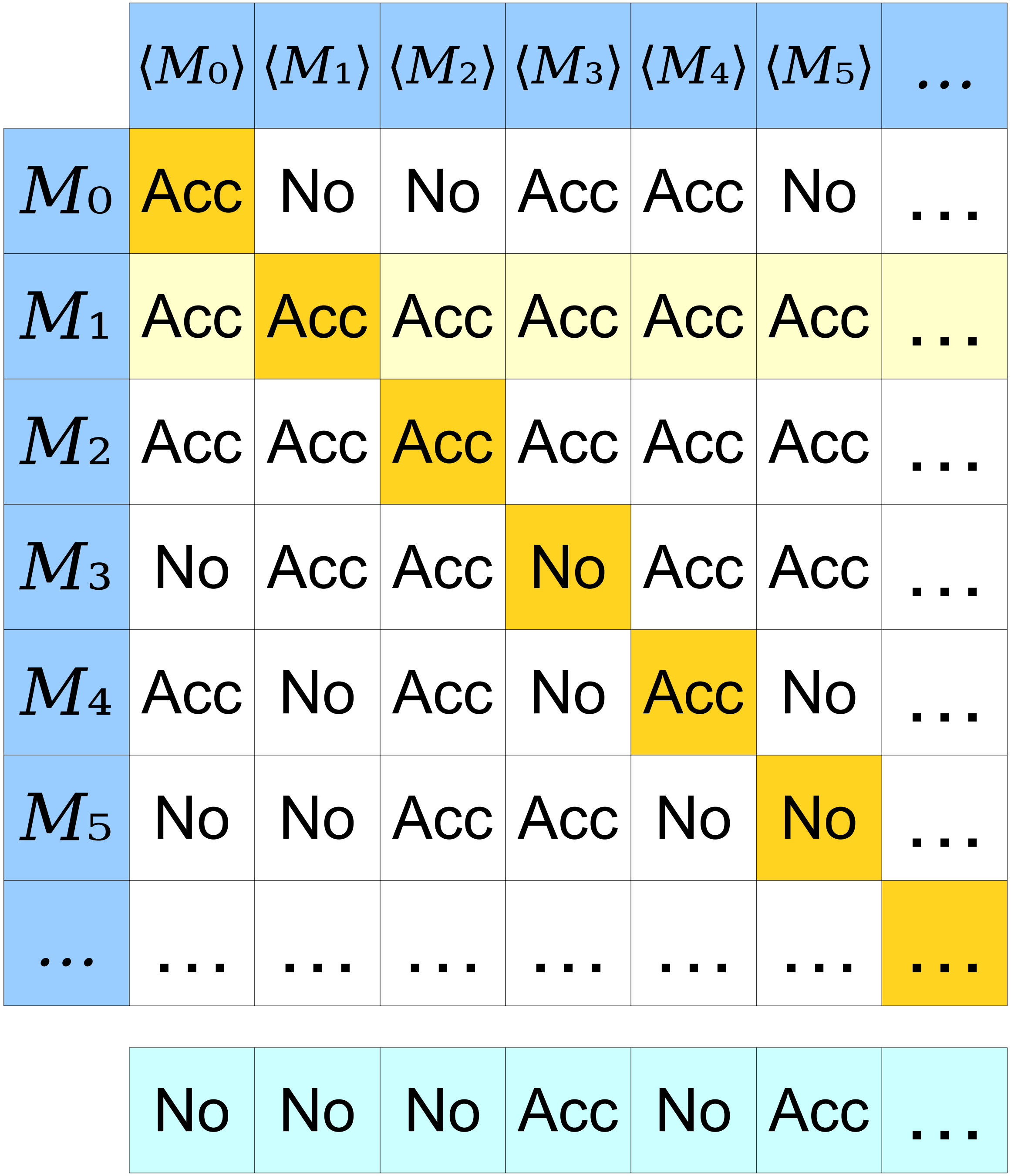


	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…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

No	No	No	Acc	No	Acc	…

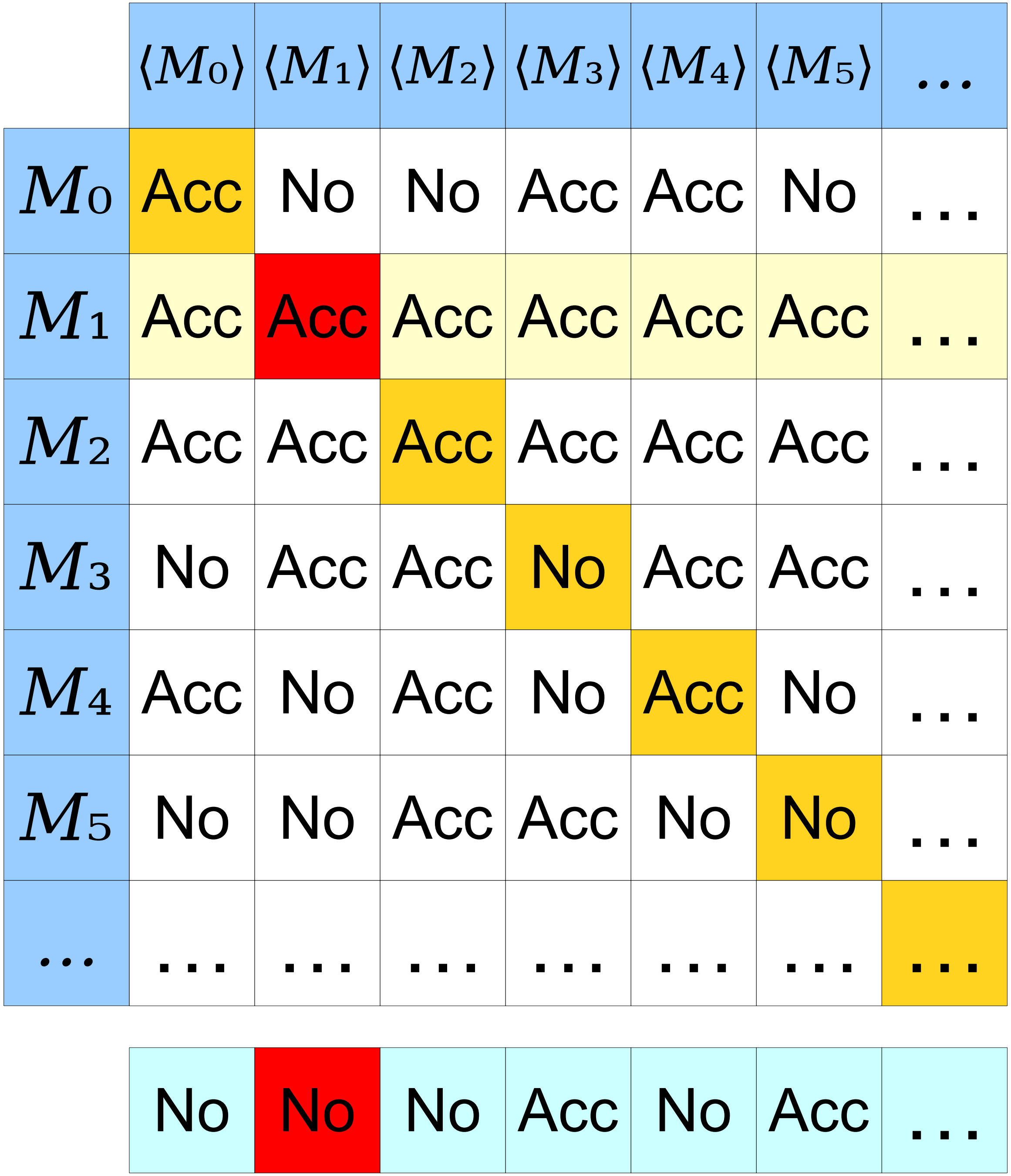
	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…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

No	No	No	Acc	No	Acc	…



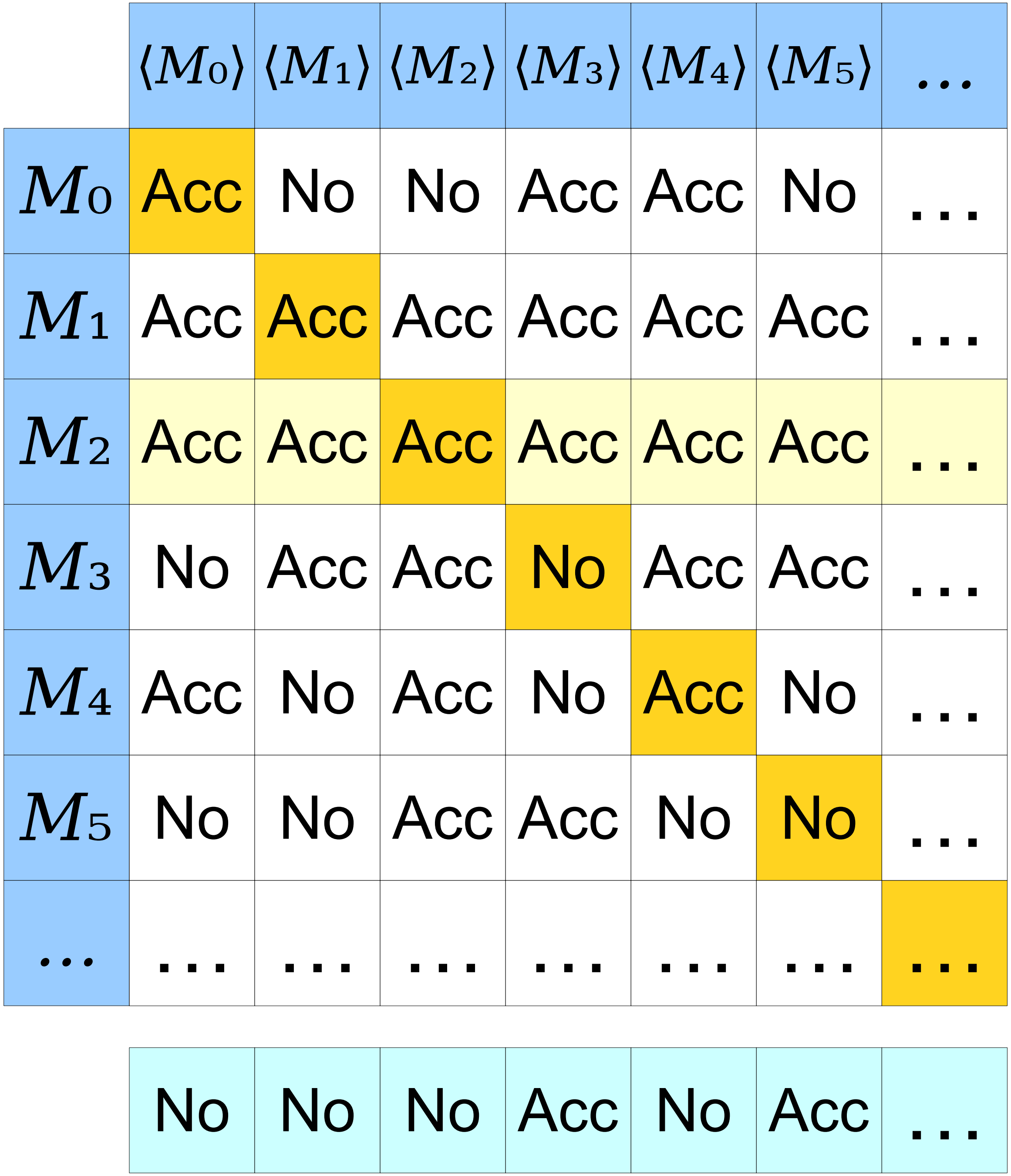
	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…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

No	No	No	Acc	No	Acc	…



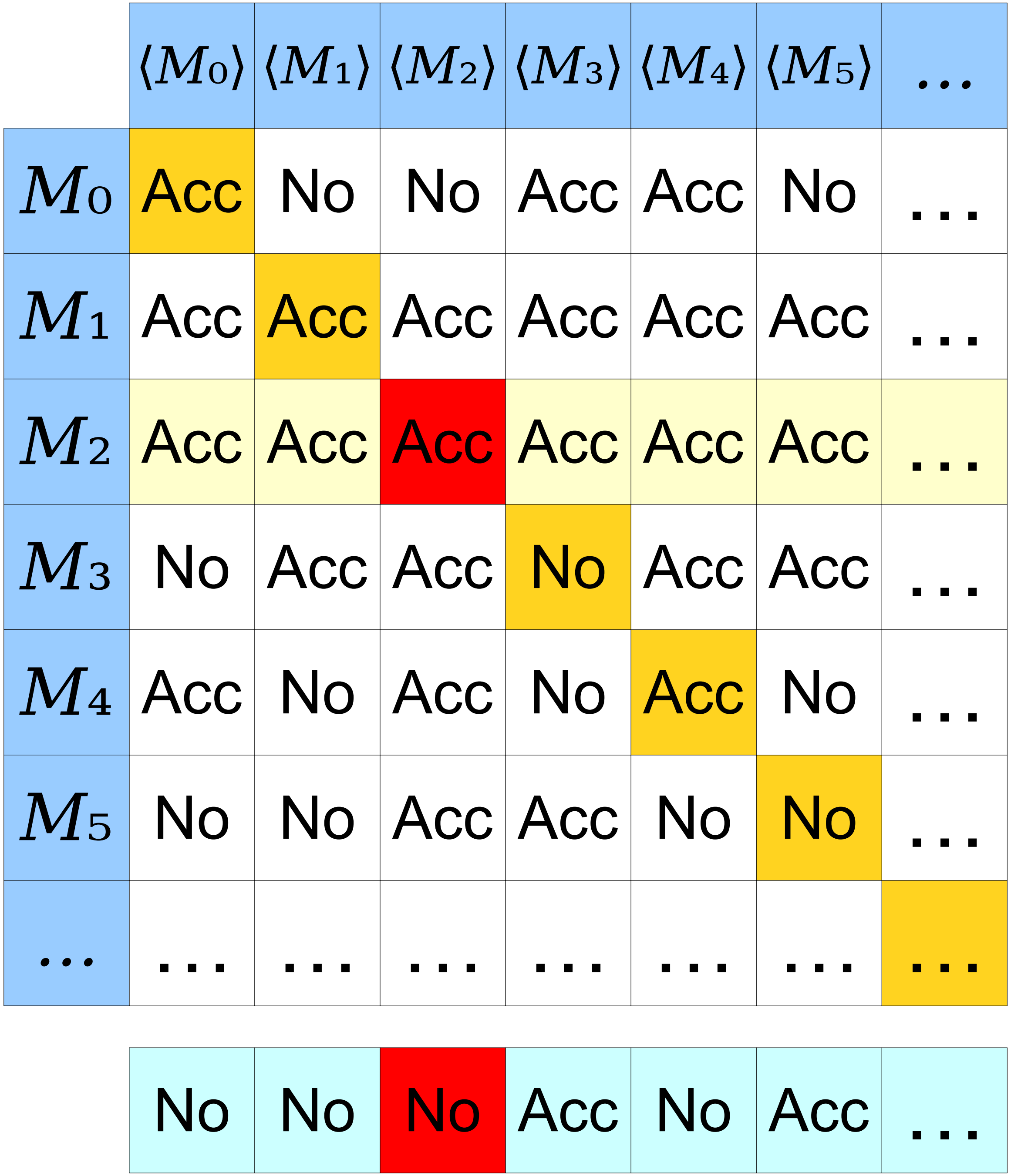
	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…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

No	No	No	Acc	No	Acc	…



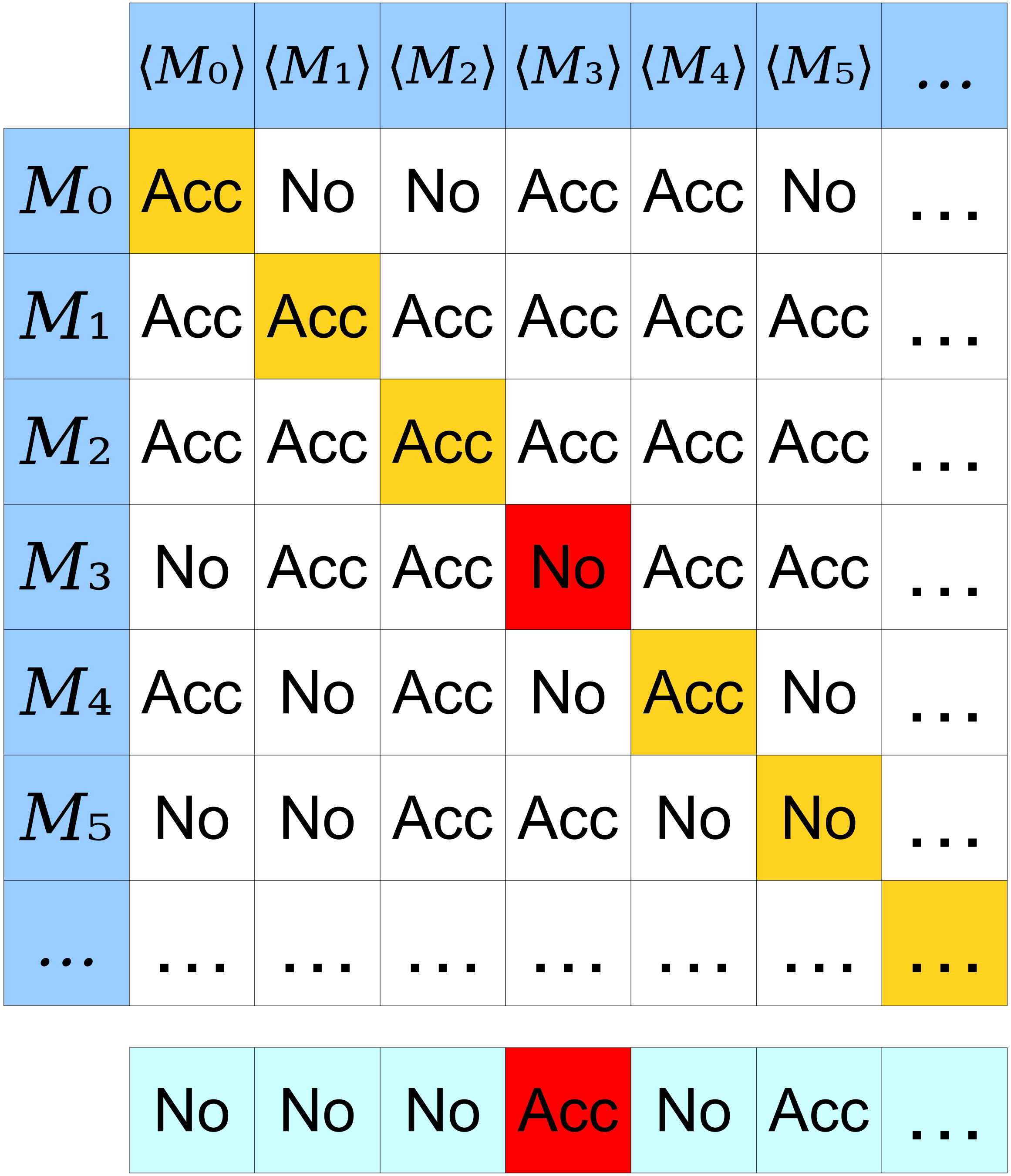
	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…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

No	No	No	Acc	No	Acc	…



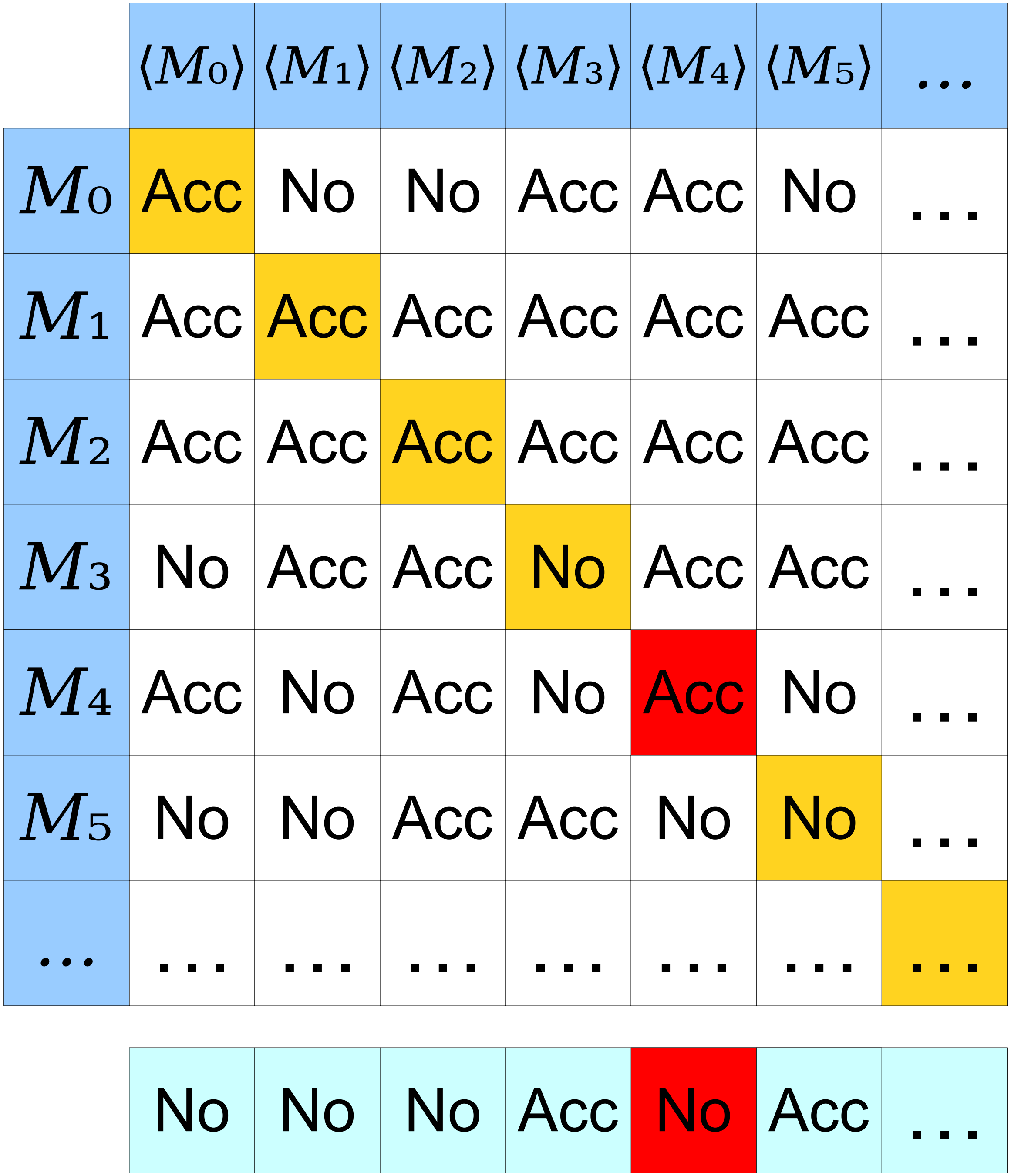
	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
M_0	Acc	No	No	Acc	Acc	No	…
M_1	Acc	Acc	Acc	Acc	Acc	Acc	…
M_2	Acc	Acc	Acc	Acc	Acc	Acc	…
M_3	No	Acc	Acc	No	Acc	Acc	…
M_4	Acc	No	Acc	No	Acc	No	…
M_5	No	No	Acc	Acc	No	No	…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

No	No	No	Acc	No	Acc	…



	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
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	…



	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
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	…
…	…	…	…	…	…	…	…

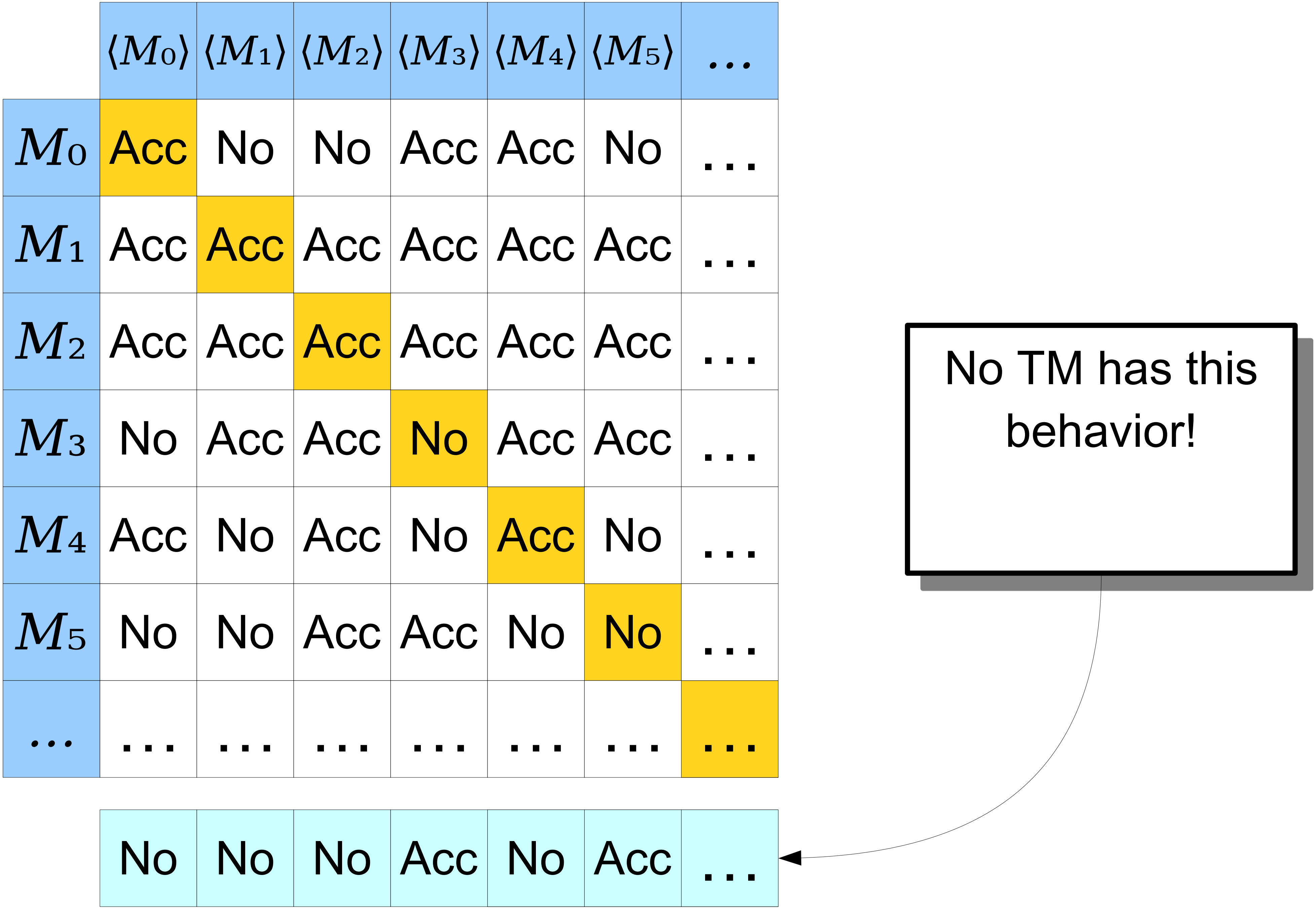
No	No	No	Acc	No	Acc	…

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
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	…

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
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	…



	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
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!

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
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	…

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	...
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	...

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
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 descriptions.”

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	…
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	…

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

Diagonalization Revisited

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

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

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

L_D = { ⟨M⟩ | M is a TM and M does not accept ⟨M⟩ }

Theorem: $L_D \notin$ **RE**.

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

Now, focus on what happens if we run recognizer R on its own string encoding (that is, running R on ⟨R⟩). Since R is a recognizer for L_D, we see that

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

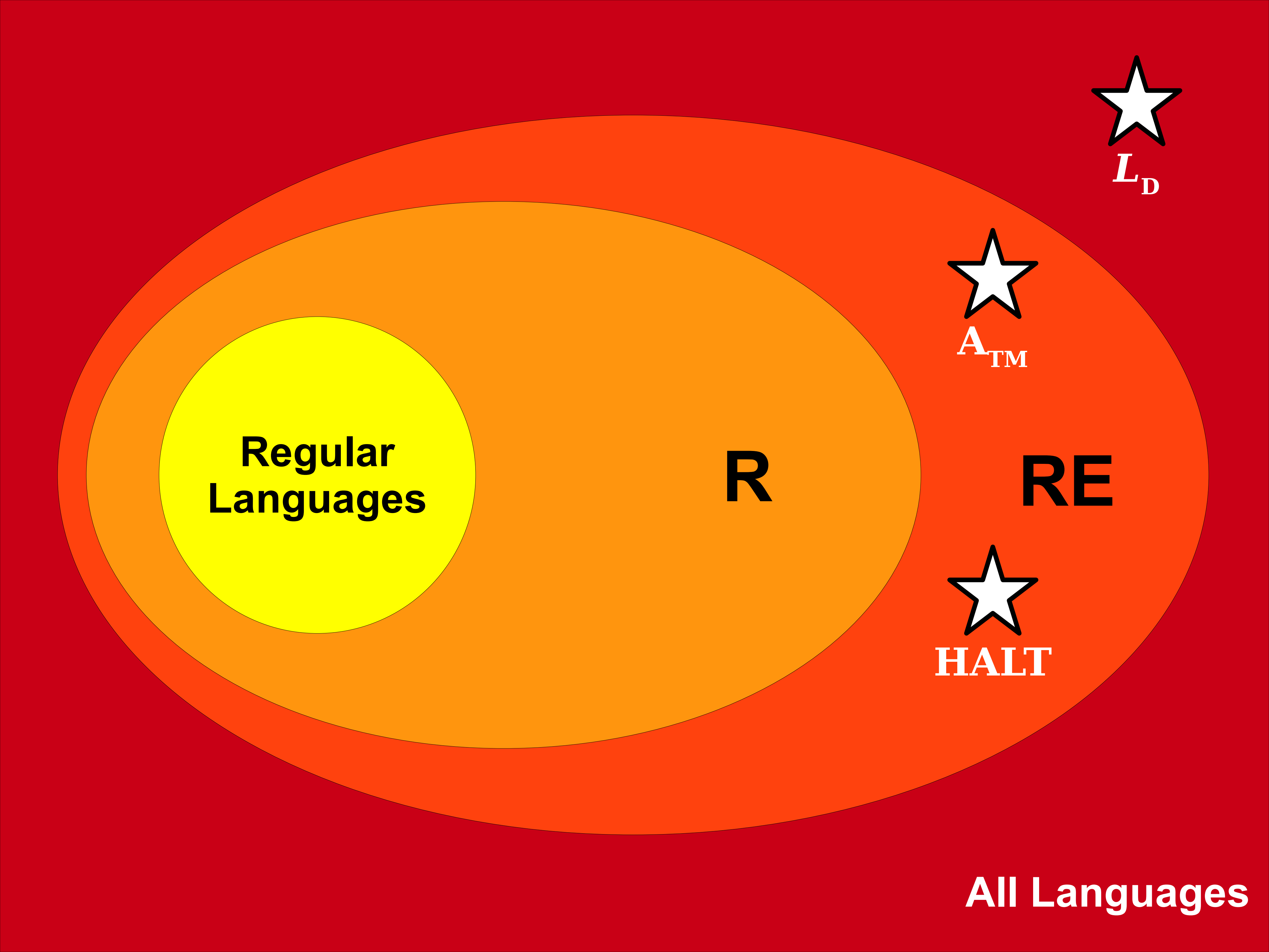
By definition of L_D, we know that

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

Combining the two above statements tells us that

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

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


L_D
A_{TM}
Regular Languages
R
RE
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.

- Intuitively, given any non-**RE** language, there will be some string in the language that *cannot* be proven to be in the language.
- 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.

There are more problems to solve than there are programs capable of solving them.

There is so much more to explore and so many big questions to ask – ***many of which haven't been asked yet!***

Our questions to you:

What problems will you ***choose*** to solve?
Why do those problems matter to you?
And how are you going to solve them?