Turing Machines

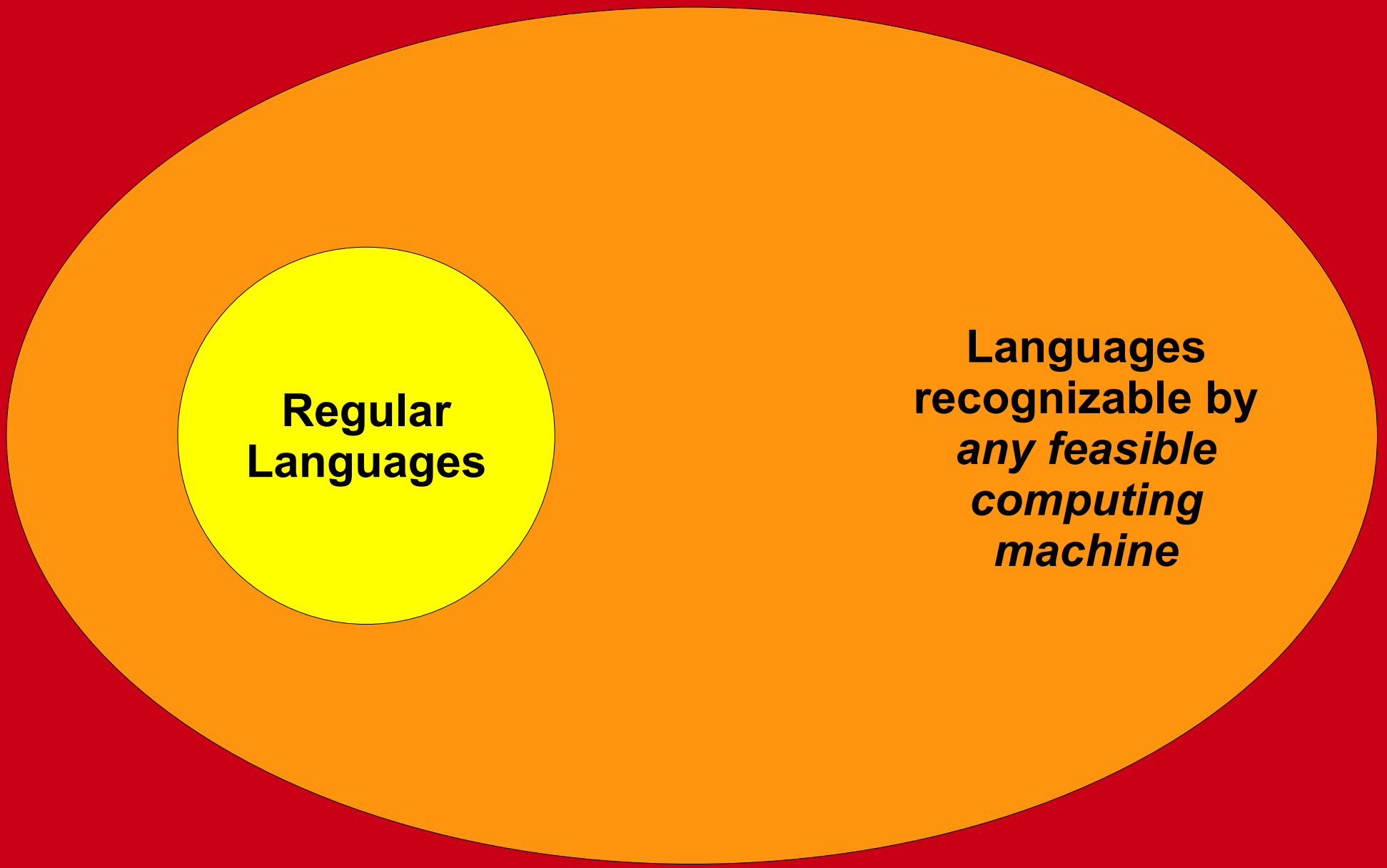
Part One

What problems can we solve with a computer?

A Brief History Lesson

Preparing for the Final Exam

- Highly recommend: go over your past assignments and the midterm and assess for any gaps.
- Use the practice finals and the practice problems to assess your understanding of the topics
- Use OHs and Ed to help you on those gaps.


Regular Languages
Languages recognizable by *any feasible computing machine*
All Languages

That same drawing, to scale.

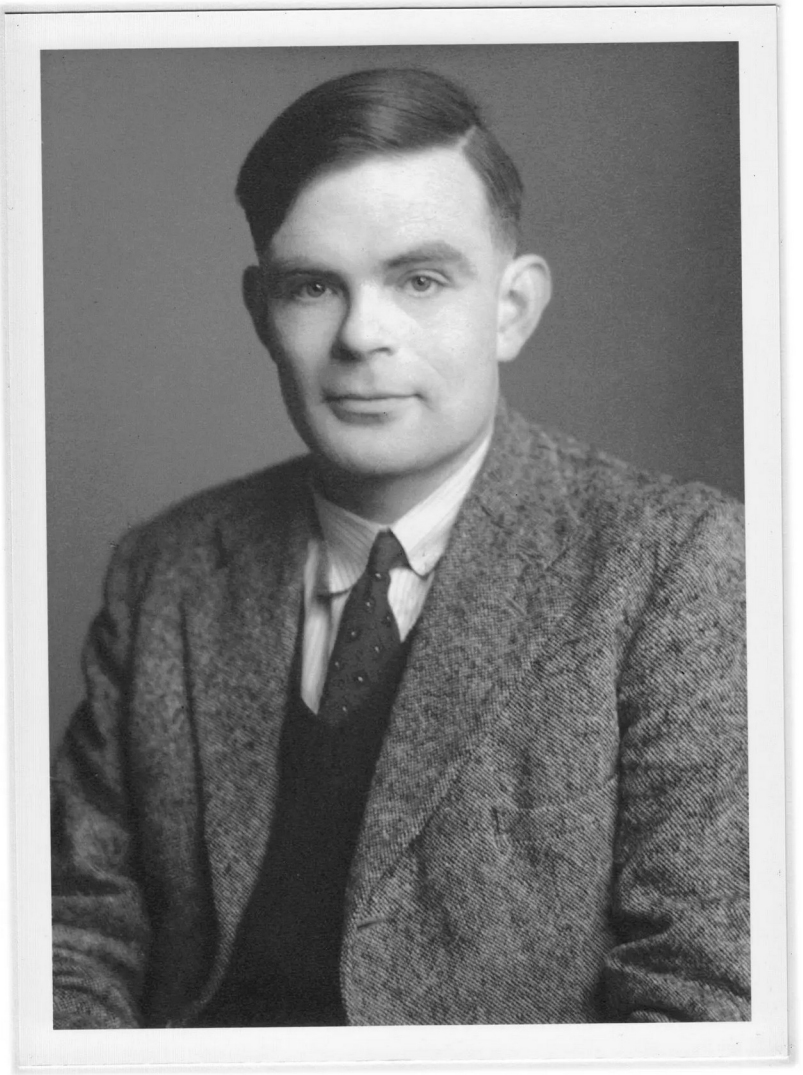
All Languages

The Problem

- Finite automata accept precisely the regular languages.
- We've already seen cases where we may need unbounded memory to recognize languages.
  - e.g. $\{ \mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N} \}$ requires unbounded counting.
- How do we model a computing device that has unbounded memory?

Turing Machines

- In March 1936, Alan Turing (aged 23!) published a paper detailing the ***a-machine*** (for ***automatic machine***), an automaton for computing on real numbers.
- They're now more popularly referred to as ***Turing machines*** in his honor.
- He also later made contributions to computational biology, artificial intelligence, cryptography, etc. Seriously, Google this guy.



https://static01.nyt.com/images/2019/06/10/obituaries/10overlooked-alanturing-1/f86fc26c86b444c391fb412bb3dcec76-superJumbo.jpg?

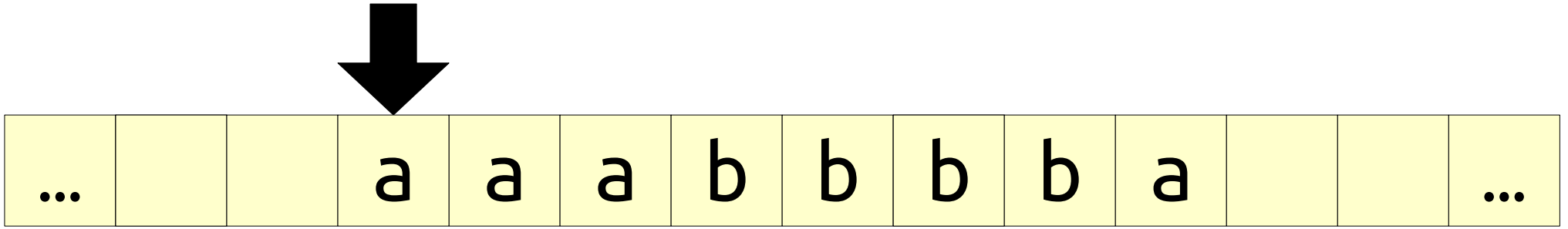
Our Next Challenge

- Let's now take aim at this more general language:

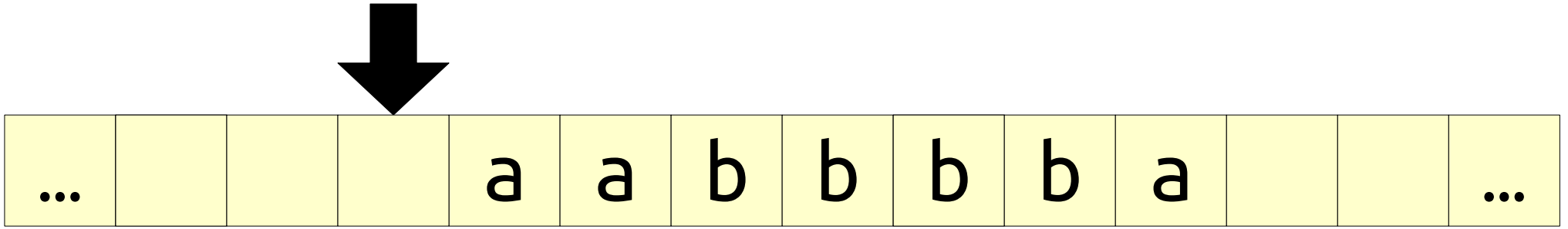
  $\{ w \in \{\texttt{a}, \texttt{b}\}^* \mid w$ has an equal number of `a`'s and `b`'s $\}$

- This language is not regular (do you see why?)
- Let's see how to design a TM for it.

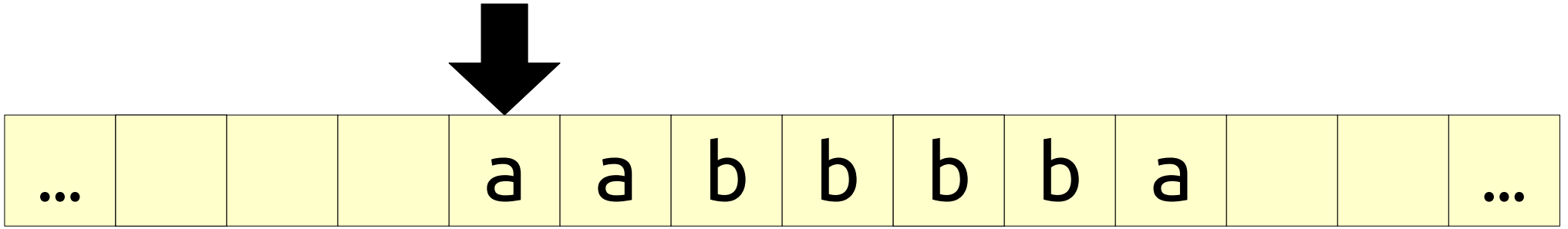
A Caveat



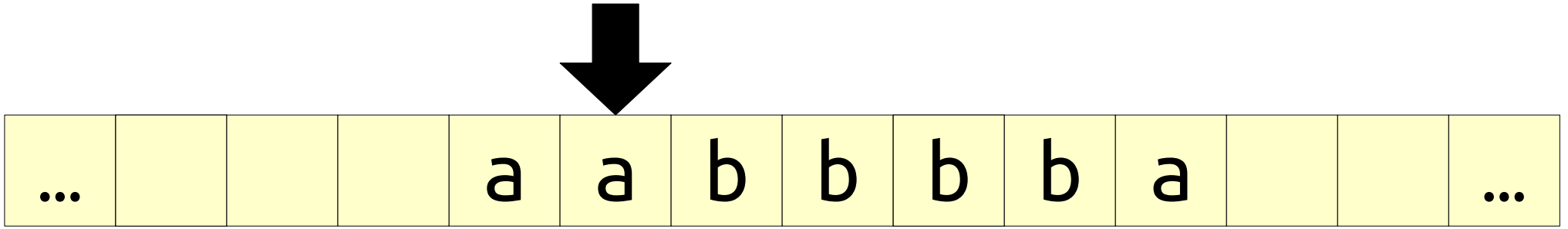
A Caveat



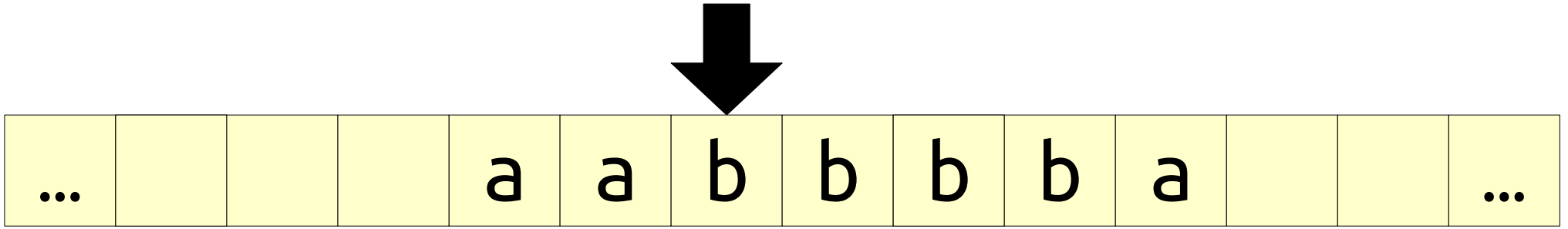
A Caveat



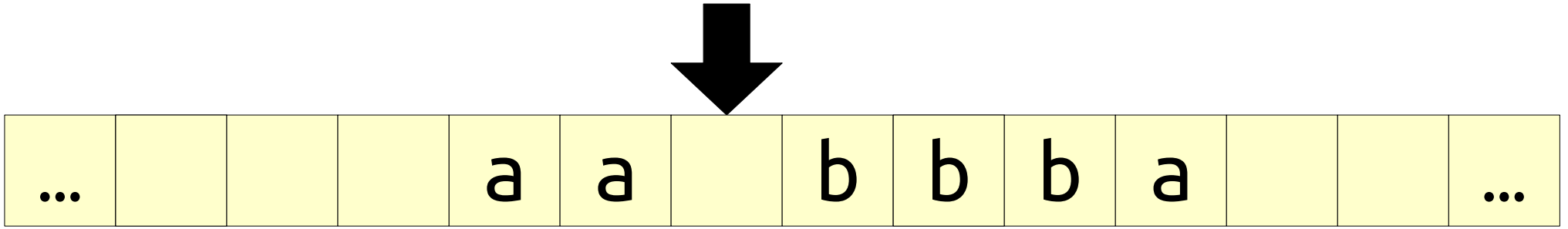
A Caveat



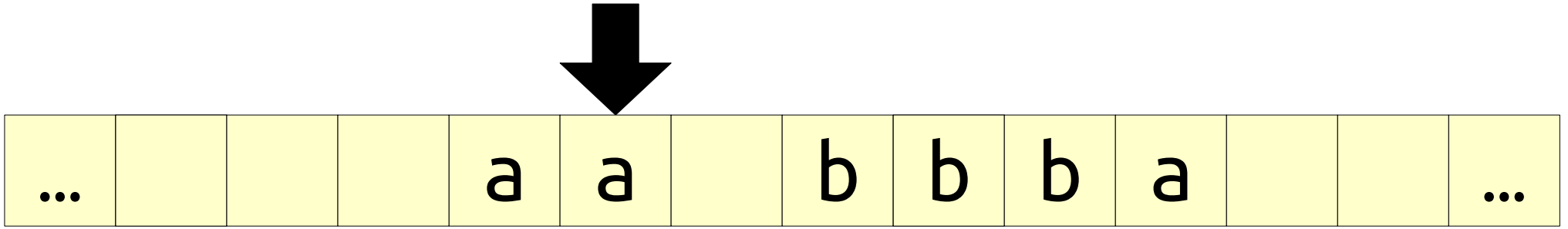
A Caveat



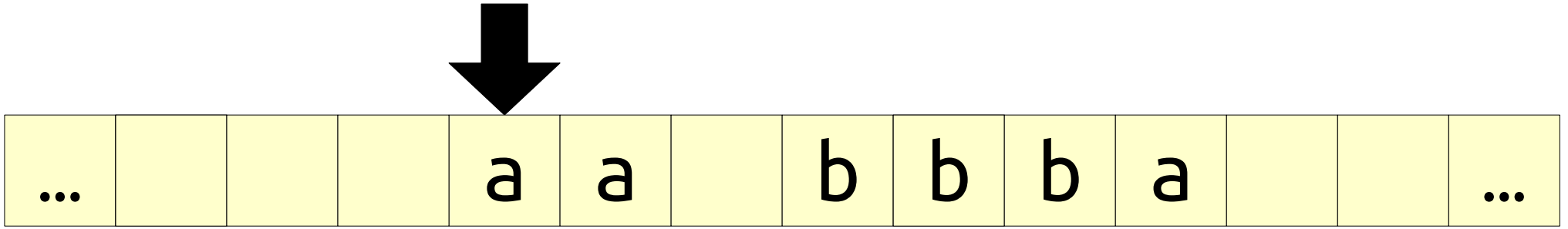
A Caveat



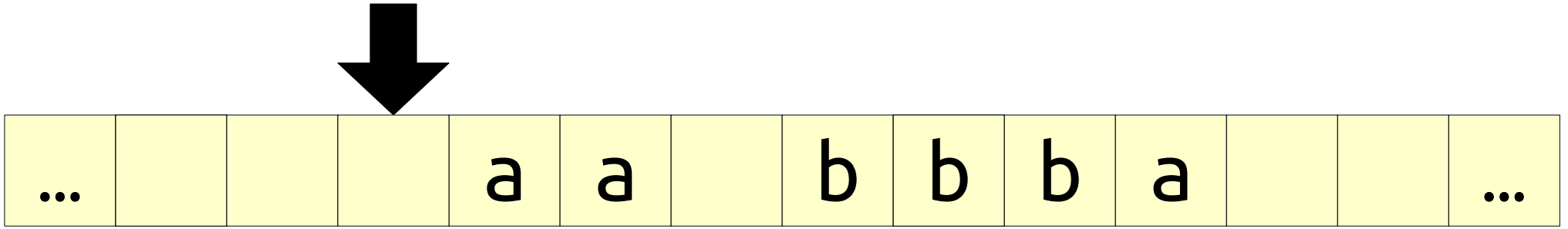
A Caveat



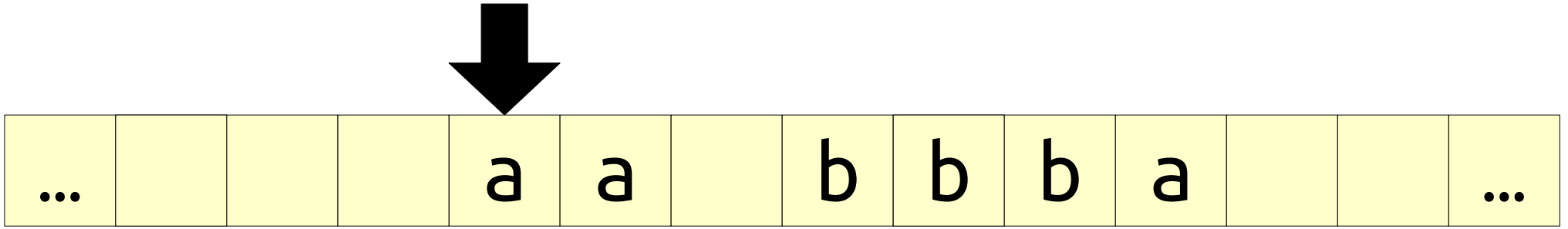
A Caveat



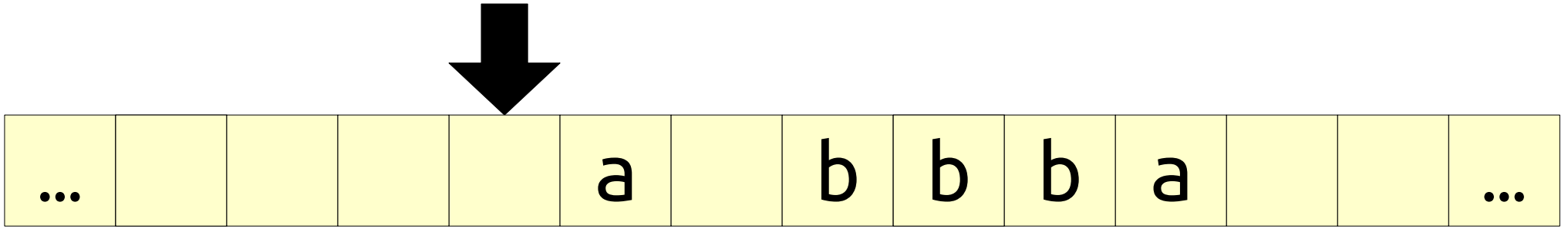
A Caveat



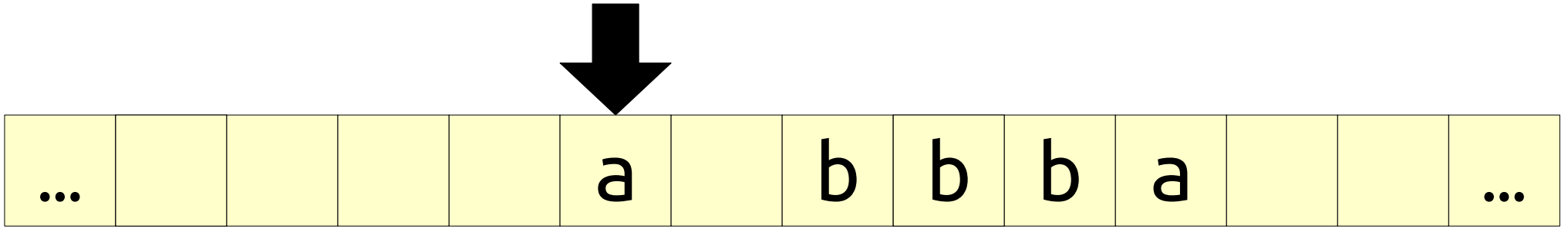
A Caveat



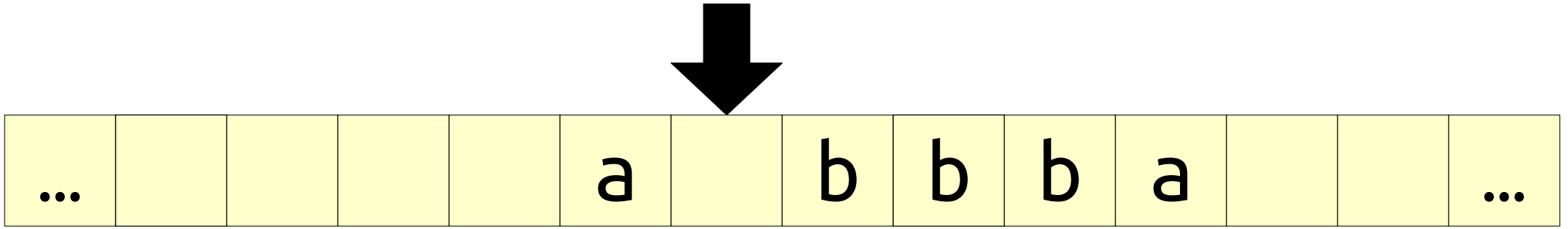
A Caveat



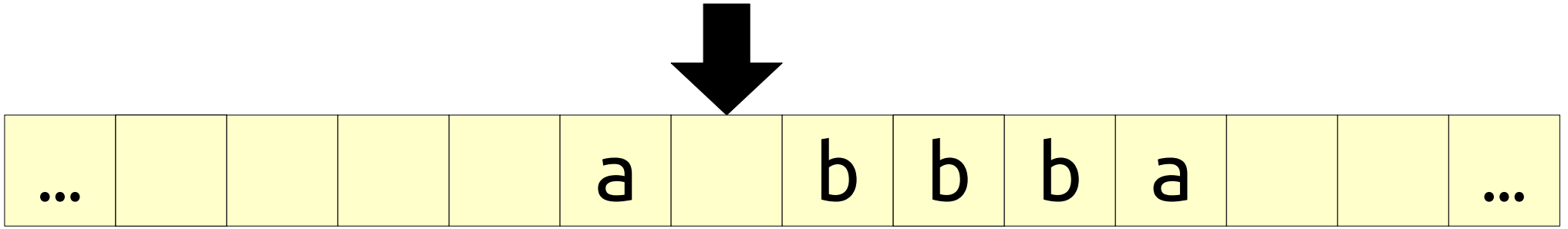
A Caveat



A Caveat

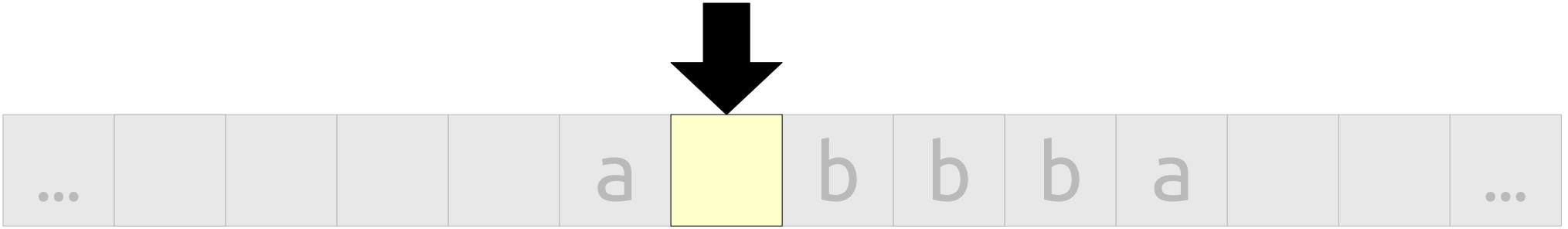


A Caveat



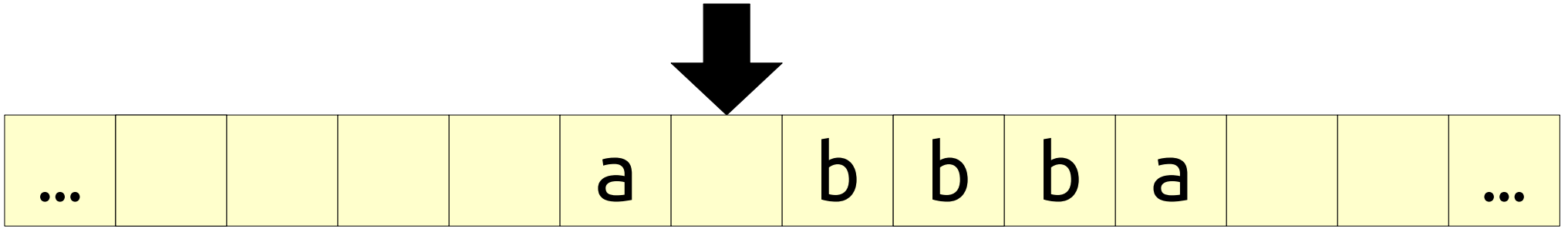
How do we know that this blank isn't one of the infinitely many blanks after our input string?

A Caveat



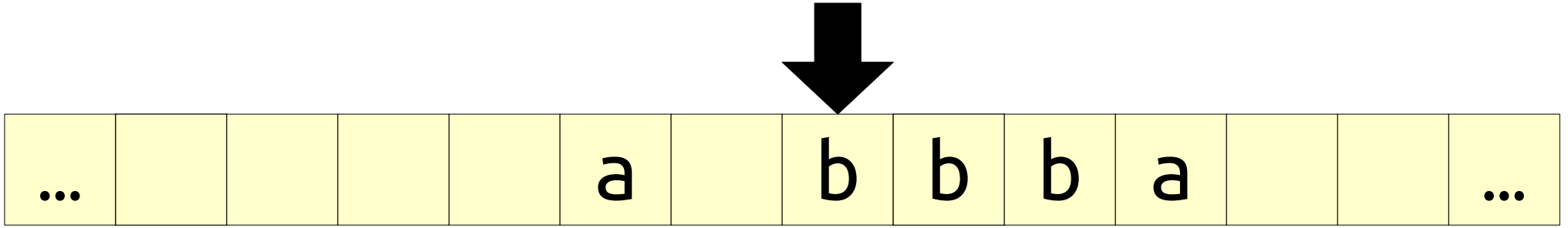
How do we know that this blank isn't one of the infinitely many blanks after our input string?

A Caveat

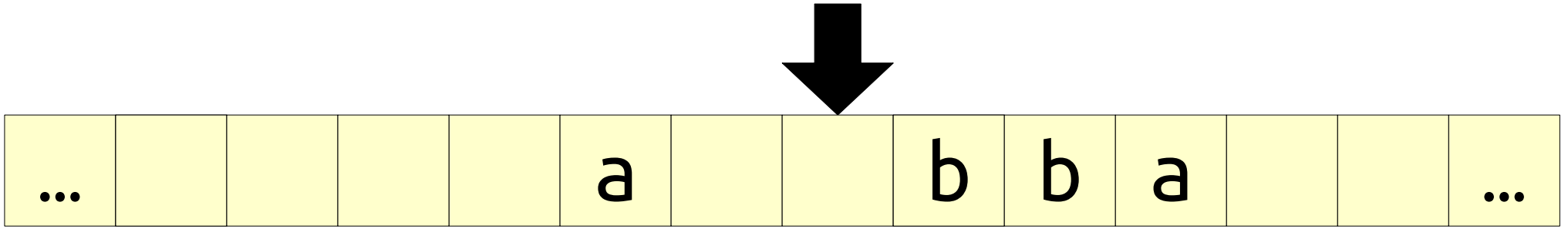


How do we know that this blank isn't one of the infinitely many blanks after our input string?

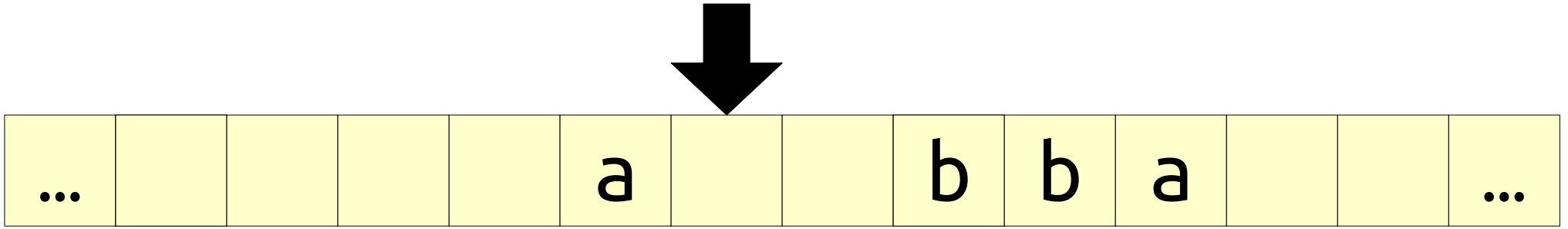
A Caveat



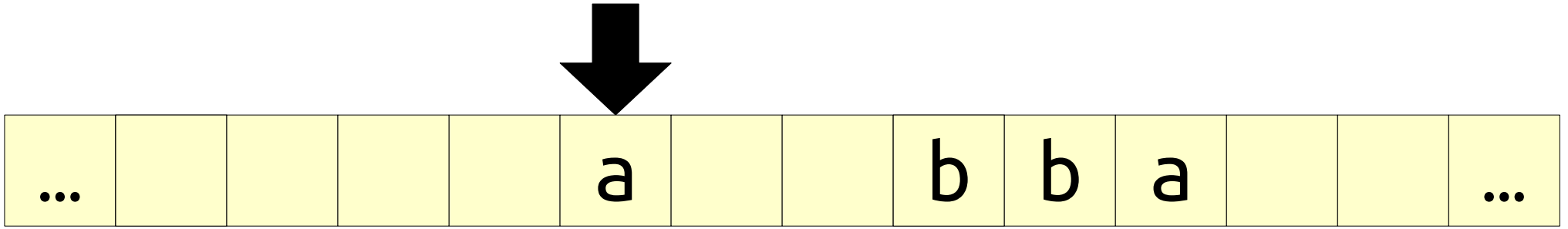
A Caveat



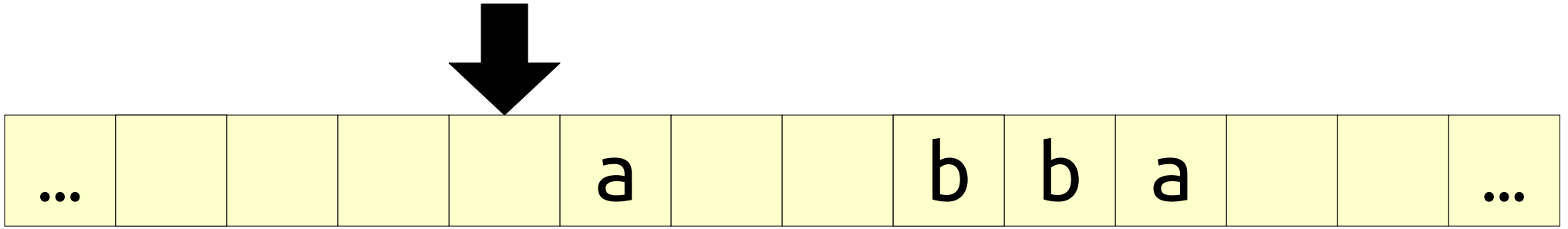
A Caveat



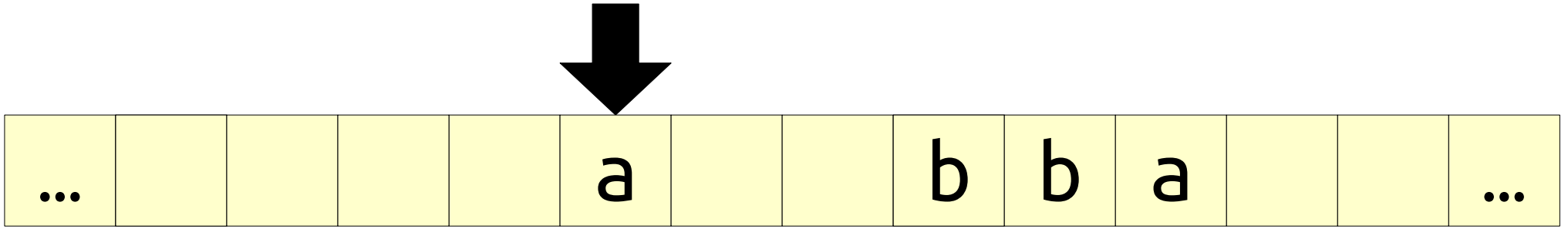
A Caveat



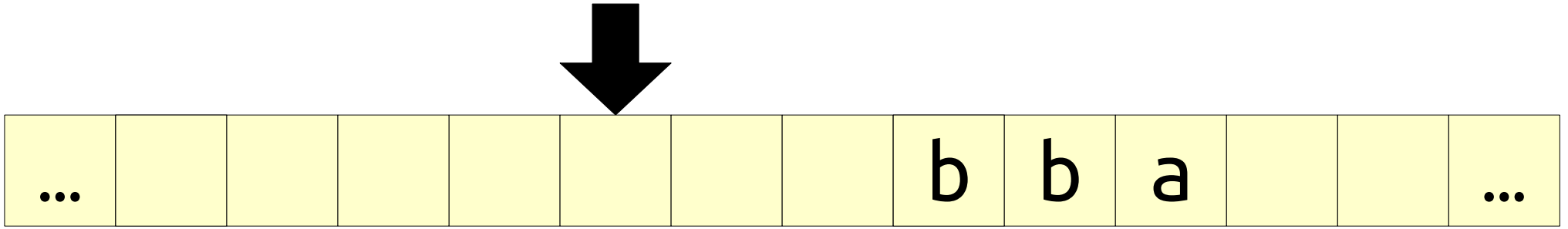
A Caveat



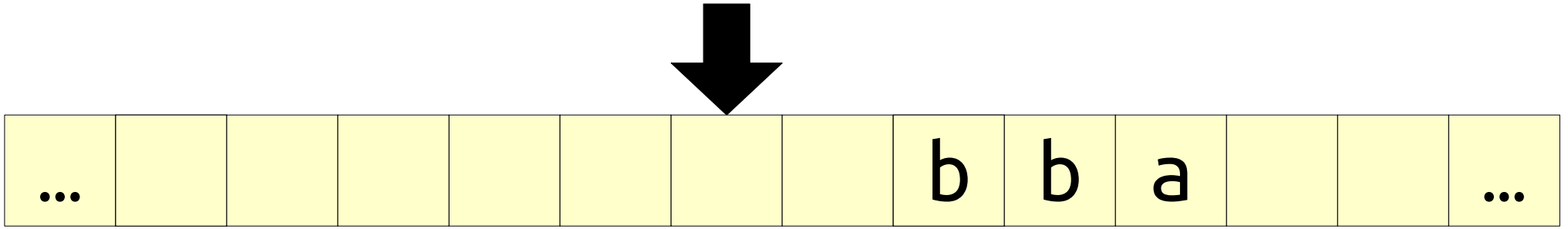
A Caveat



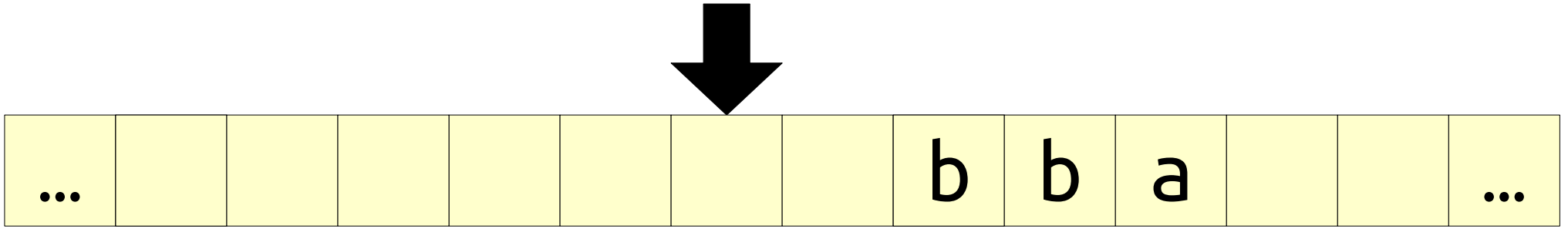
A Caveat



A Caveat

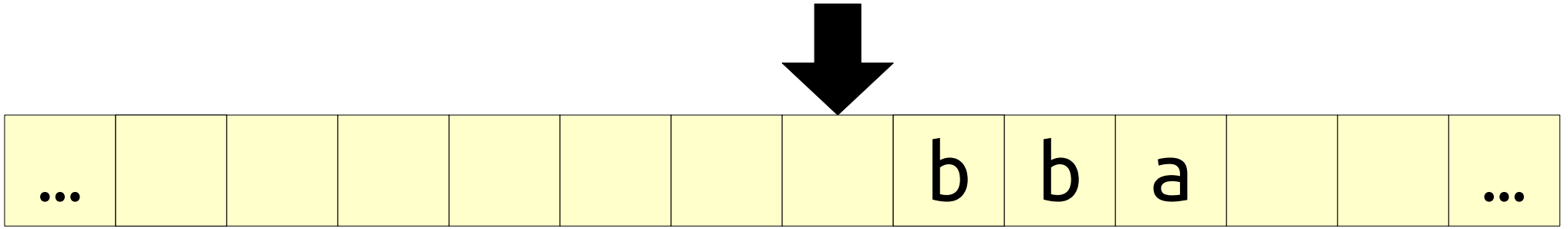


A Caveat

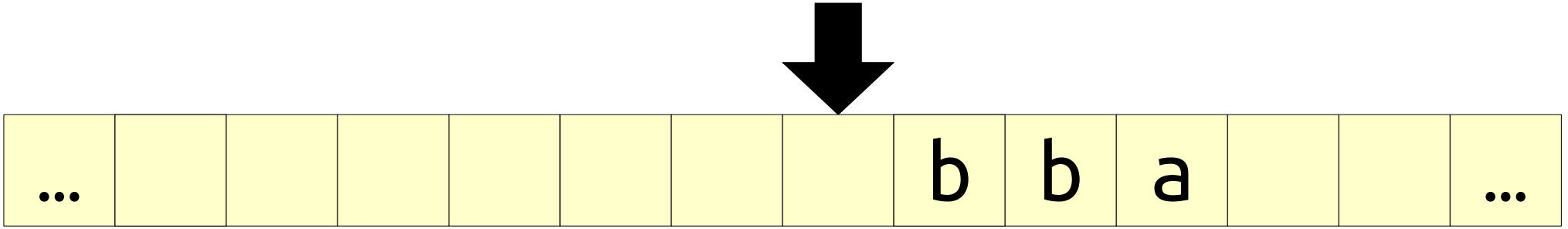


How do we know that this blank isn't one of the infinitely many blanks after our input string?

A Caveat

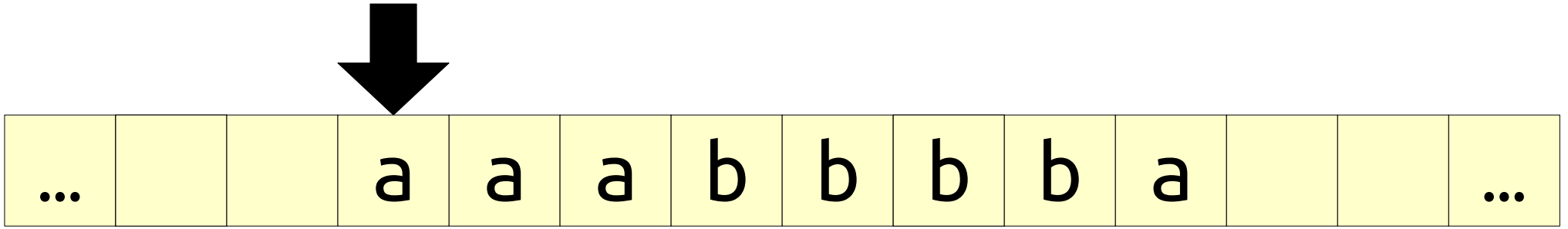


A Caveat

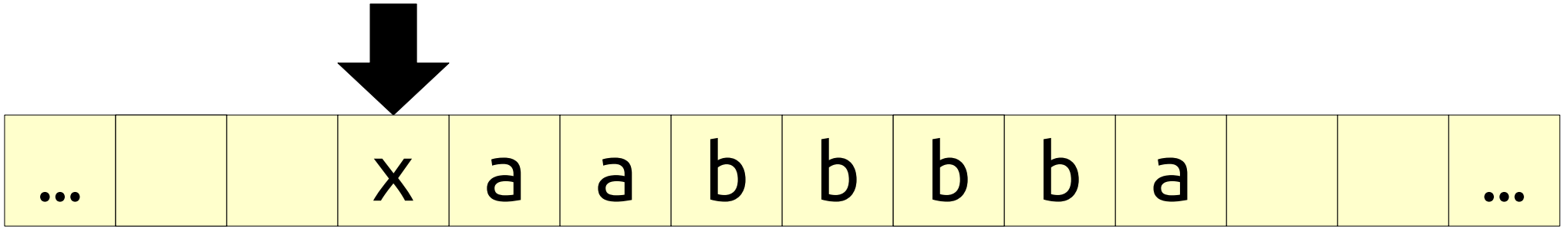


How do we know that this blank isn't one of the infinitely many blanks after our input string?

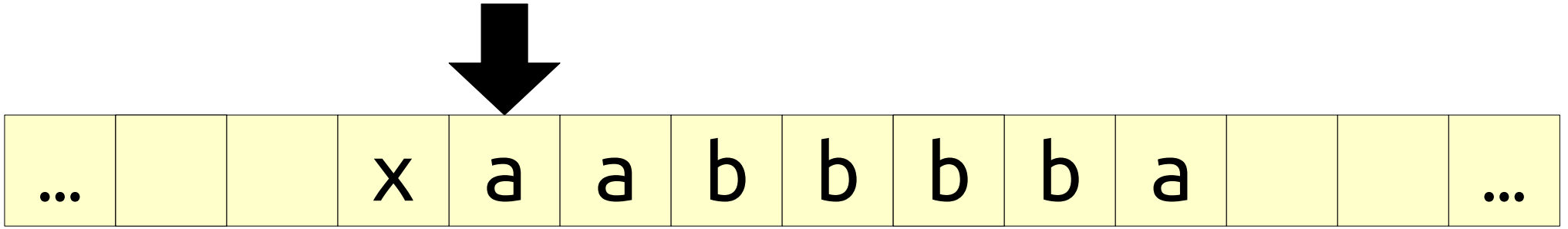
One Solution



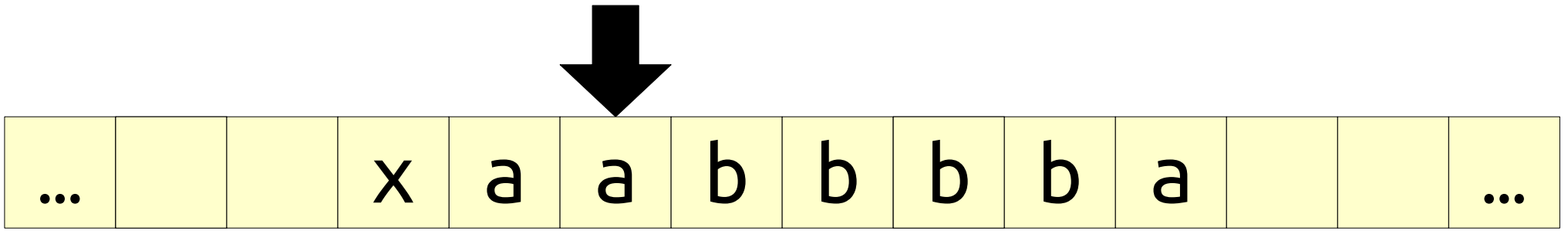
One Solution



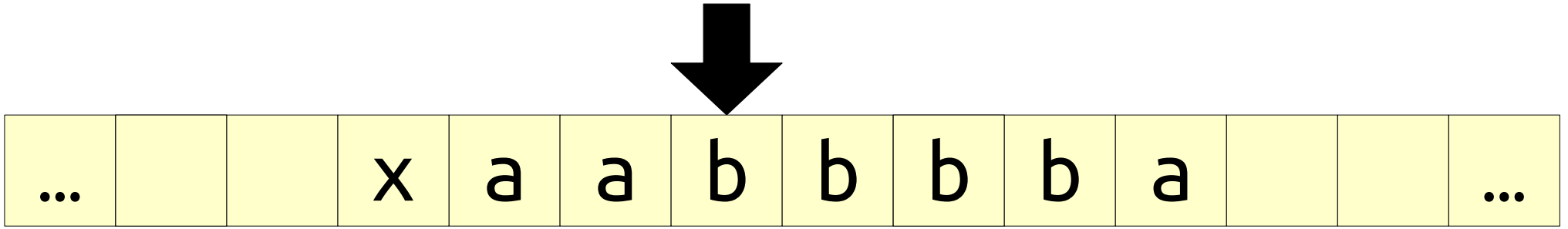
One Solution



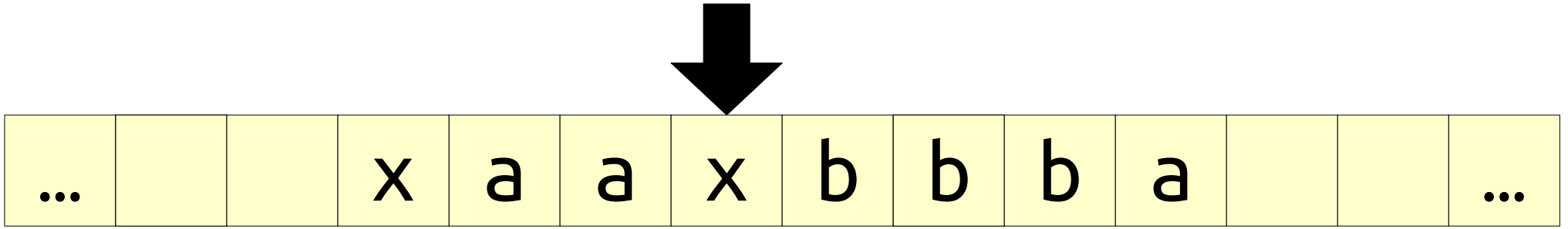
One Solution



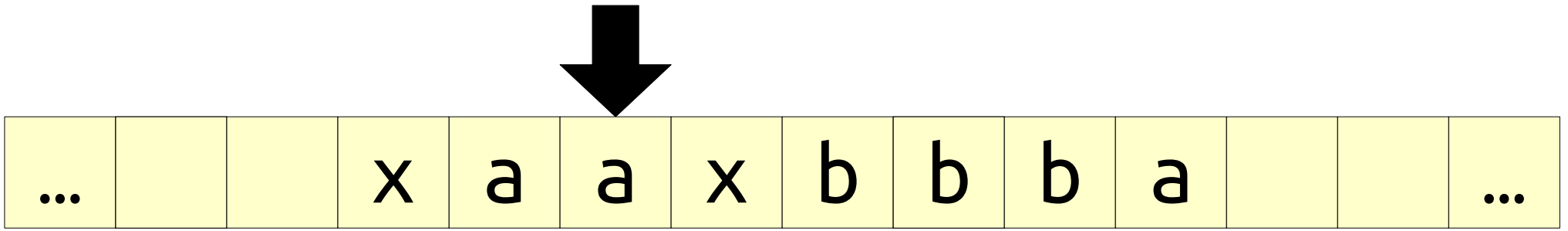
One Solution



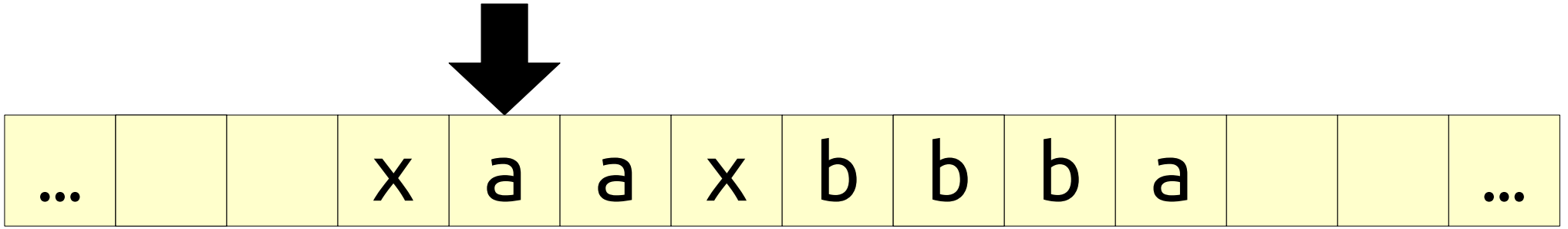
One Solution



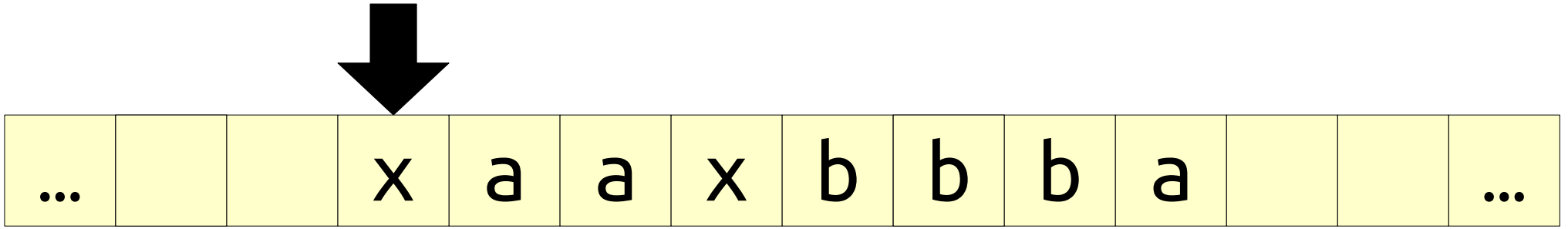
One Solution



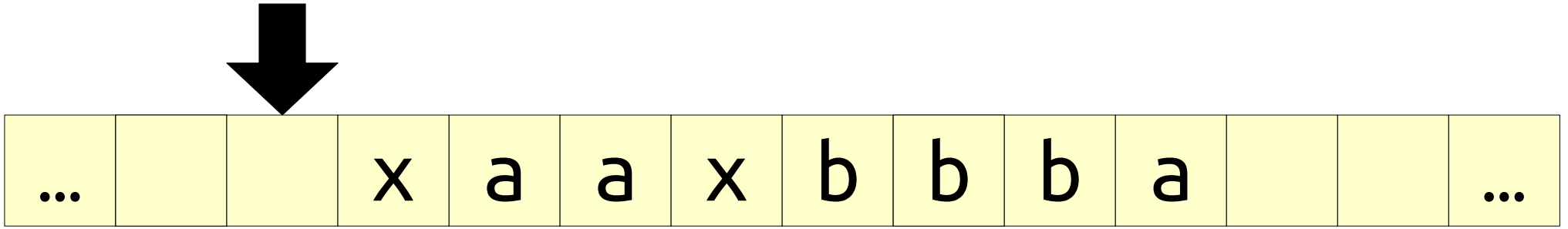
One Solution



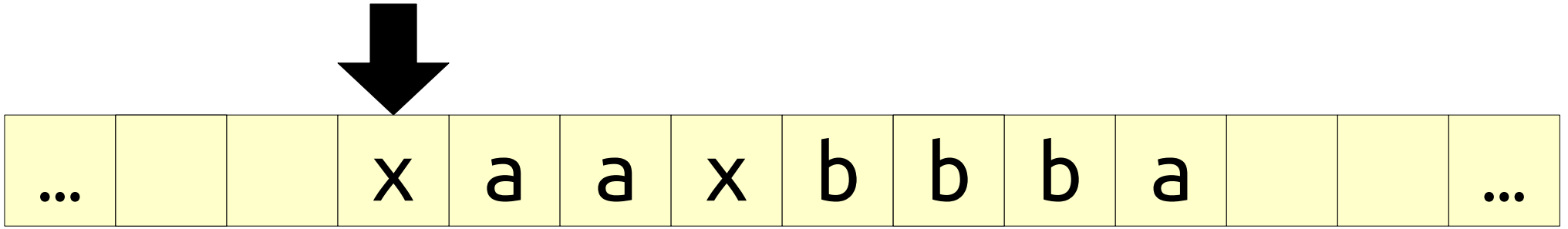
One Solution



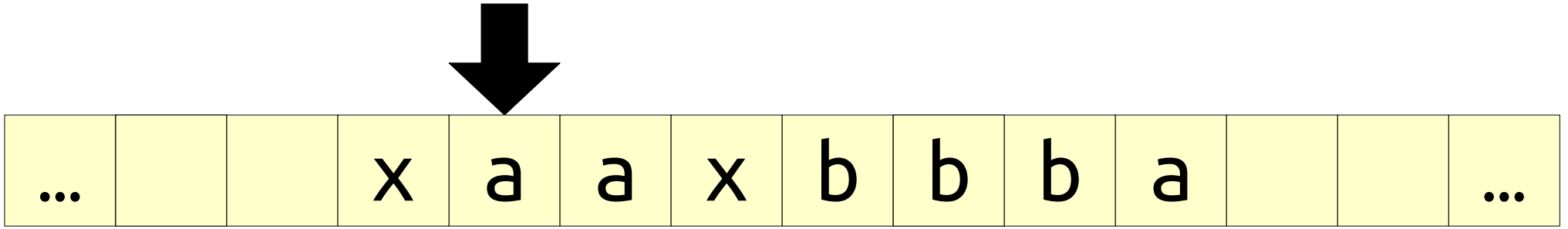
One Solution



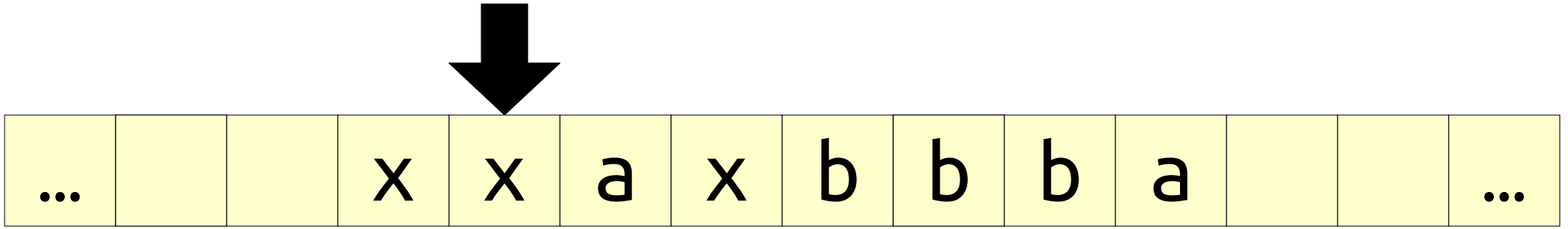
One Solution



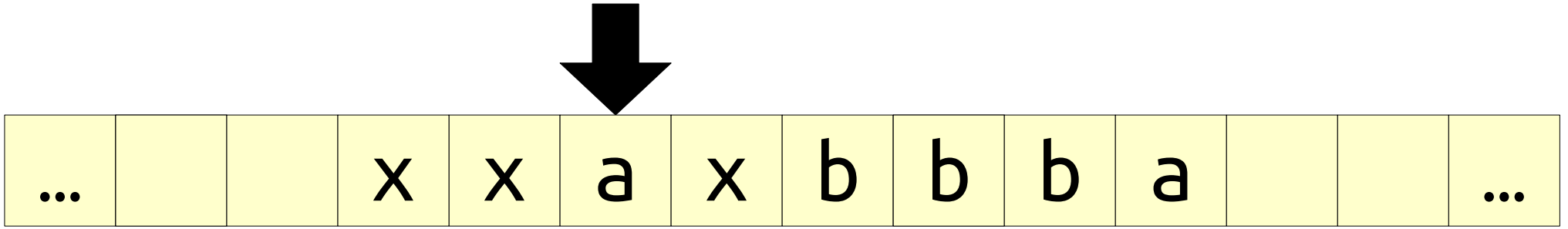
One Solution



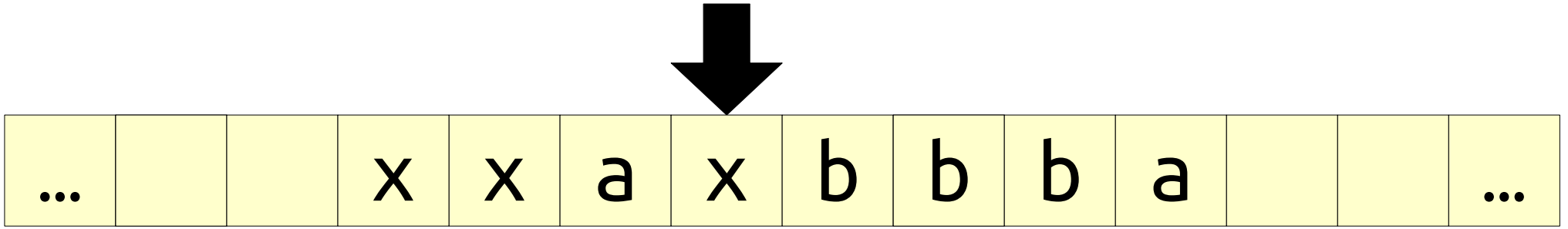
One Solution



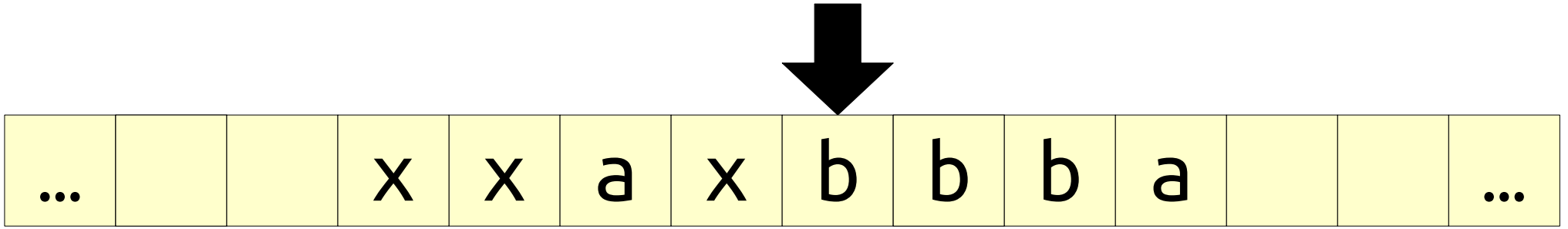
One Solution



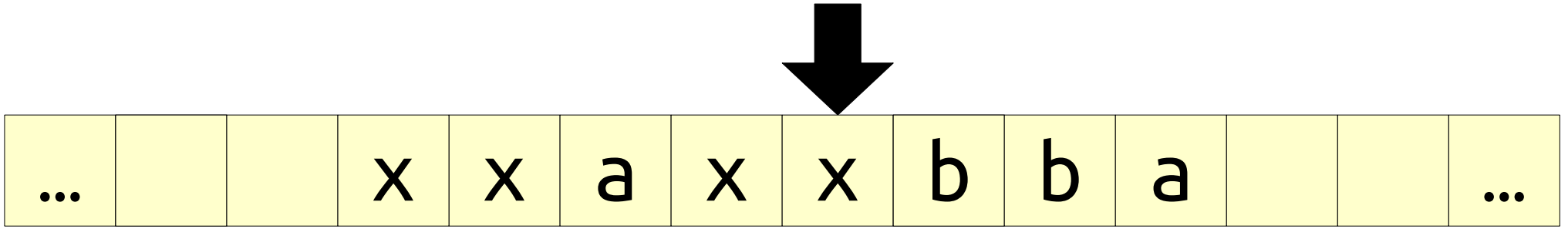
One Solution



One Solution



One Solution



Let’s take a look at this in action!

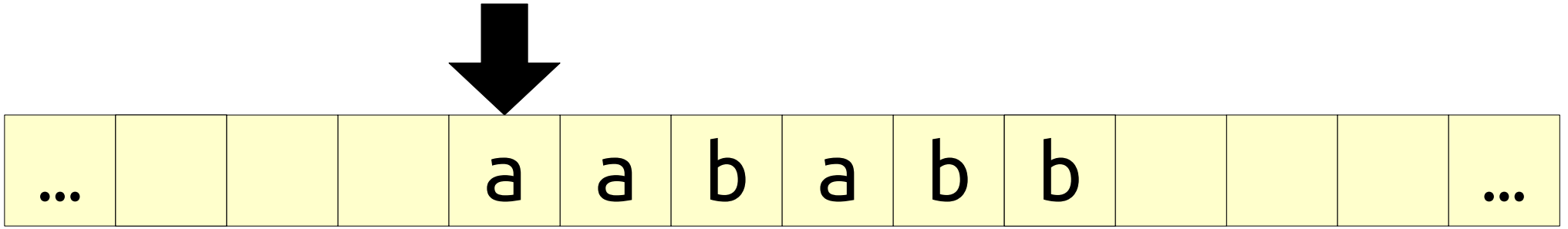
Another Idea

- We just built a TM for the language

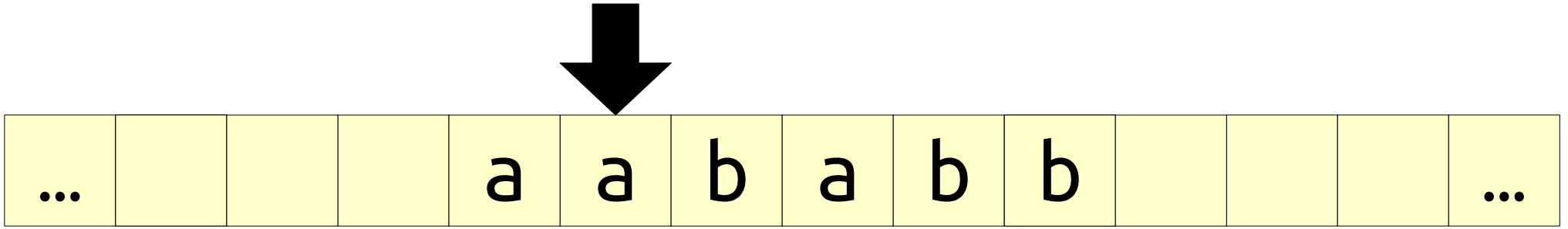
  $$\{ w \in \{\mathtt{a}, \mathtt{b}\}^* \mid w \text{ has the same number of } \mathtt{a}\text{'s and } \mathtt{b}\text{'s } \}.$$

- An observation: this would be a *lot* easier to test for if all the **a**'s came before all the **b**'s.
  - In fact, that would turn this into checking if the string has the form $\mathtt{a}^n\mathtt{b}^n$, which we already know how to do!
- ***Idea:*** Could we sort the characters of our input string?

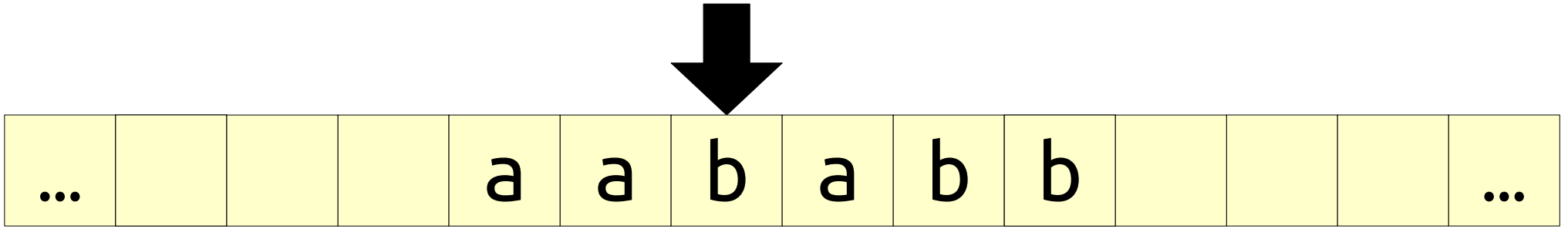
The Idea



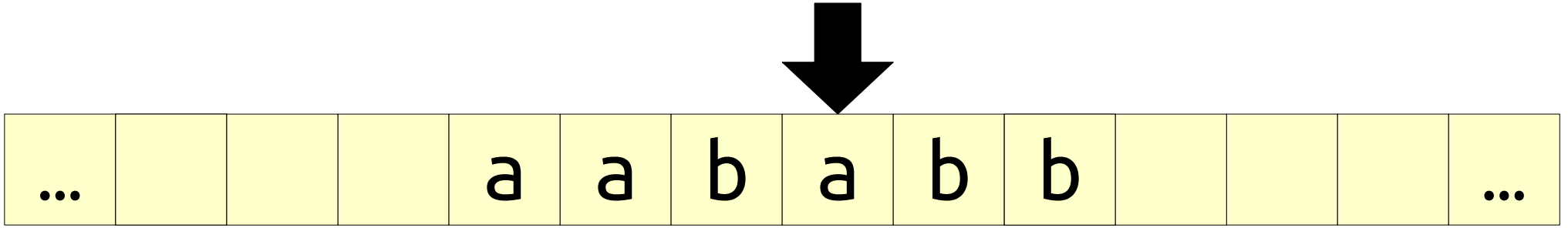
The Idea



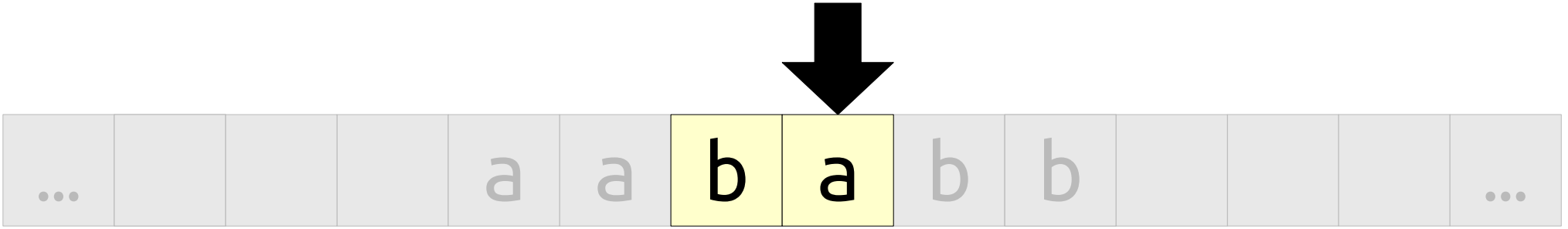
The Idea



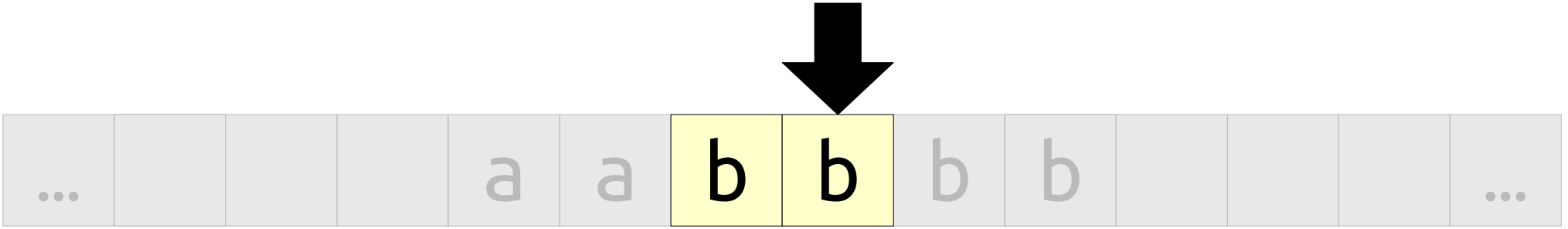
The Idea



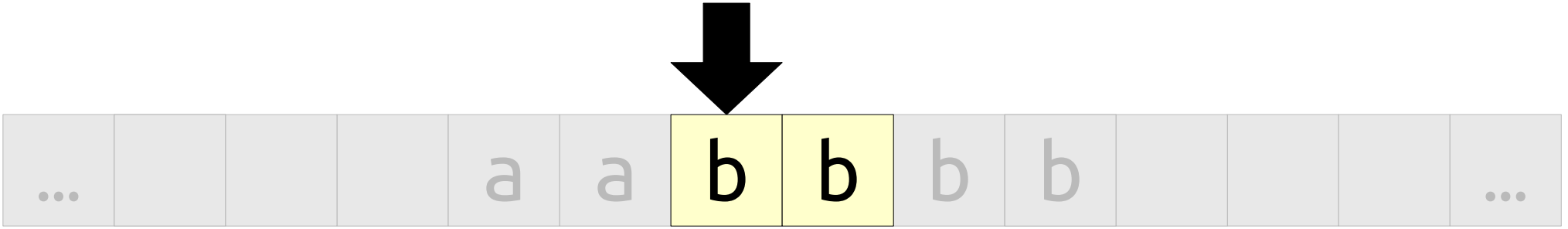
The Idea



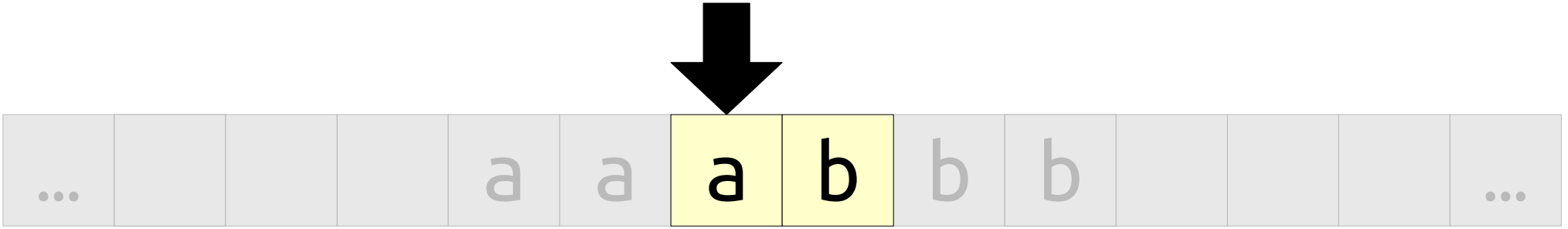
The Idea



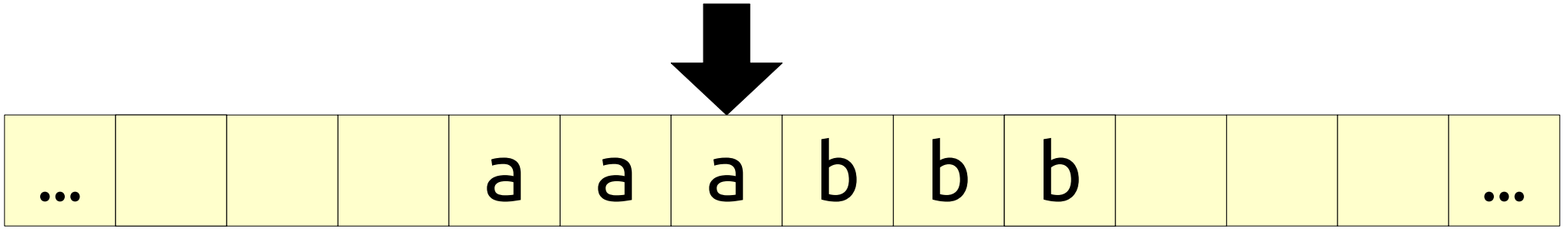
The Idea



The Idea



The Idea



Exploring This Idea

Cool TM Tricks 1: ***Fibonacci Numbers***

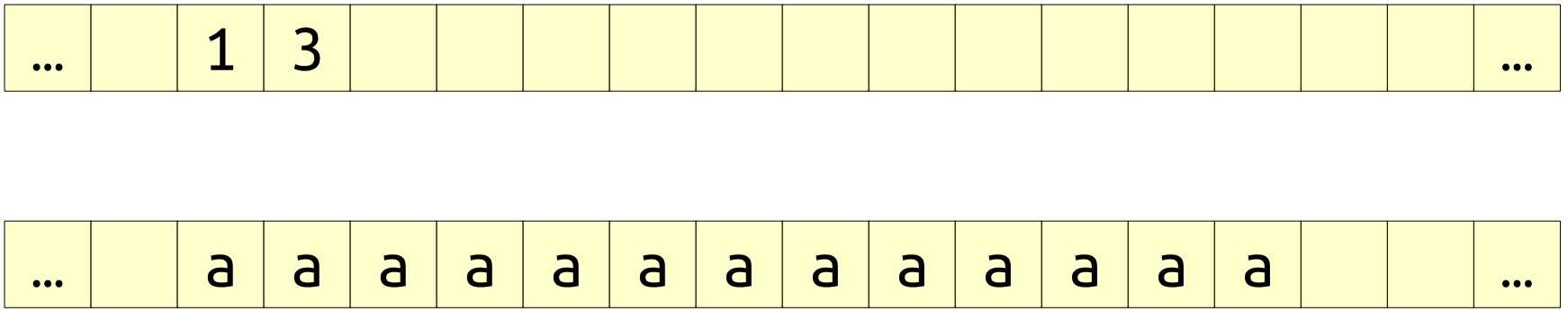
Fibonacci Numbers

$\{ \mathbf{a}^n \mid n \text{ is a Fibonacci number} \}$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

Cool TM Tricks 2: ***Decimal Fibonacci***

Decimal Fibonacci



{ *w* ∈ {**0**, **1**, **2**, … **9**}* | *w*, interpreted as a base-10 number, is a Fibonacci number. }

Just how powerful are Turing machines?

Real and “Ideal” Computers

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

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

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

Simulating a TM

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

```
Move   Write   Goto   Return   If
```

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

Simulating a TM

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

Simulating a Computer

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

What We've Seen

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

What We've Seen

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

What We've Seen

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

What We've Seen

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

What Else Can TMs Do?

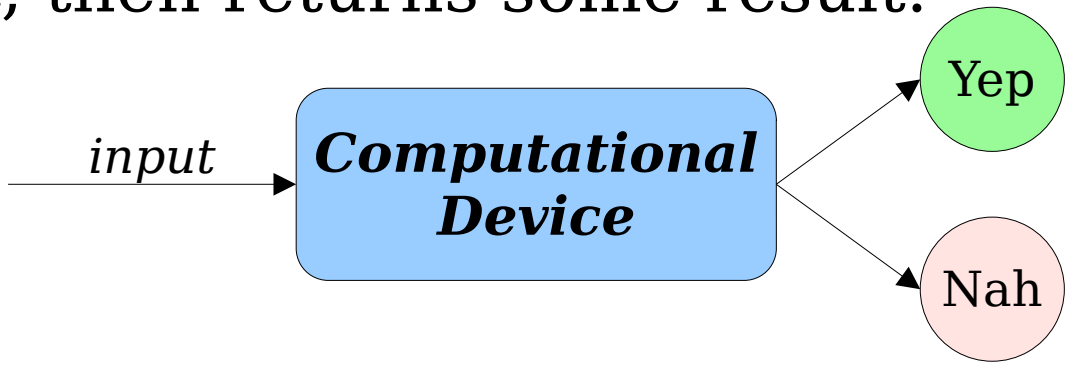
- Maintain strings and arrays.
  - Store their elements separated with some special separator character.
- Support pointers.
  - Maintain an array of what's in memory, where each item is tagged with its "memory address."
- Support function call and return.
  - It's hard, but you can do this if you can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

A CS107 Perspective

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

A Leap of Faith

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

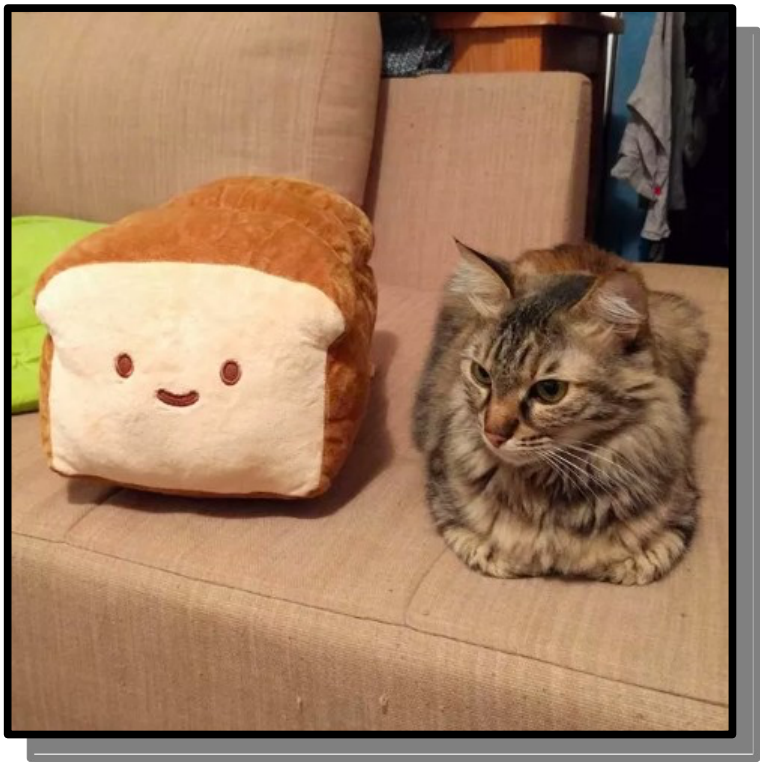


- The resulting TM might be colossal, or really slow, or both, but it would still faithfully simulate the computer.
- We're going to take this as an article of faith in CS103. If you curious for more details, come talk to me after class.

Can a TM Work With…

“cat pictures?”

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



Can a TM Work With…

~~“cat pictures?”~~

“cat videos?”

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

Can a TM Work With…

“music?”

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

“deep learning?”

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

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

Effective Computation

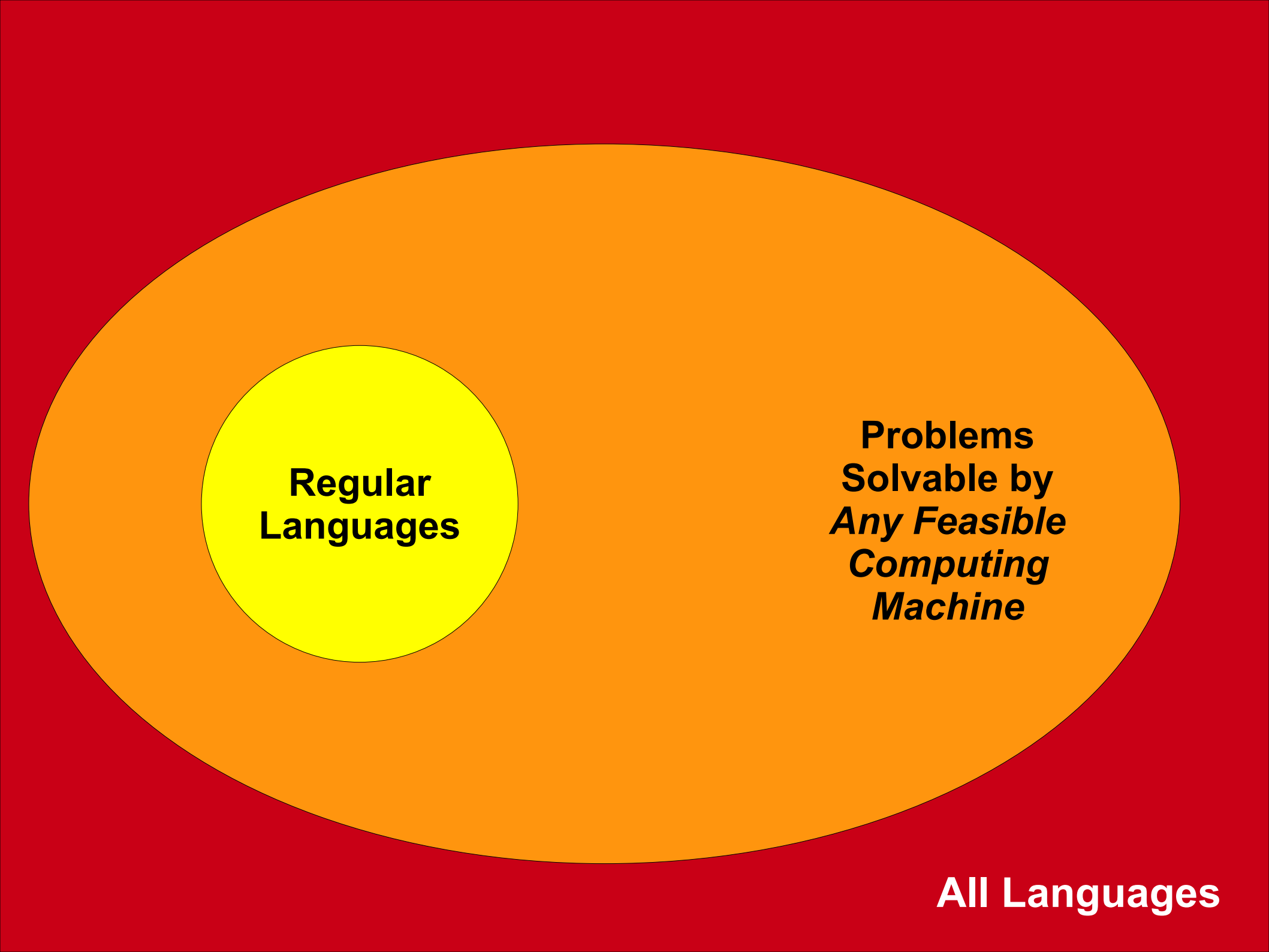
- An ***effective method of computation*** is a form of computation with the following properties:
  - The computation consists of a set of steps.
  - There are fixed rules governing how one step leads to the next.
  - Any computation that yields an answer does so in finitely many steps.
  - Any computation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

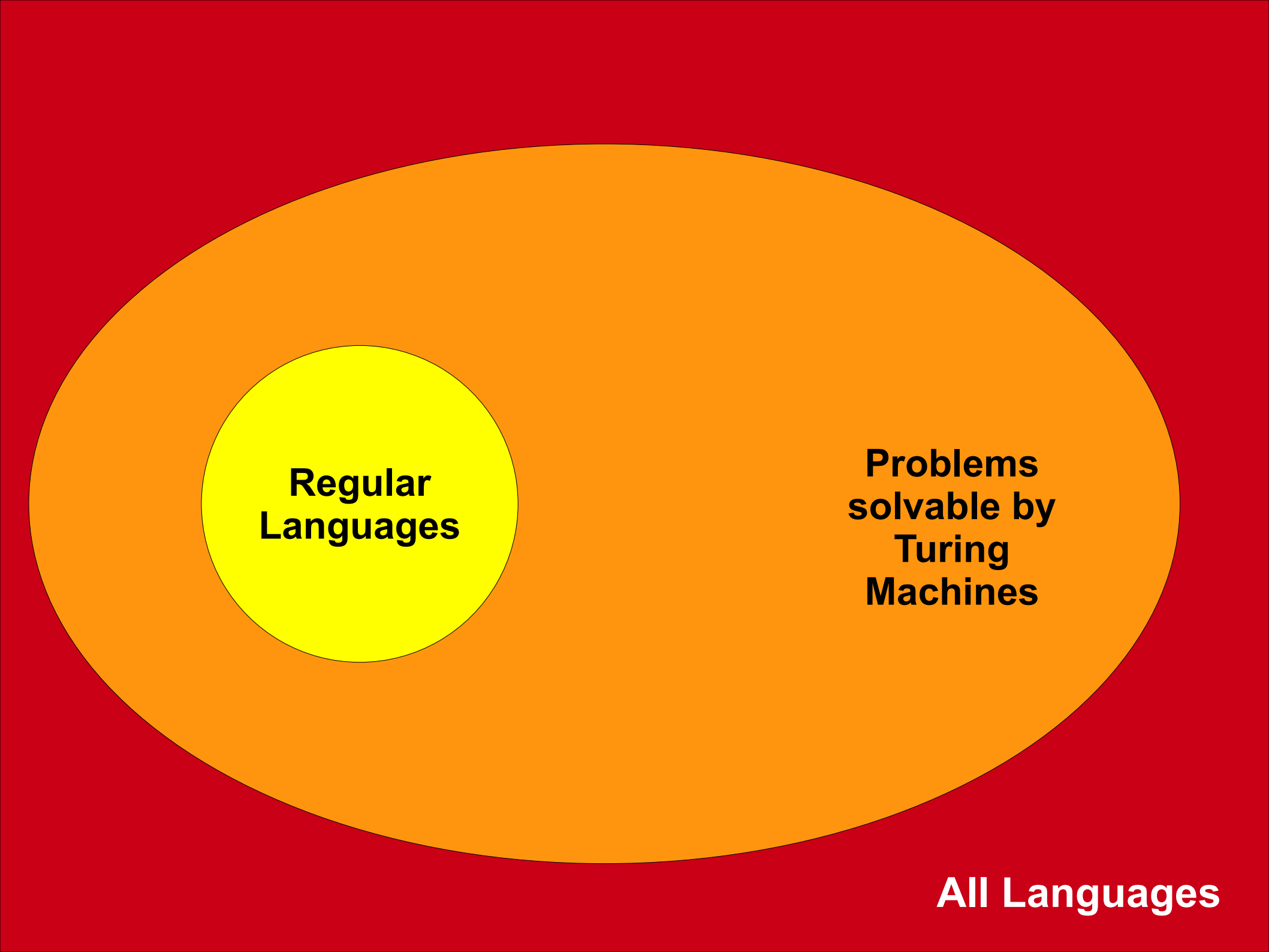
The ***Church-Turing Thesis*** claims that

every effective method of computation is either equivalent to or weaker than a Turing machine.

"This is not a theorem – it is a falsifiable scientific hypothesis. And it has been thoroughly tested!"

- Ryan Williams


Regular Languages
Problems Solvable by *Any Feasible Computing Machine*
All Languages


Regular Languages
Problems solvable by Turing Machines
All Languages

TMs and Computation

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

Decidability and Recognizability

What problems can we solve with a computer?

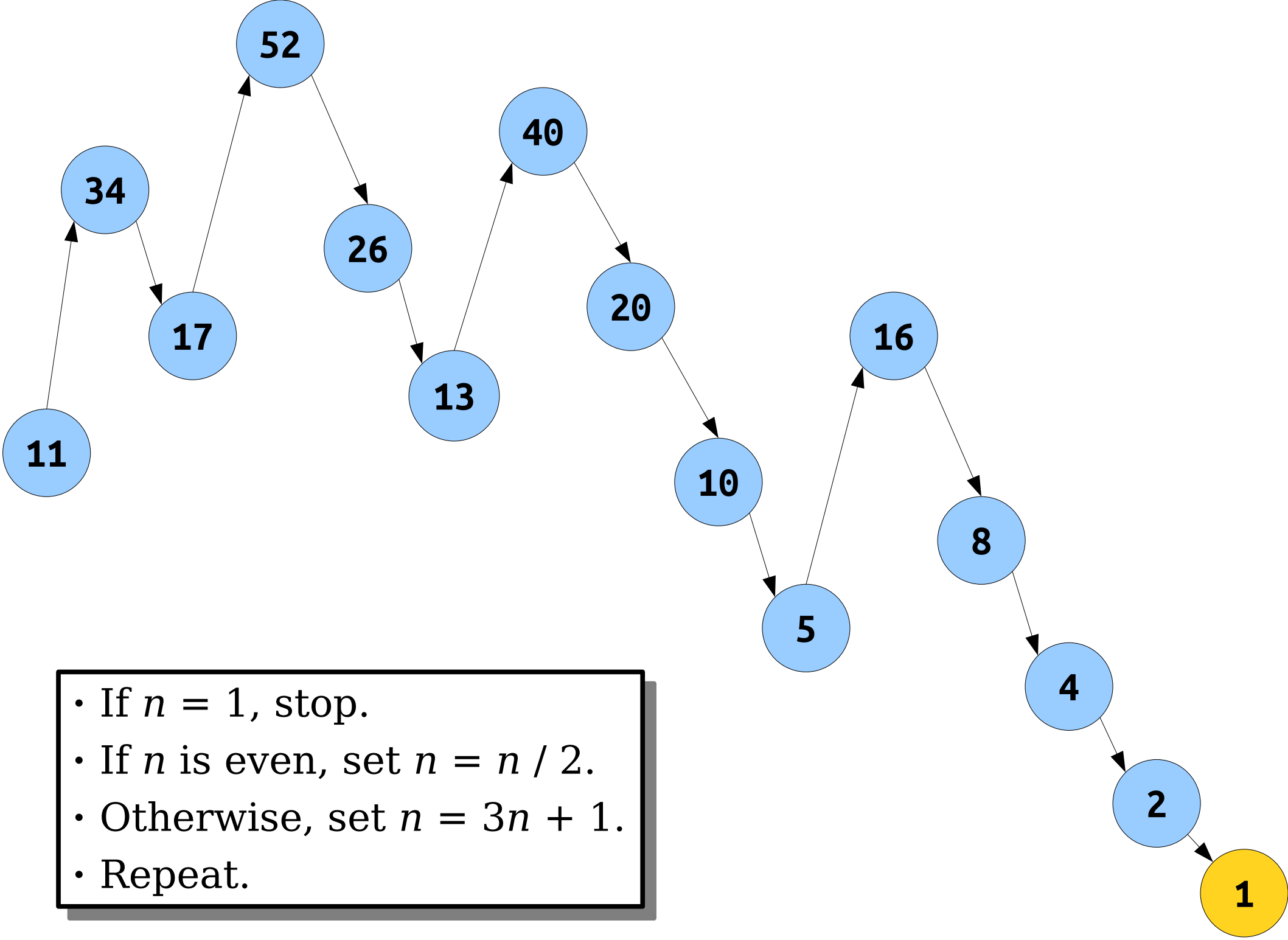
What kind of computer?

What problems can we solve with a computer?

What does it mean
to “solve” a
problem?

The Hailstone Sequence

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



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

The Hailstone Sequence

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

The Hailstone Sequence

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

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

- Could we build a TM for L?

The Hailstone Turing Machine

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

Does this Turing machine accept all nonempty strings?

The Collatz Conjecture

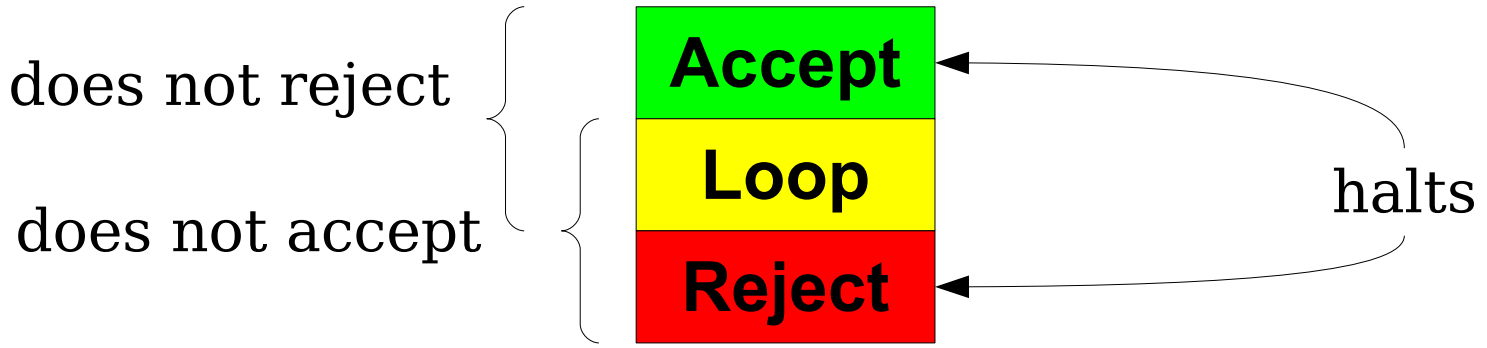
- It is *unknown* whether this process will terminate for all natural numbers.
- In other words, no one knows whether the TM described in the previous slides will always stop running!
- The conjecture (unproven claim) that the hailstone sequence always terminates is called the ***Collatz Conjecture***.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

An Important Observation

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

Very Important Terminology

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



Recognizers and Recognizability

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

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

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

Recognizers and Recognizability

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

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

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

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

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

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

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

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

  return true;
}
```

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

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

Recognizers and Recognizability

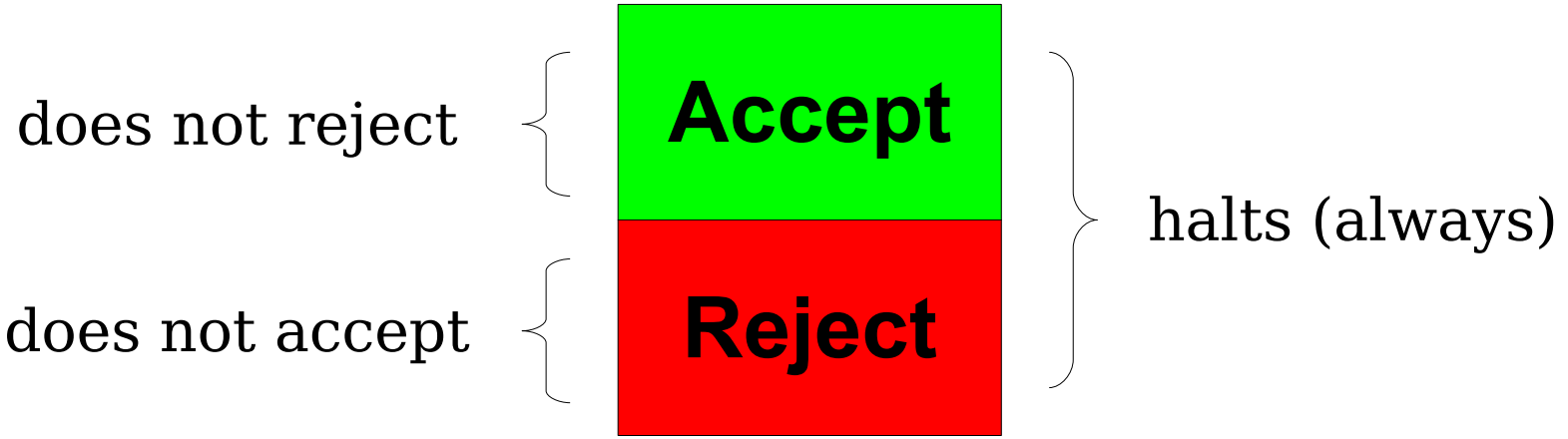
- The class **RE** consists of all recognizable languages.
- Formally speaking:

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

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

Deciders and Decidability

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



Deciders and Decidability

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

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

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

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

Deciders and Decidability

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

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

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

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

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

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

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

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

  return true;
}
```

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

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

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

Deciders and Decidability

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

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

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

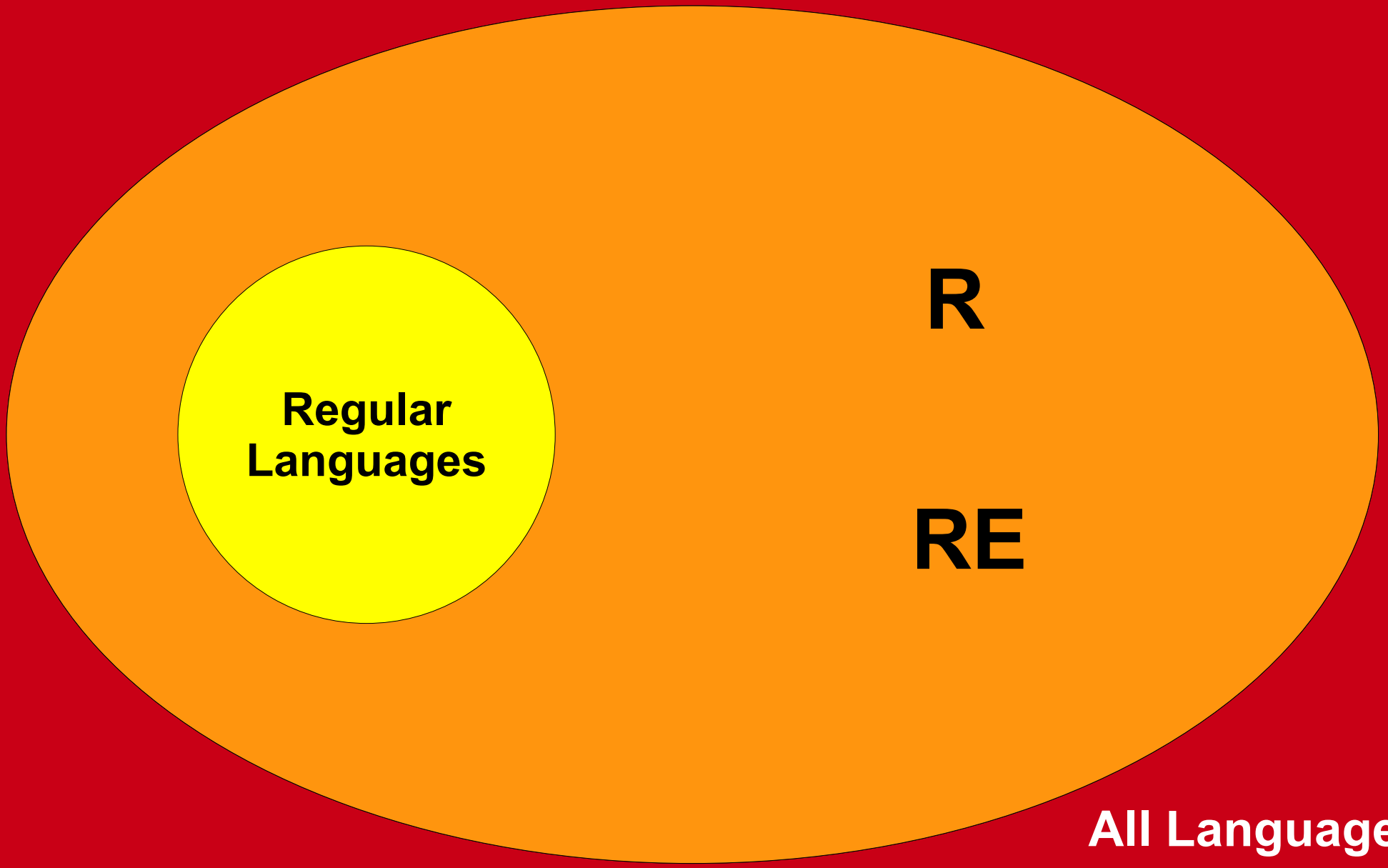
R and **RE** Languages

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

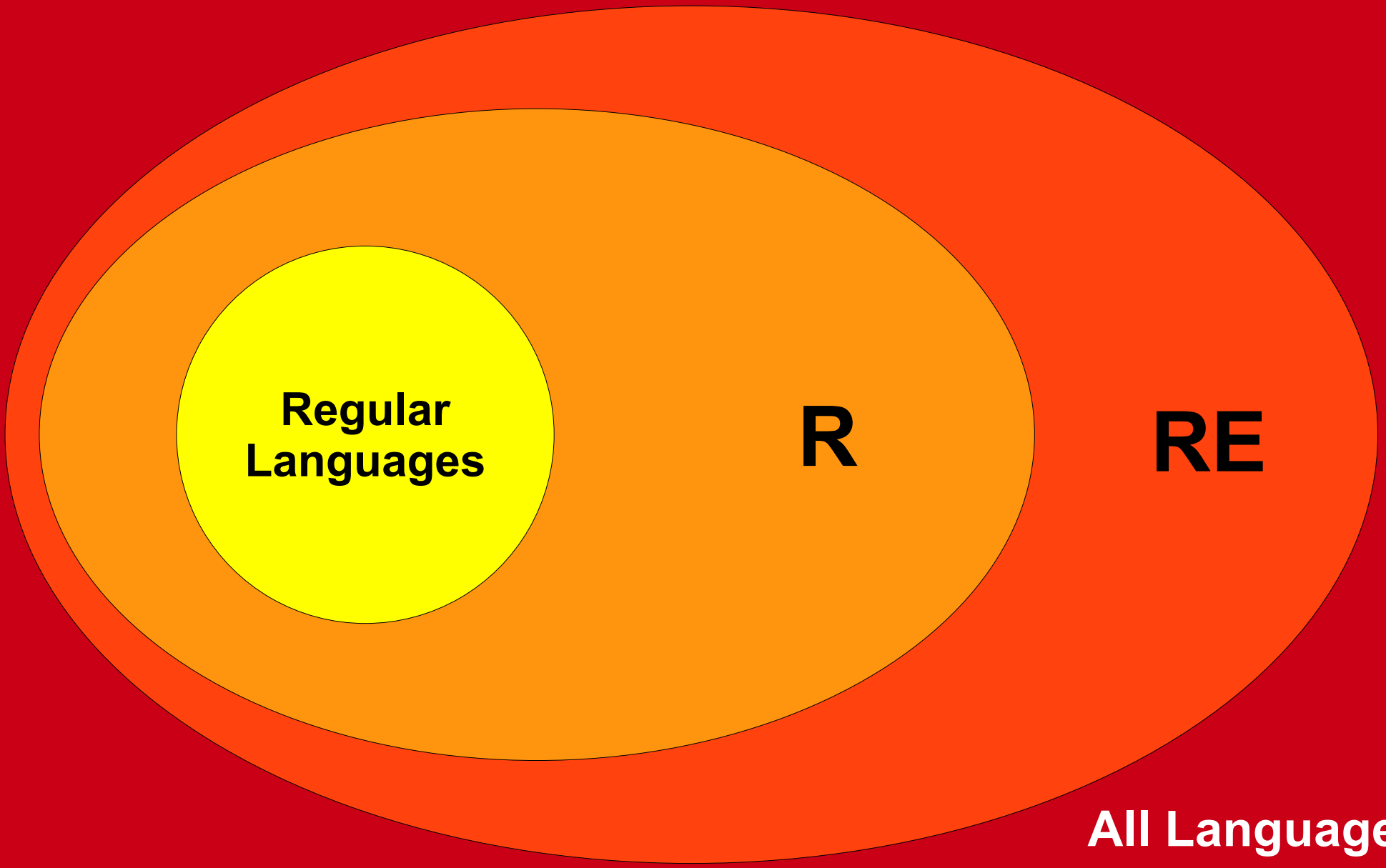
$$\mathbf{R} \stackrel{?}{=} \mathbf{RE}$$

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

Which Picture is Correct?



Which Picture is Correct?



Unanswered Questions

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

Turing Machines

Part Two

Outline for Today

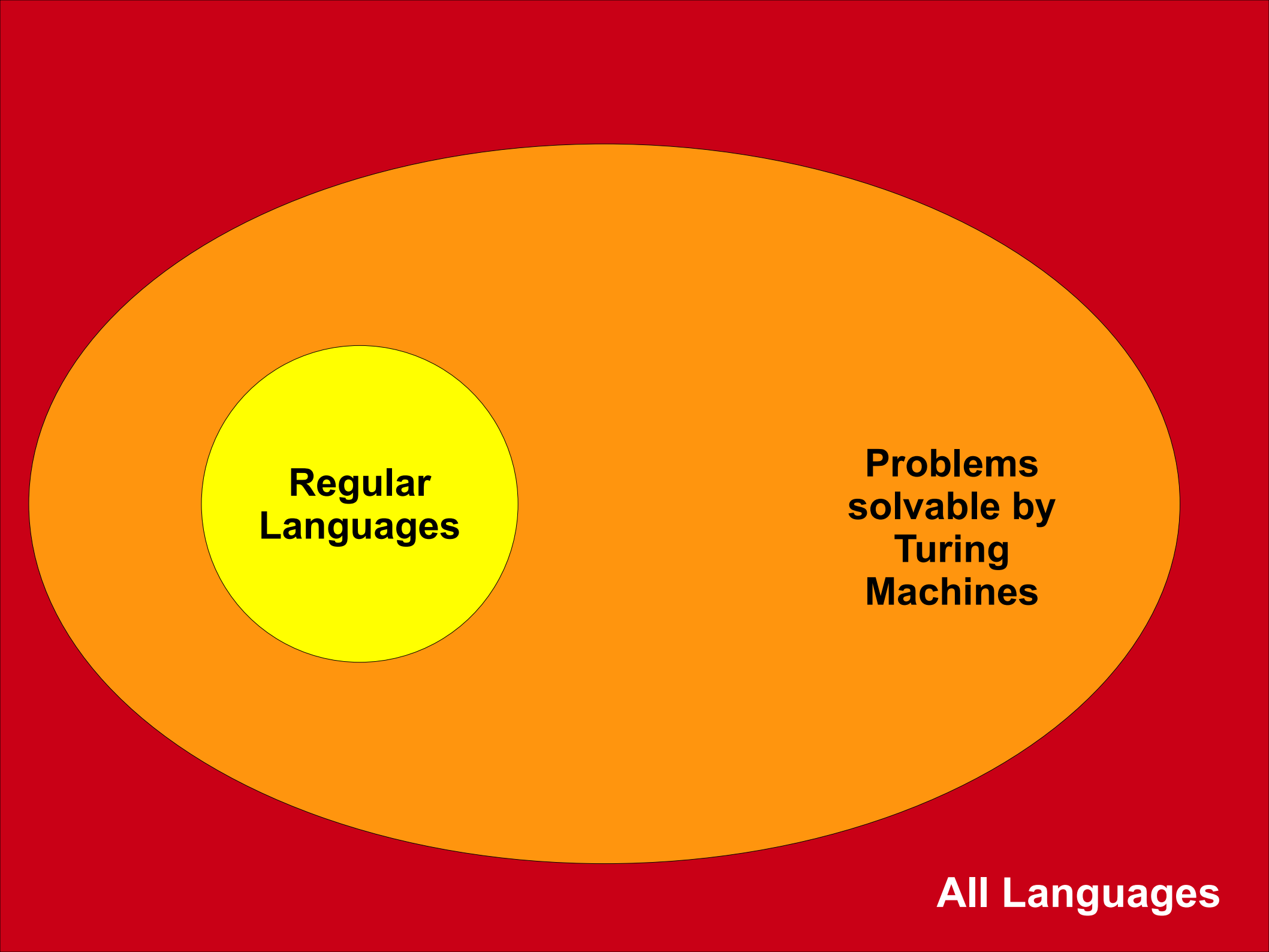
- ***The Church-Turing Thesis***
  - How powerful are Turing machines?
- ***Decidability and Recognizability***
  - Two notions of “solving a problem.”
- ***Universal Machines***
  - A single computer that can compute anything computable anywhere.
- ***Self-Referential Software***
  - Programs that compute on themselves.

The ***Church-Turing Thesis*** claims that

every effective method of computation is either equivalent to or weaker than a Turing machine.

"This is not a theorem – it is a falsifiable scientific hypothesis. And it has been thoroughly tested!"

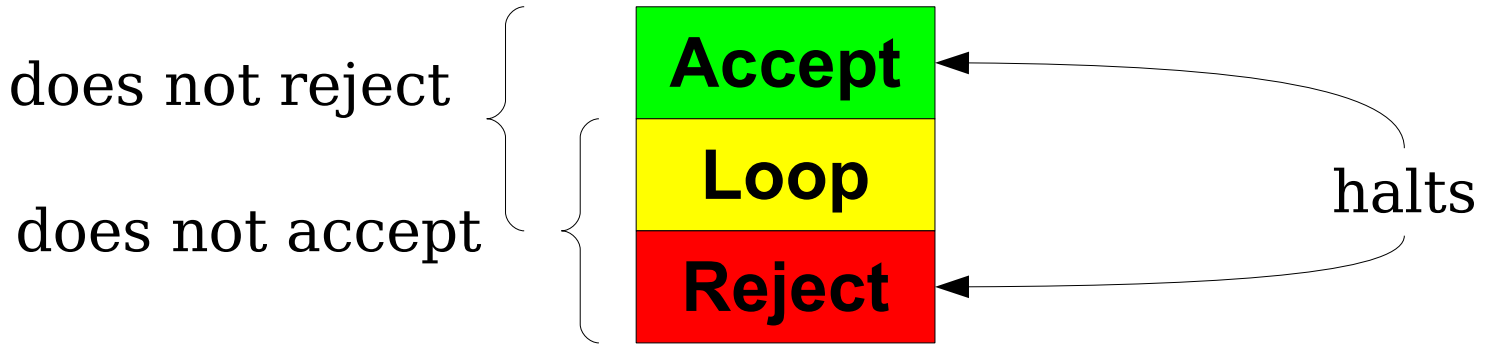
- Ryan Williams


Regular Languages
Problems solvable by Turing Machines
All Languages

Decidability and Recognizability

Very Important Terminology

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



Recognizers and Recognizability

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

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

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

Recognizers and Recognizability

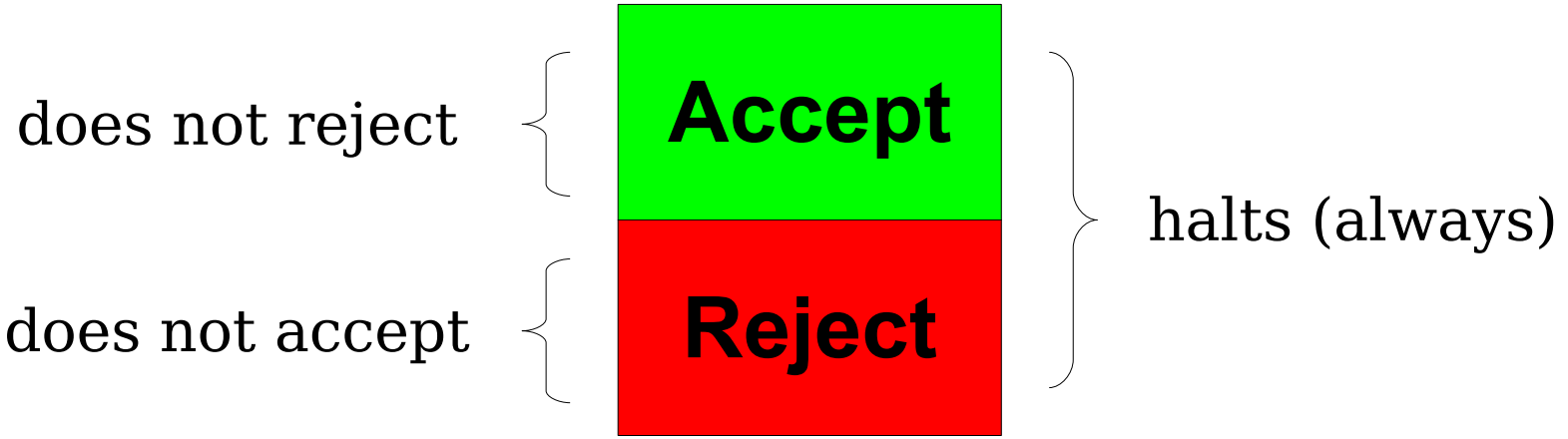
- The class **RE** consists of all recognizable languages.
- Formally speaking:

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

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

Deciders and Decidability

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



Deciders and Decidability

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

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

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

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

Deciders and Decidability

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

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

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

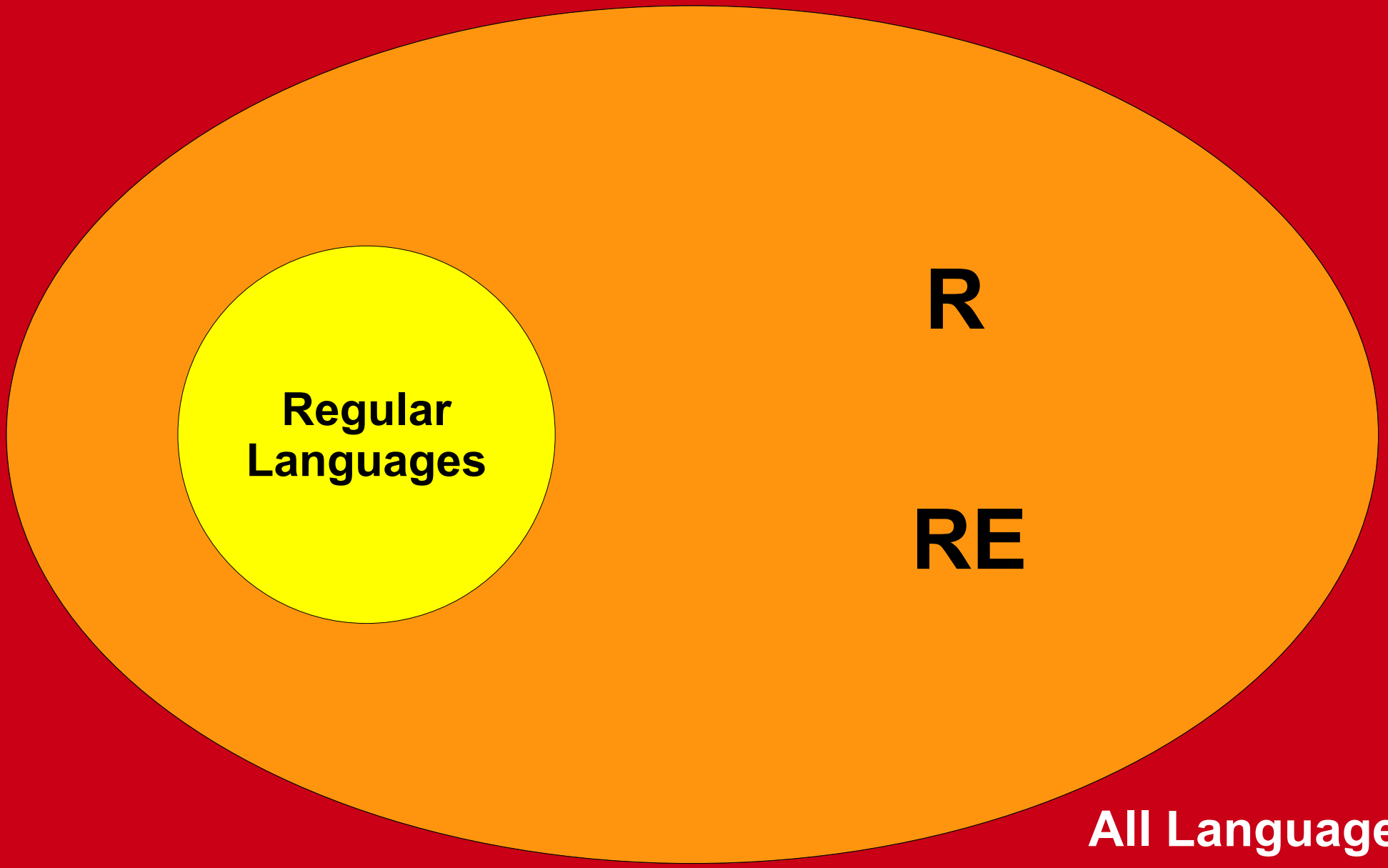
R and **RE** Languages

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

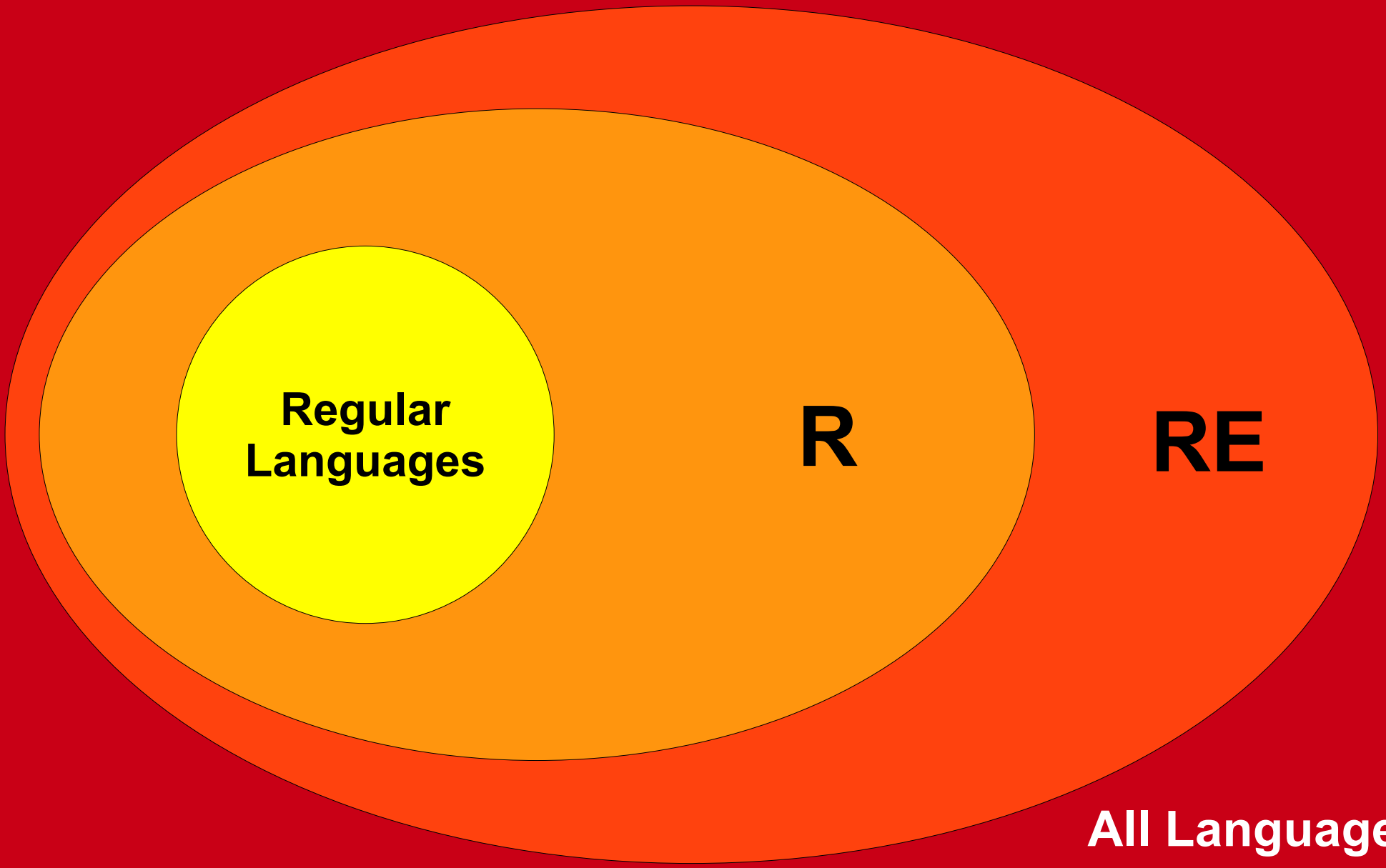
$$\mathbf{R} \stackrel{?}{=} \mathbf{RE}$$

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

Which Picture is Correct?



Which Picture is Correct?



Strings, Languages, and Encodings

What problems can we solve with a computer?

What is a
“problem?”

Decision Problems

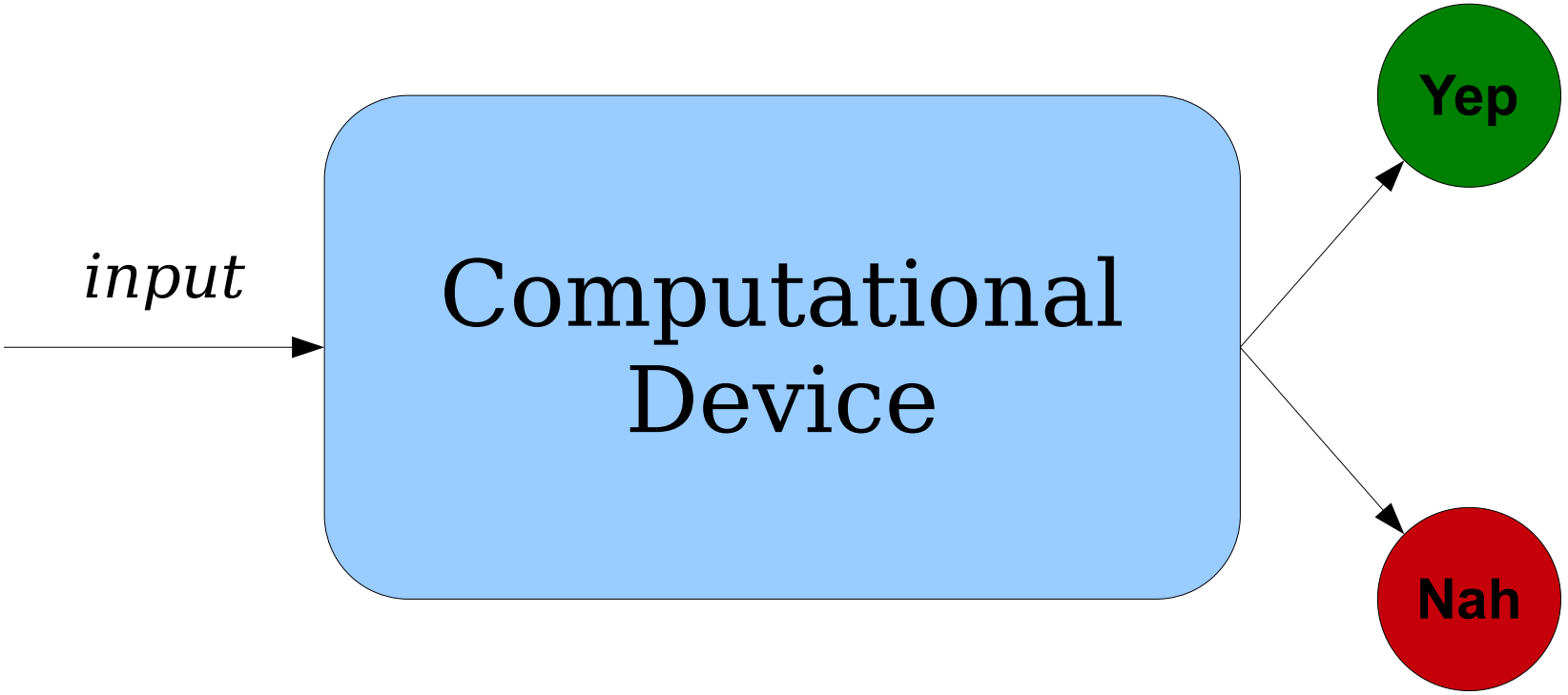
- A ***decision problem*** is a type of problem where the goal is to provide a yes or no answer.
- Example: Bin Packing

  You're given a list of patients who need to be seen and how much time each one needs to be seen for. You're given a list of doctors and how much free time they have. Is there a way to schedule the patients so that they can all be seen?

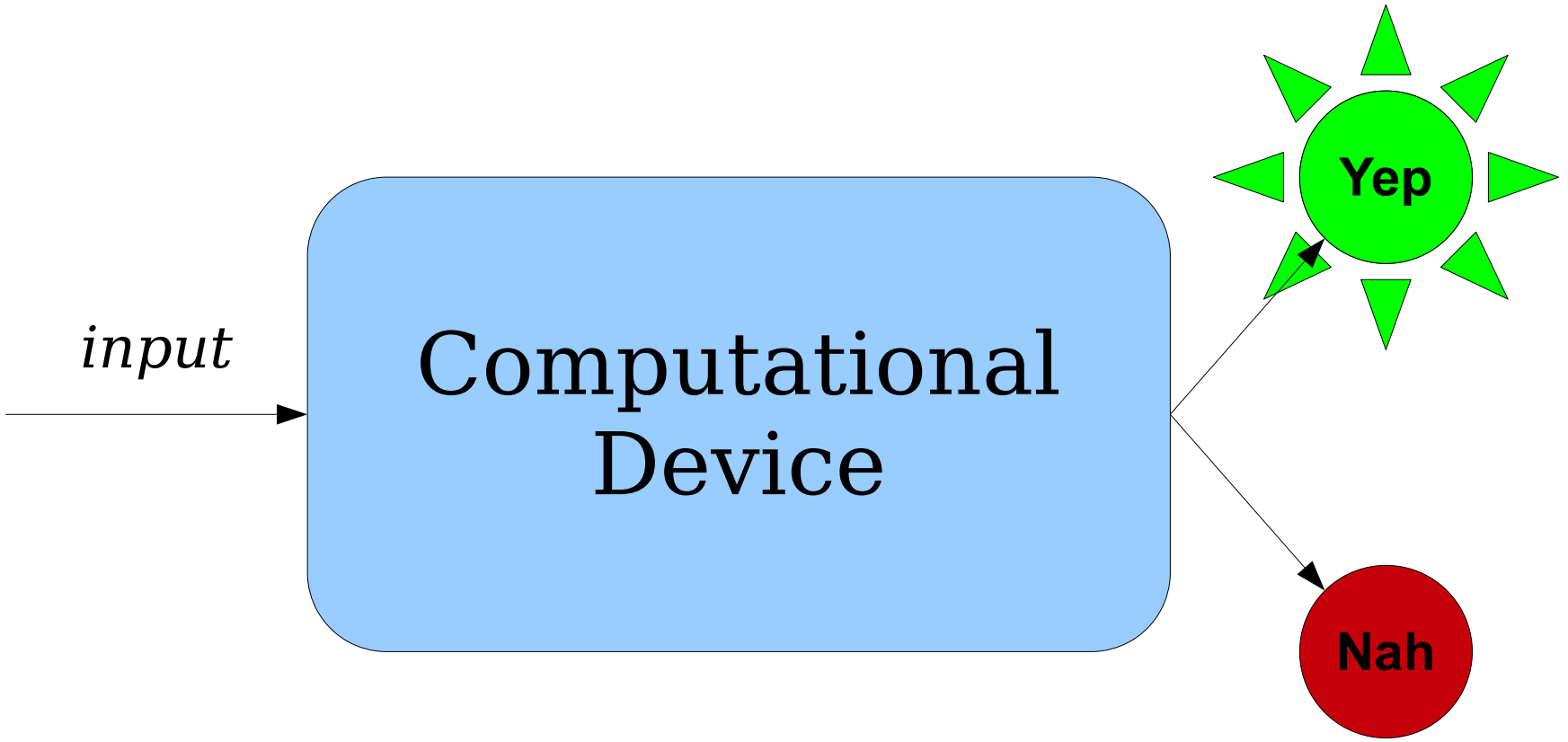
- Example: Dominating Set Problem

  You're given a transportation grid and a number k. Is there a way to place emergency supplies in at most k cities so that every city either has emergency supplies or is adjacent to a city that has emergency supplies?

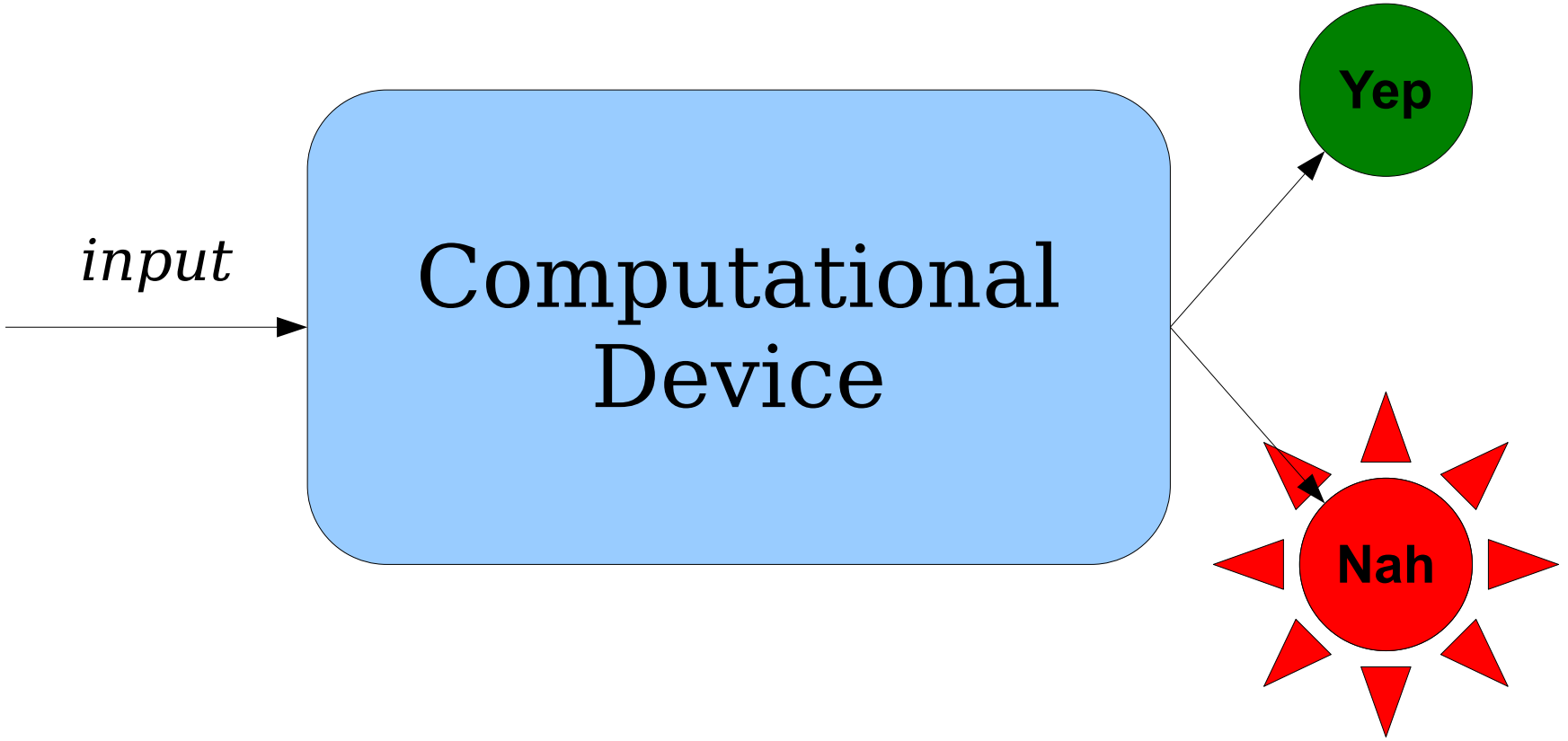
A Model for Solving Problems



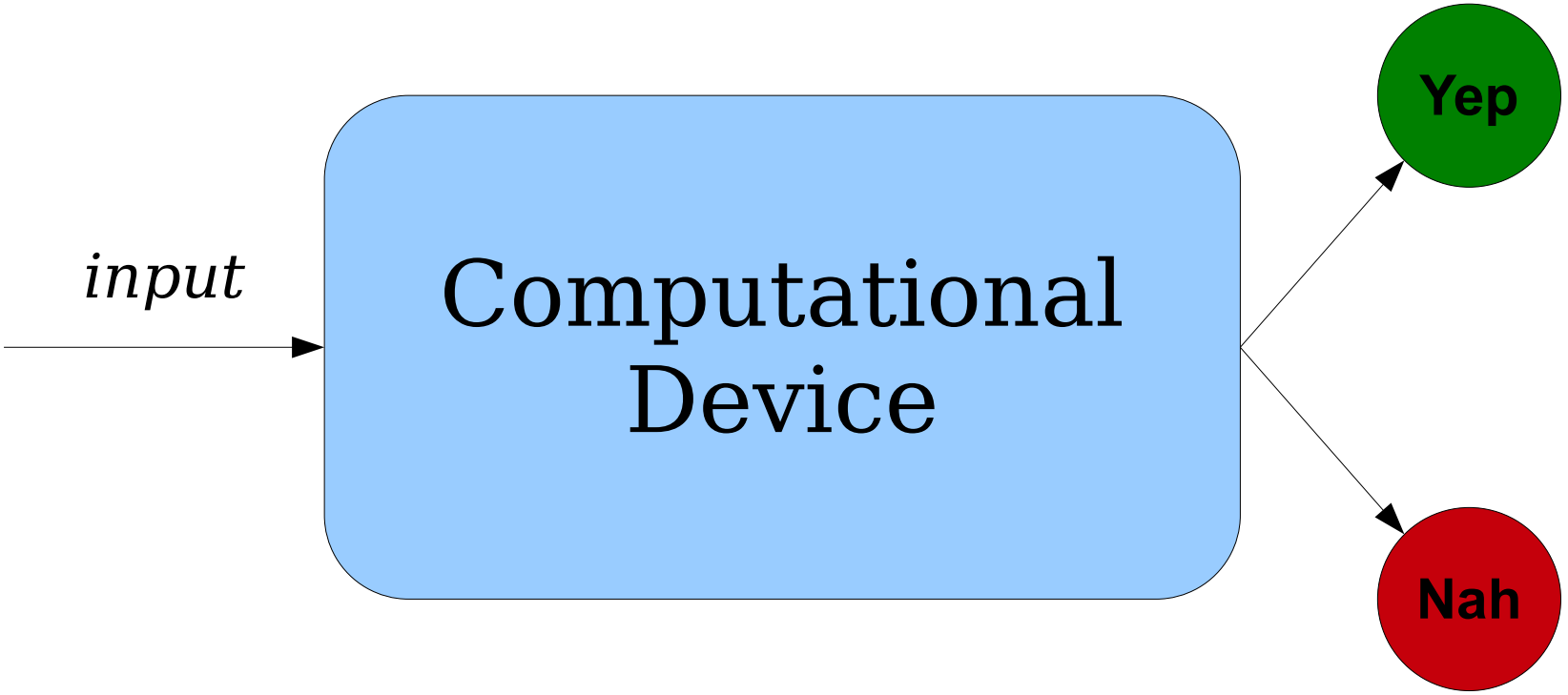
A Model for Solving Problems



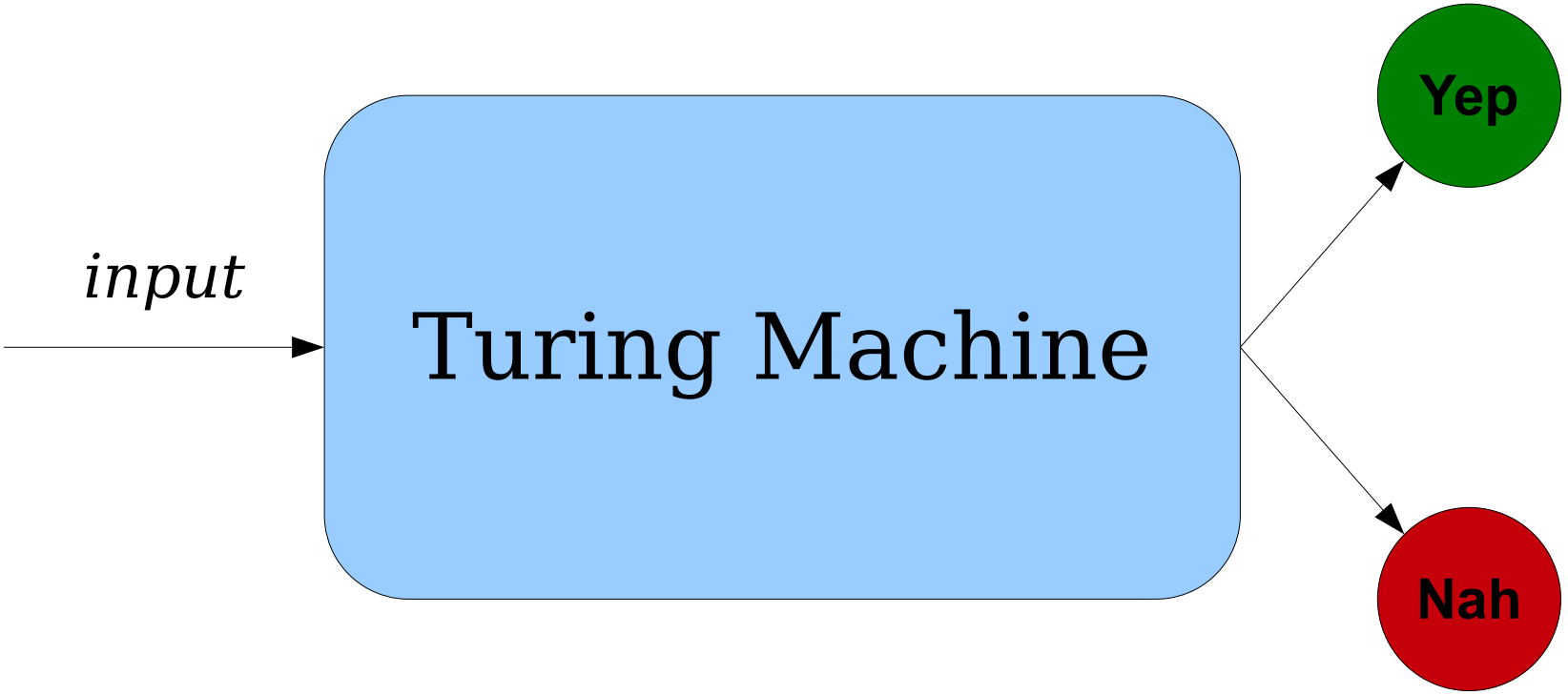
A Model for Solving Problems



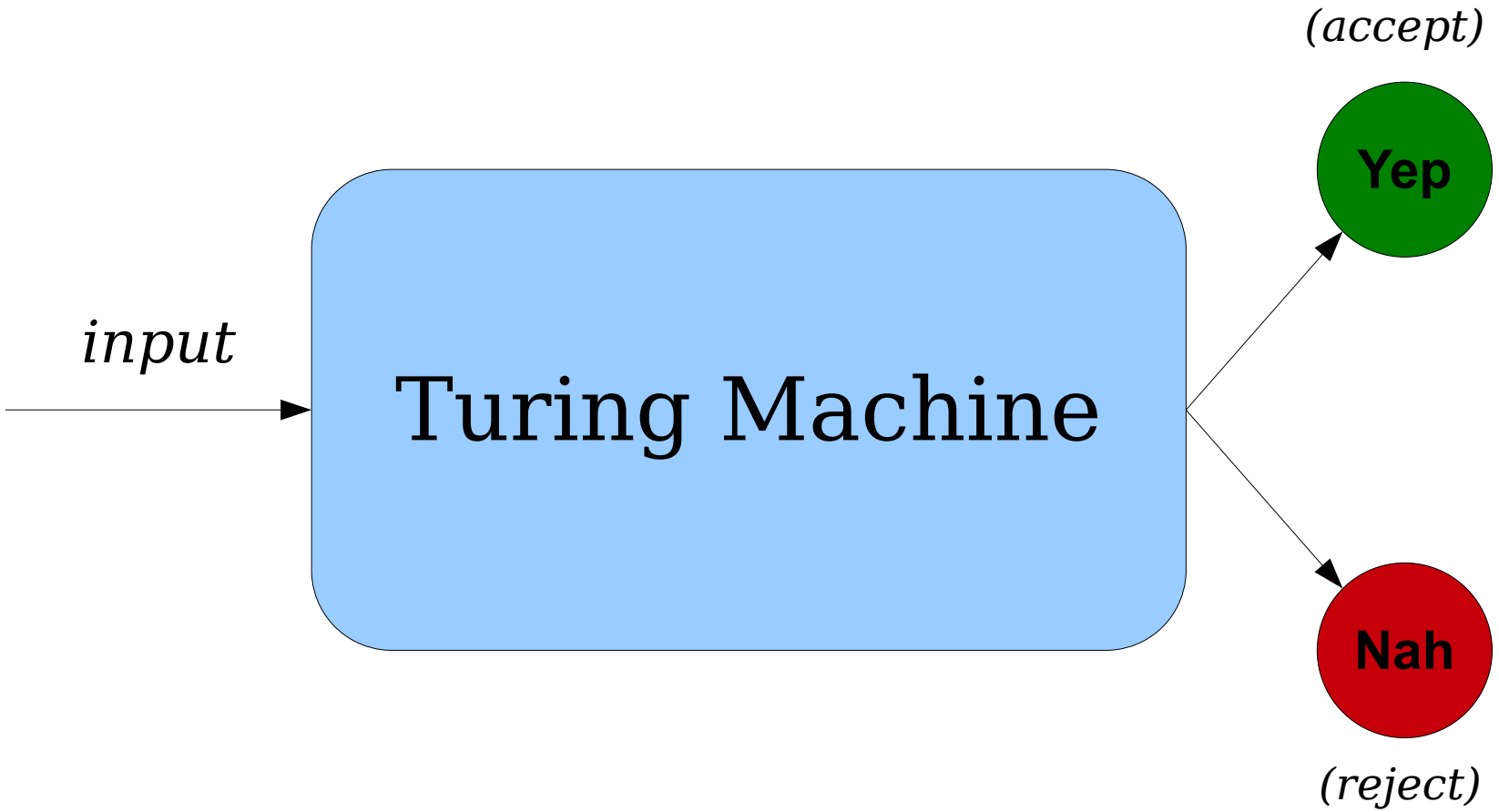
A Model for Solving Problems



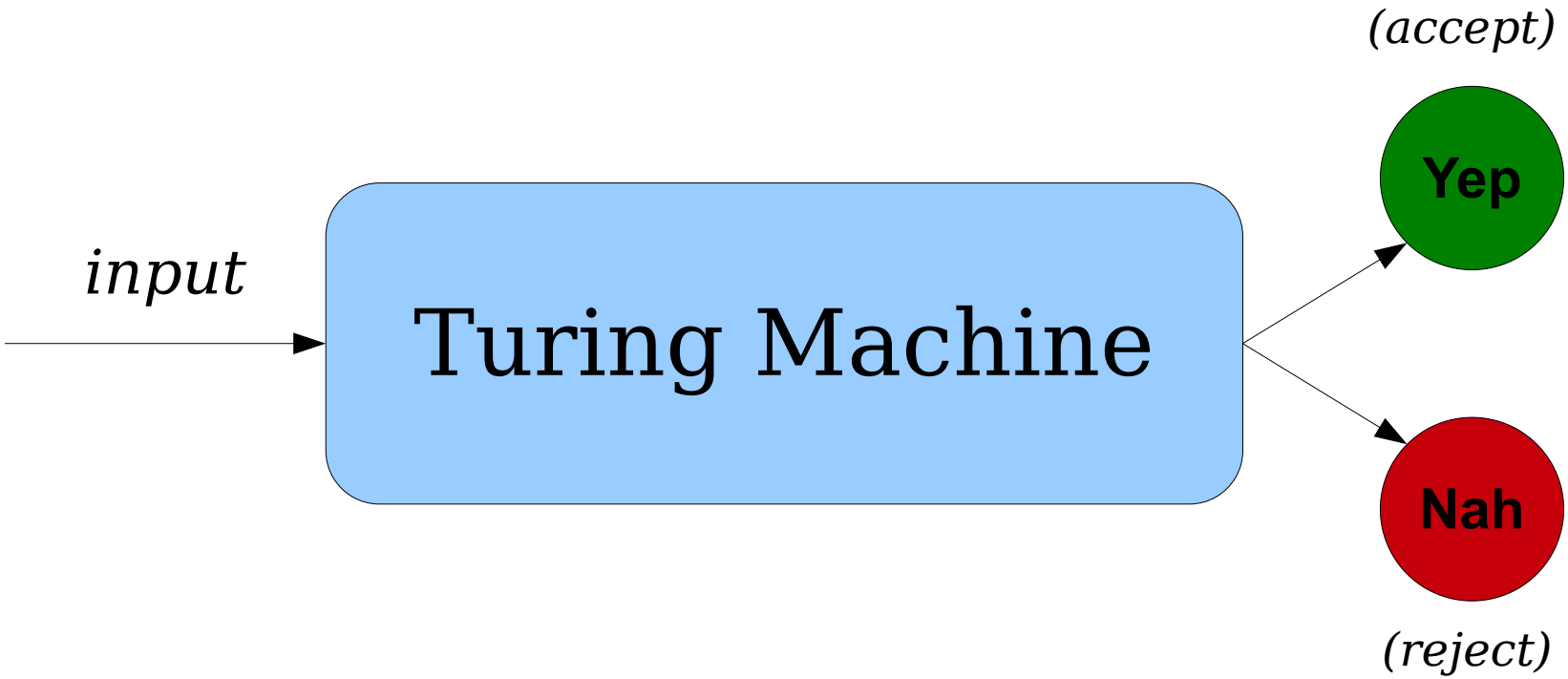
A Model for Solving Problems



A Model for Solving Problems

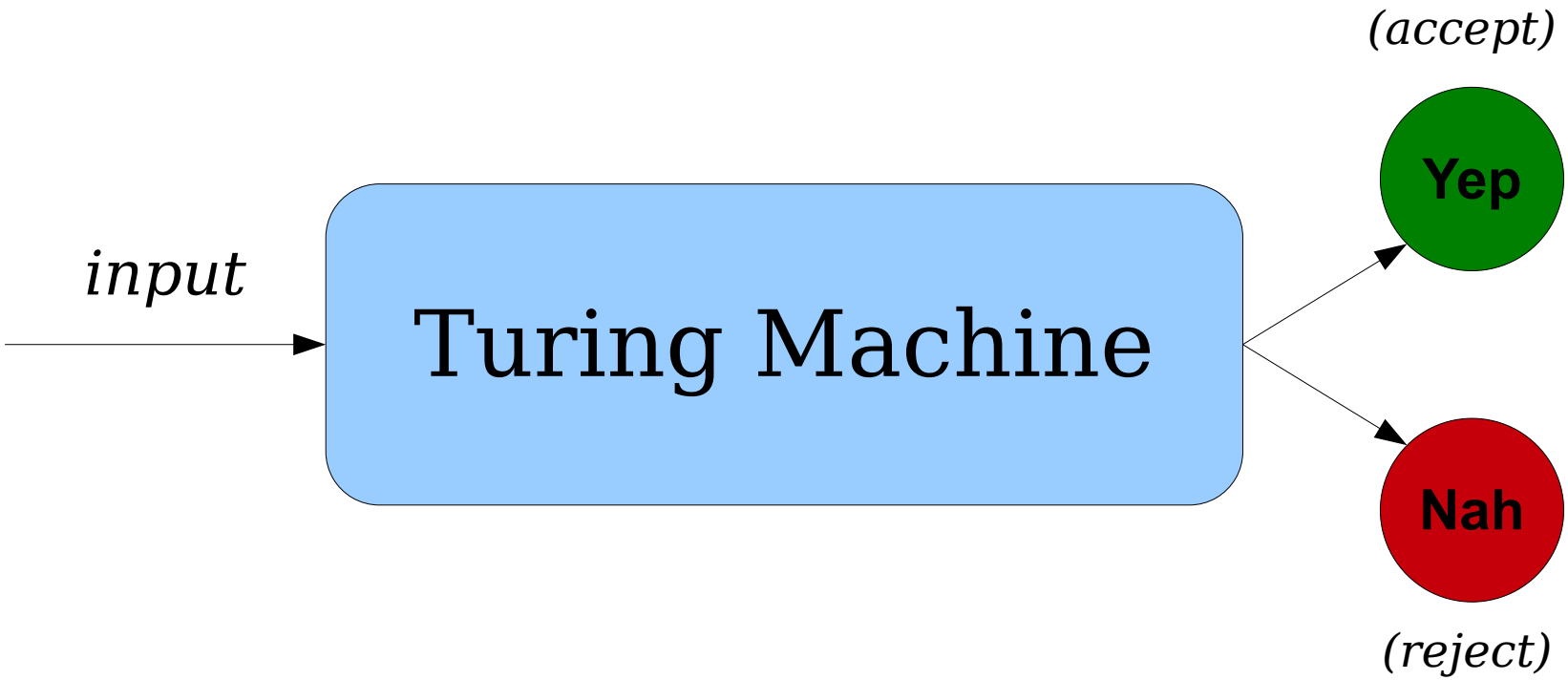


A Model for Solving Problems



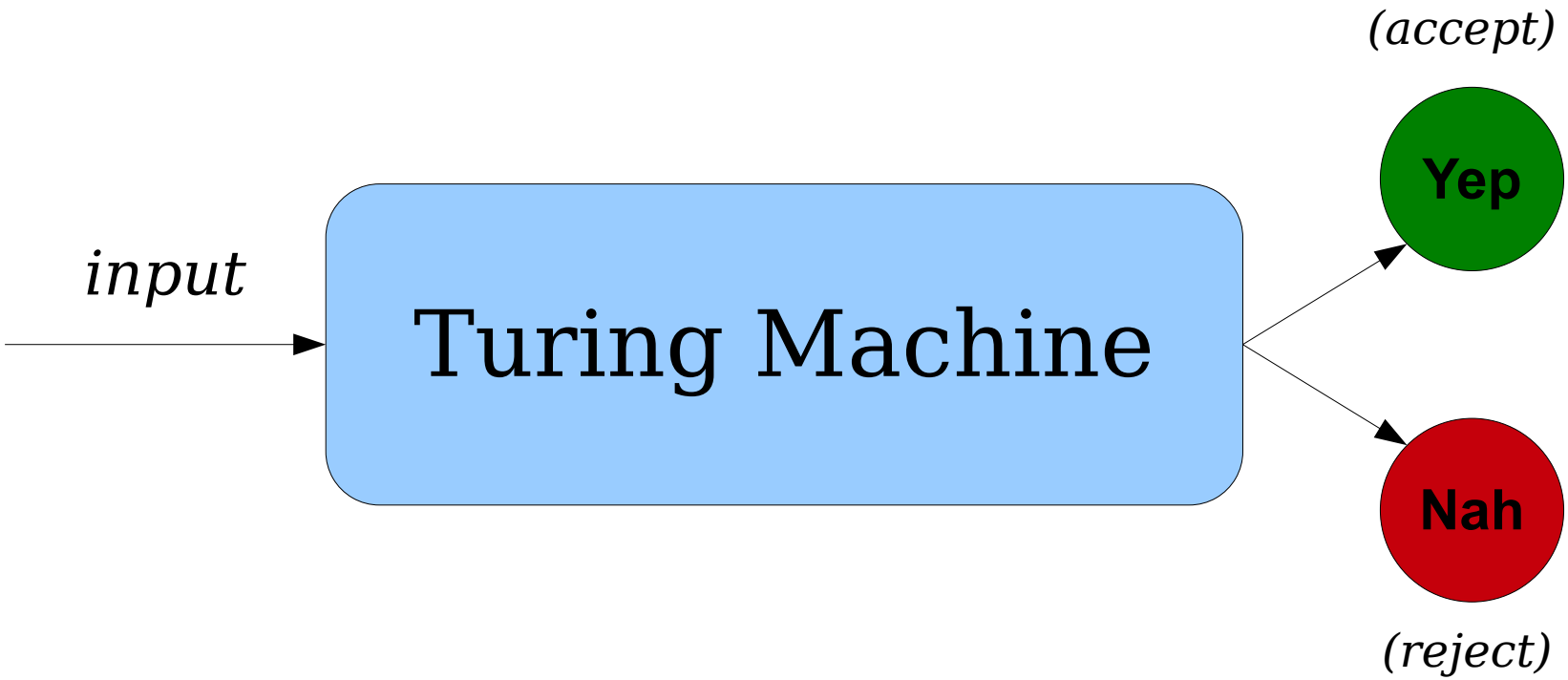

```
bool someFunctionName(string input) {

    // … do something …

}
```

A Model for Solving Problems



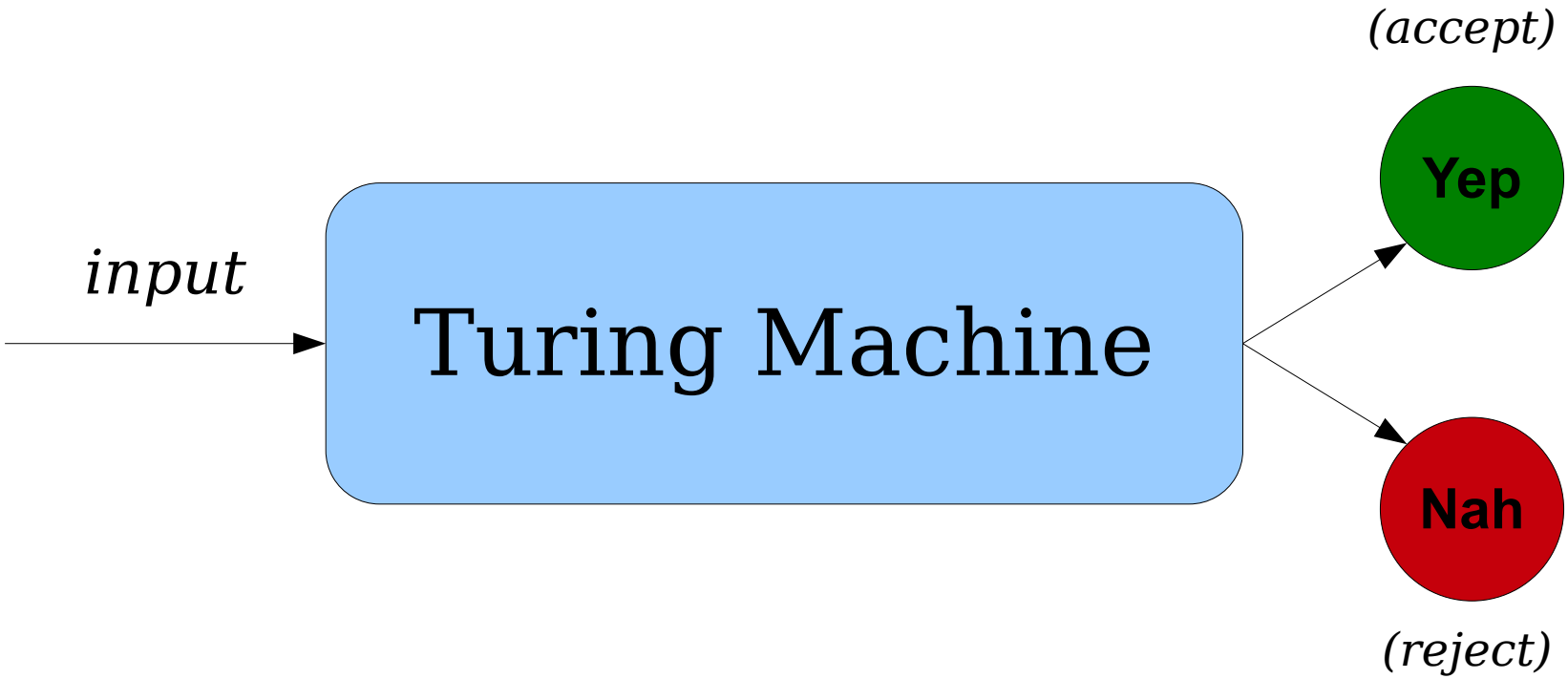

```
bool isAnBn(string input) {

    // … do something …

}
```

A Model for Solving Problems




```
bool isPalindrome(string input) {

    // … do something …

}
```

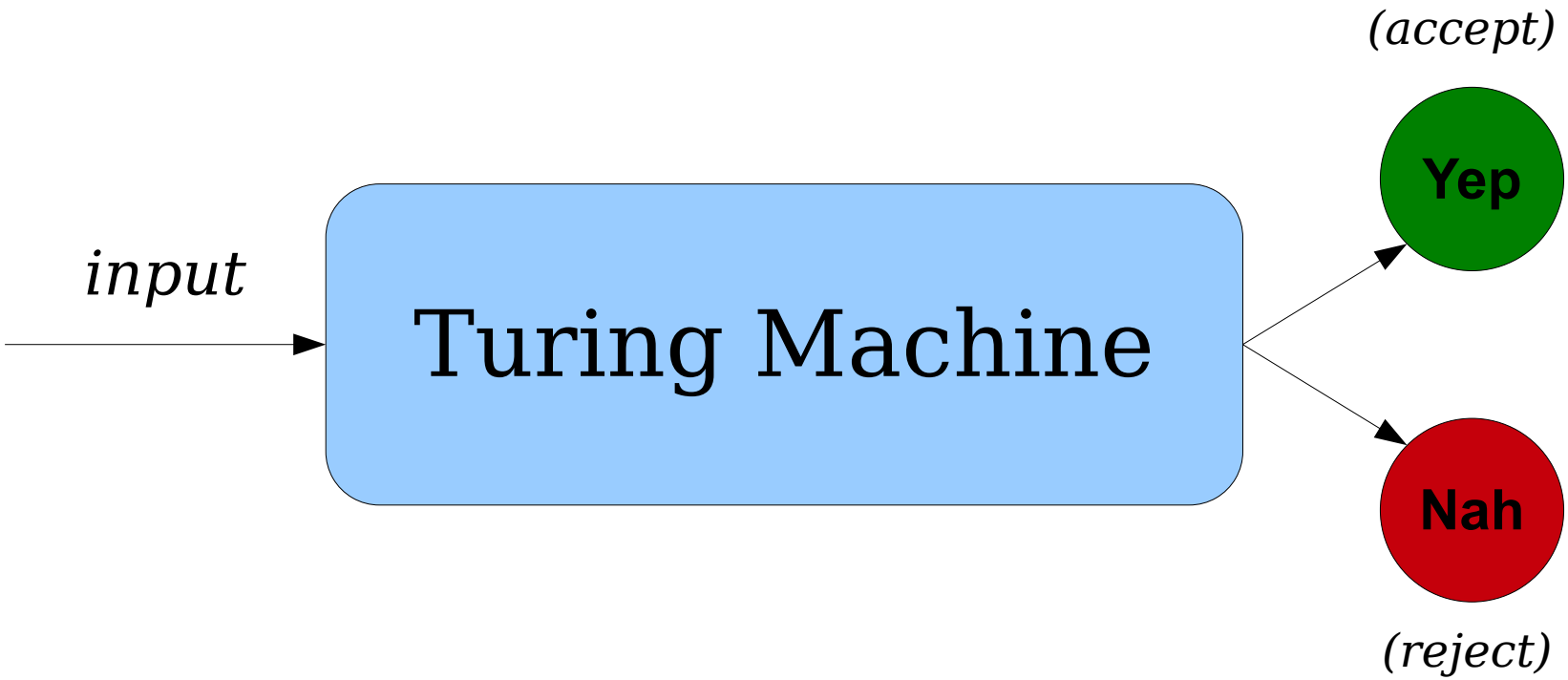
A Model for Solving Problems




```
bool isLinkageGraph(Graph G) {
    // … do something …
}
```

How does this match our model?

A Model for Solving Problems




```
bool containsCat(Picture P) {

    // … do something …

}
```

How does this match our model?

Humbling Thought:

Everything on your computer is a string over {0, 1}.

Strings and Objects

- Think about how my computer encodes the image on the right.
- Internally, it's just a series of zeros and ones sitting on my hard drive.

Strings and Objects

- A different sequence of `0`s and `1`s gives rise to the image on the right.
- Every image can be encoded as a sequence of `0`s and `1`s, though not all sequences of `0`s and `1`s correspond to images.

Object Encodings

- If *Obj* is some mathematical object that is *discrete* and *finite*, then we'll use the notation ***⟨Obj⟩*** to refer to some way of encoding that object as a string.
- Think of ⟨*Obj*⟩ like a file on disk – it encodes some high-level object as a series of characters.

⟨  ⟩ = 110111001011…110

Object Encodings

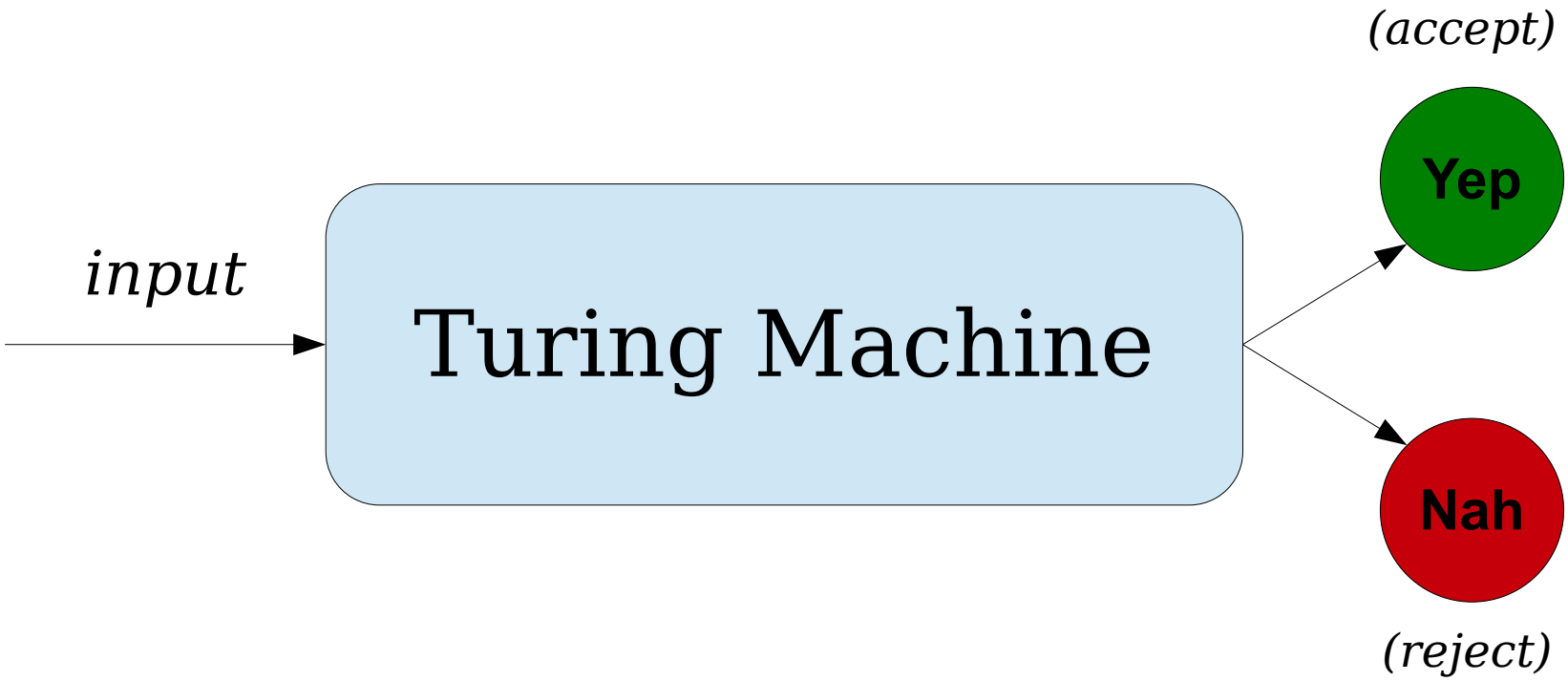
- If *Obj* is some mathematical object that is *discrete* and *finite*, then we'll use the notation **⟨*Obj*⟩** to refer to some way of encoding that object as a string.
- Think of ⟨*Obj*⟩ like a file on disk – it encodes some high-level object as a series of characters.

⟨  ⟩ = 001101010001…001

Object Encodings

- For the purposes of what we're going to be doing, we aren't going to worry about exactly *how* objects are encoded.
- For example, we can say ⟨137⟩ to mean "some encoding of 137" without worrying about how it's encoded.
  - Analogy: do you need to know how numbers are represented in Python to be a Python programmer? That's more of a CS107 or CS41 question.
- We'll assume, whenever we're dealing with encodings, that some Smart, Attractive, Witty person has figured out an encoding system for us and that we're using that encoding system.

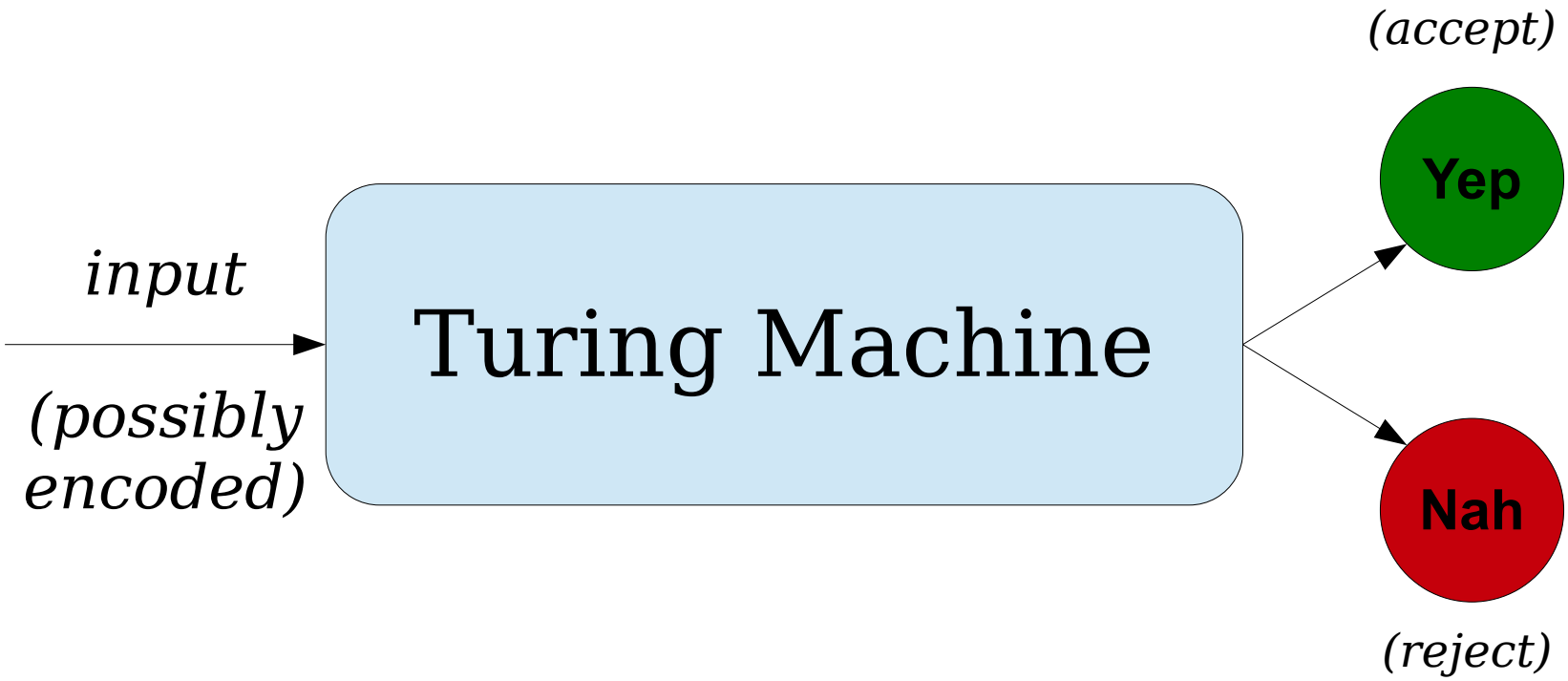
A Model for Solving Problems




```
bool containsCat(Picture P) {

    // … do something …

}
```

Internally, this is a sequence of **0**s and **1**s.

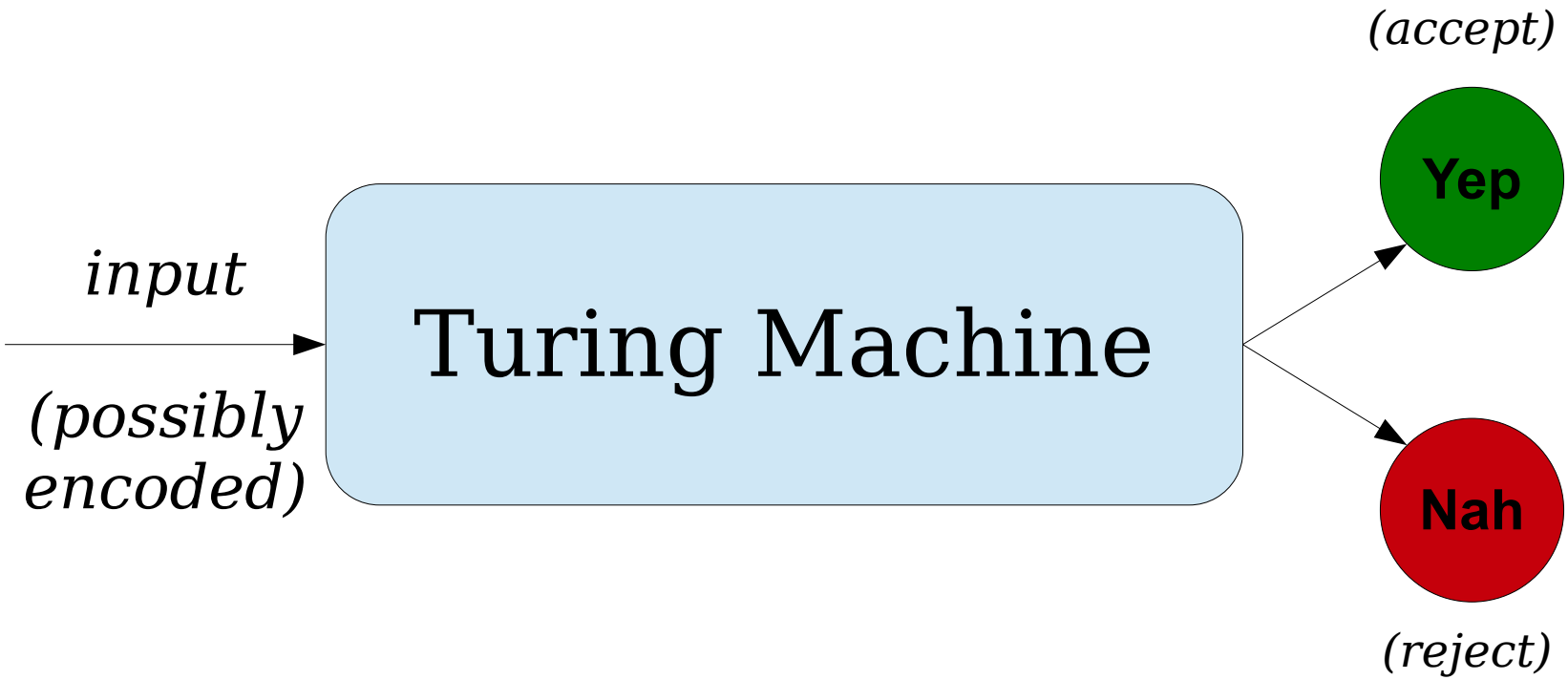
A Model for Solving Problems




```
bool containsCat(Picture P) {

    // … do something …

}
```

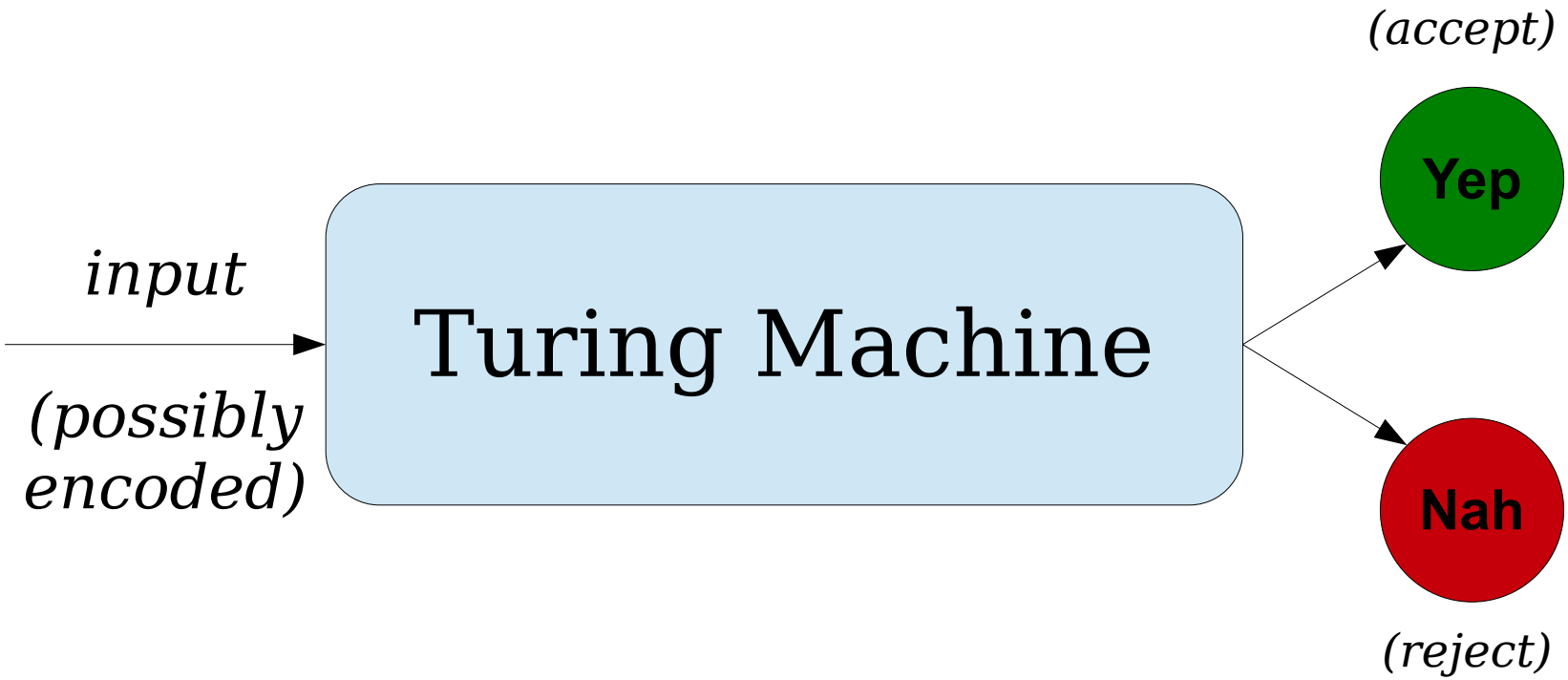
Internally, this is a sequence of **0**s and **1**s.

A Model for Solving Problems



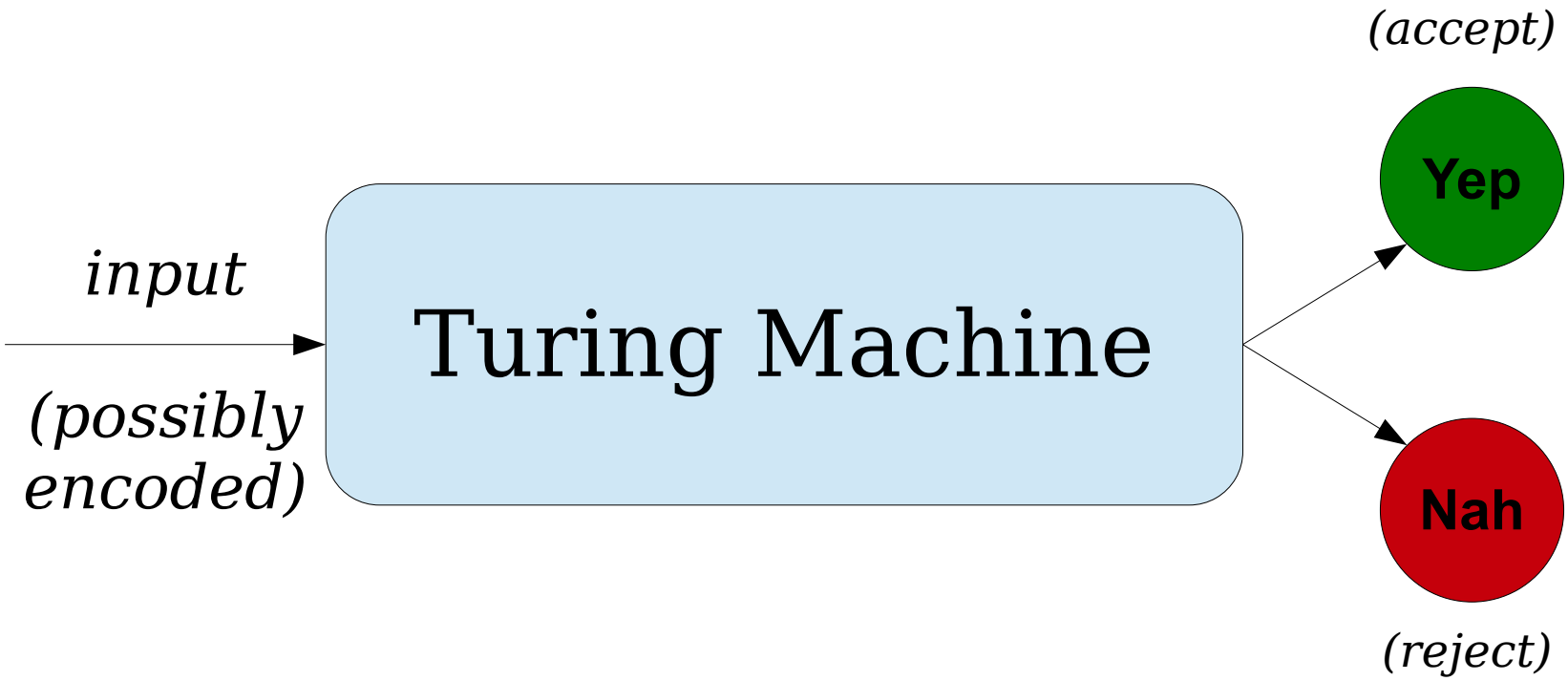

```
bool containsCat(Picture P) {

    // … do something …

}
```

A Model for Solving Problems




```
bool isLinkageGraph(Graph G) {

    // … do something …

}
```

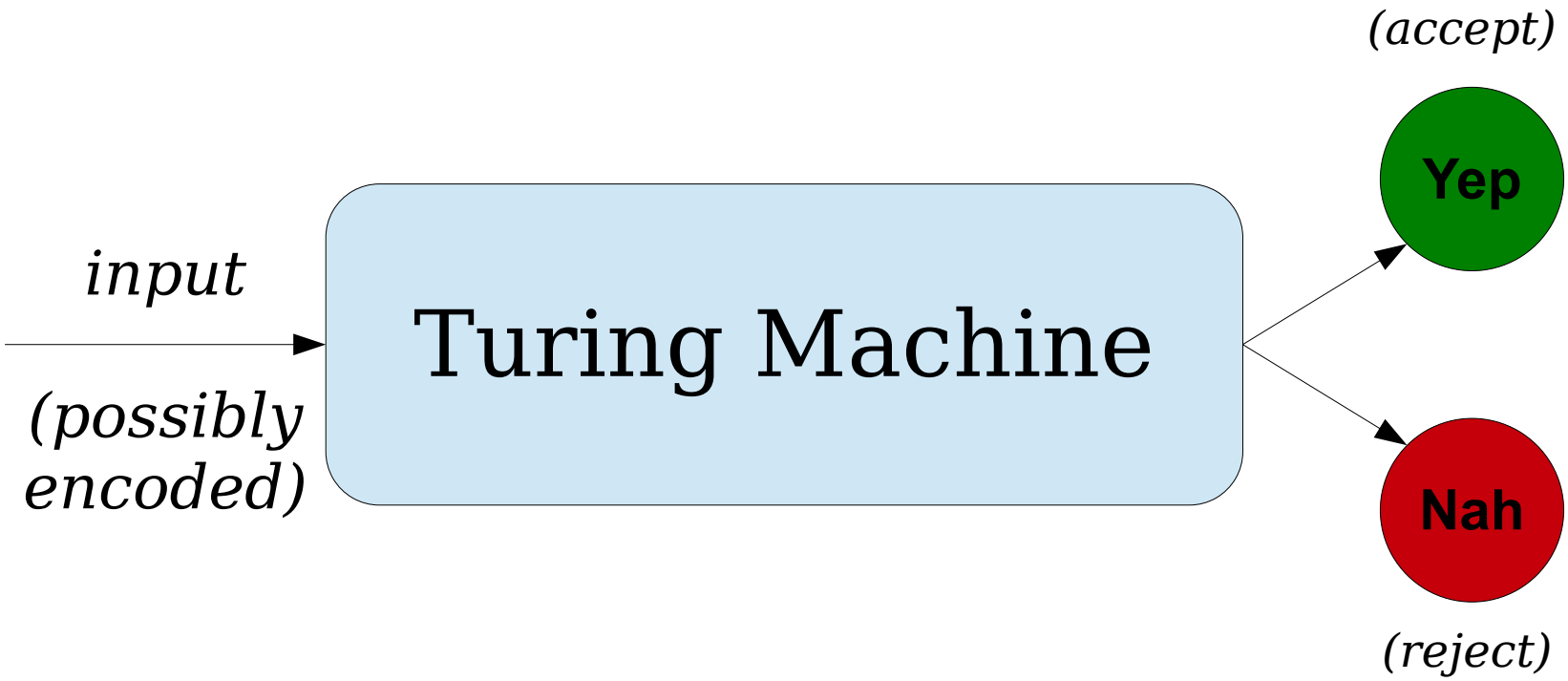
A Model for Solving Problems




```
bool isDominatingSet(Graph G, Set D) {

    // … do something …

}
```

How does this match our model?

A Model for Solving Problems



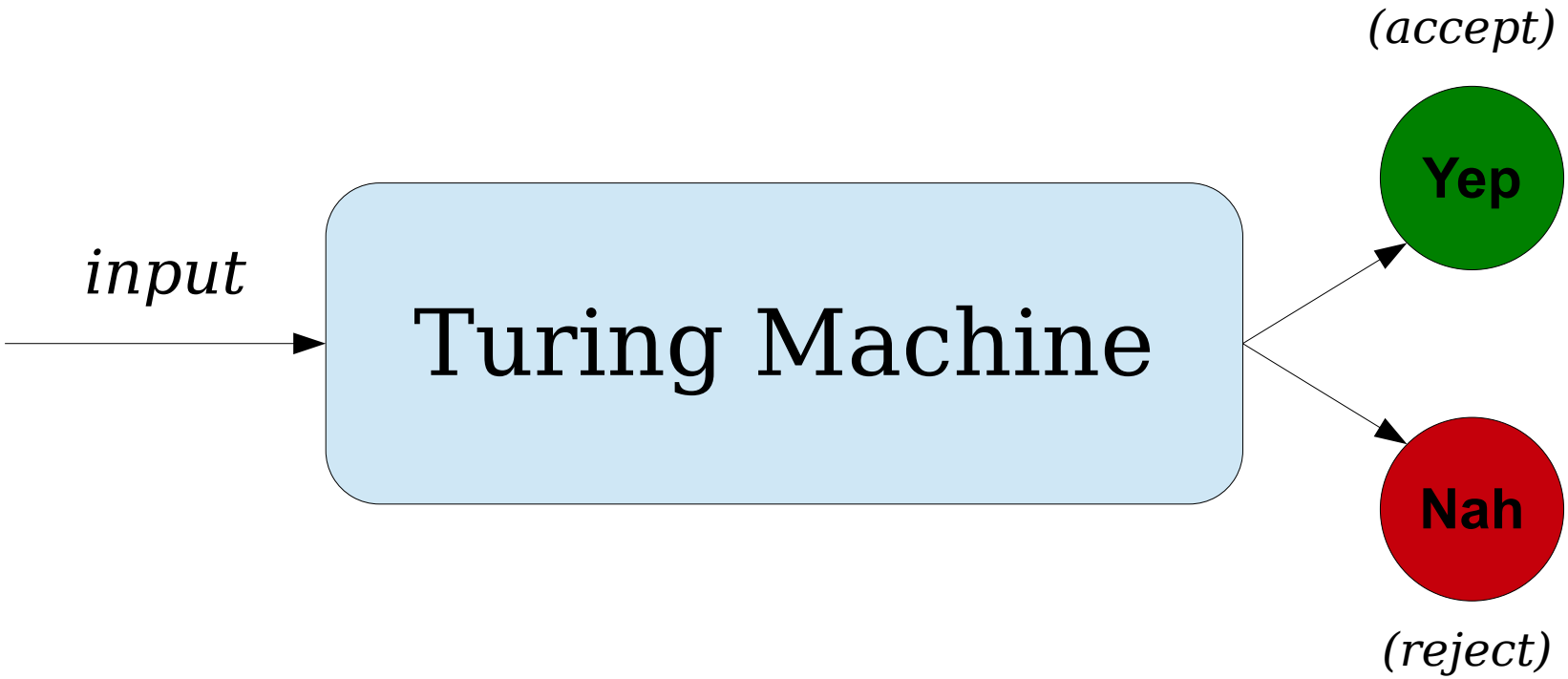

```
bool matchesRegex(string w, Regex R) {

    // … do something …

}
```

How does this match our model?

Encoding Groups of Objects

- Given a group of objects Obj_1, Obj_2, …, Obj_n, we can create a single string encoding all these objects.
  - ***Intuition 1:*** Think of it like a .zip file, but without the compression.
  - ***Intuition 2:*** Think of it like a tuple or `struct`.
- We'll denote the encoding of all of these objects as a single string by $\langle Obj_1, \ldots, Obj_n \rangle$.

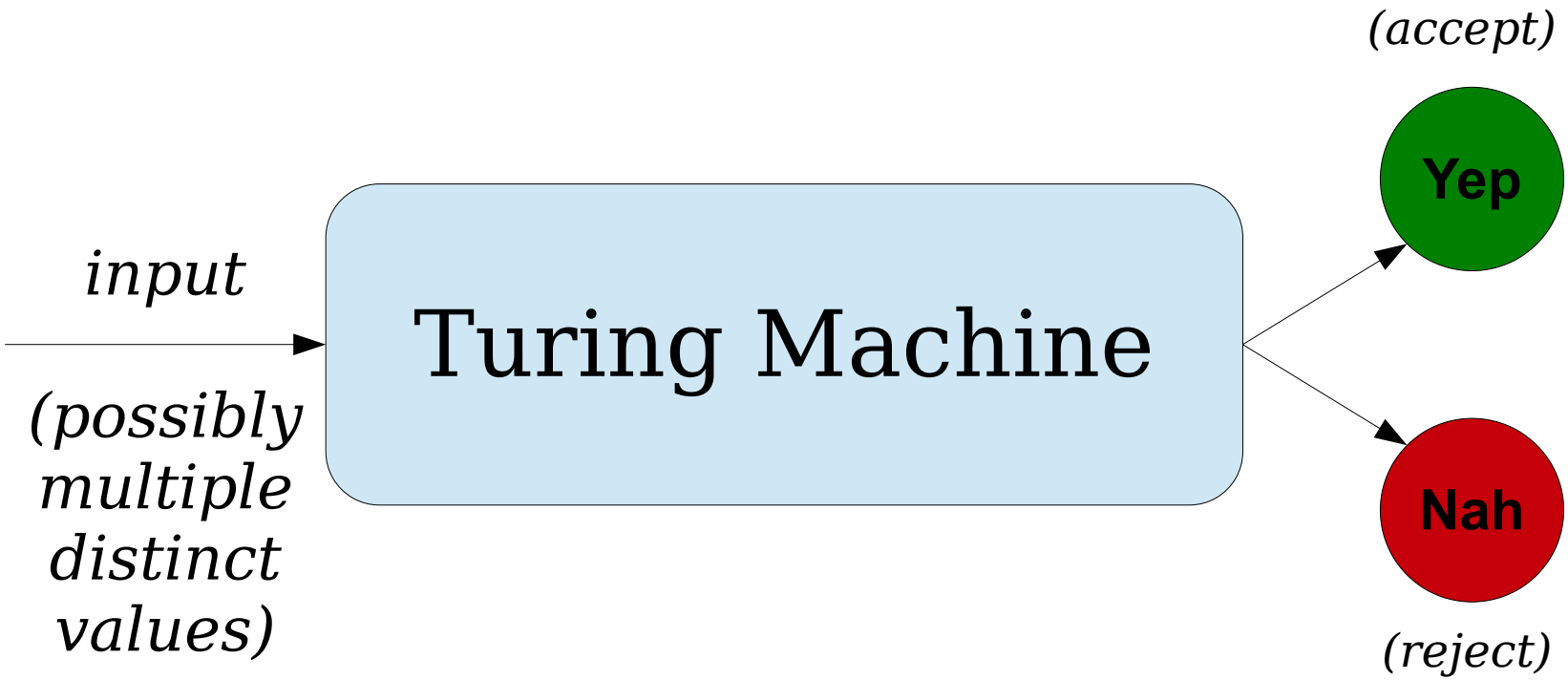
A Model for Solving Problems




```
bool matchesRegex(string w, Regex R) {

    // … do something …

}
```

These form one large bitstring.

A Model for Solving Problems




```
bool matchesRegex(string w, Regex R) {

    // … do something …

}
```

These form one large bitstring.

What problems can we solve with a computer?

Emergent Properties

Emergent Properties

- An ***emergent property*** of a system is a property that arises out of smaller pieces that doesn't seem to exist in any of the individual pieces.
- Examples:
  - Individual neurons work by firing in response to particular combinations of inputs. Somehow, this leads to consciousness, love, and ennui.
  - Individual atoms obey the laws of quantum mechanics and just interact with other atoms. Somehow, it's possible to combine them together to make iPhones and pumpkin pie.

Emergent Properties of Computation

- All computing systems equal to Turing machines exhibit several surprising emergent properties.
- If we believe the Church-Turing thesis, these emergent properties are, in a sense, “inherent” to computation. Computation can’t exist without them.
- These emergent properties are what ultimately make computation so interesting and so powerful.
- As we'll see, though, they're also computation's Achilles heel – they're how we find concrete examples of impossible problems.

Two Emergent Properties

- There are two key emergent properties of computation that we will discuss:
  - ***Universality***: There is a single computing device capable of performing any computation.
  - ***Self-Reference***: Computing devices can ask questions about their own behavior.
- As you'll see, the combination of these properties leads to simple examples of impossible problems and elegant proofs of impossibility.

Universal Machines

An Observation

- Think about how you interact with your physical computer.
  - You have a single, physical computer.
  - That computer then runs multiple programs.
- Contrast that with how we've worked with TMs.
  - We have a TM for $\{ \mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N} \}$. That TM will always perform that calculation and never do anything else.
  - We have a TM for the hailstone sequence. That TM can't compose poetry, write music, etc.
- How do we reconcile this difference?

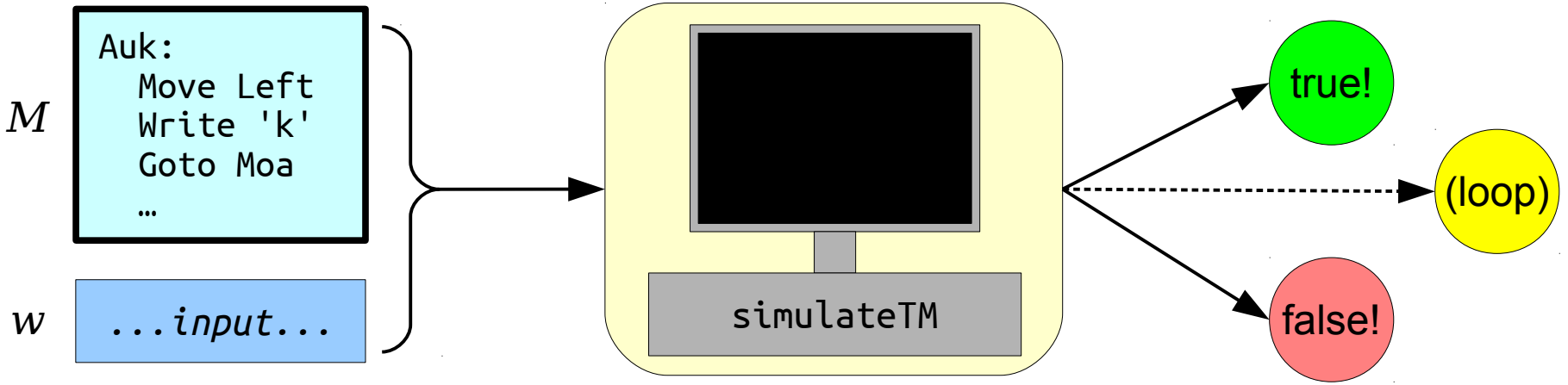
Can we make a “reprogrammable Turing machine?”

A TM Simulator

- It is possible to program a TM simulator on an unbounded-memory computer.
  - You've seen this in class.
- We could imagine it as a method

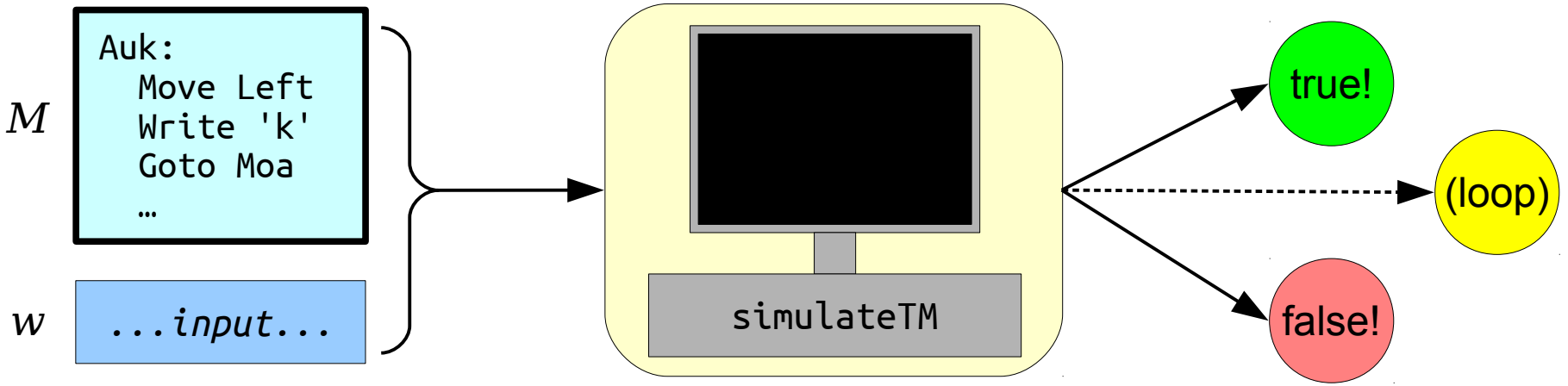
```
bool simulateTM(TM M, string w)
```

  with the following behavior:
  - If M accepts w, then `simulateTM(M, w)` returns **`true`**.
  - If M rejects w, then `simulateTM(M, w)` returns **`false`**.
  - If M loops on w, then `simulateTM(M, w)` loops infinitely.



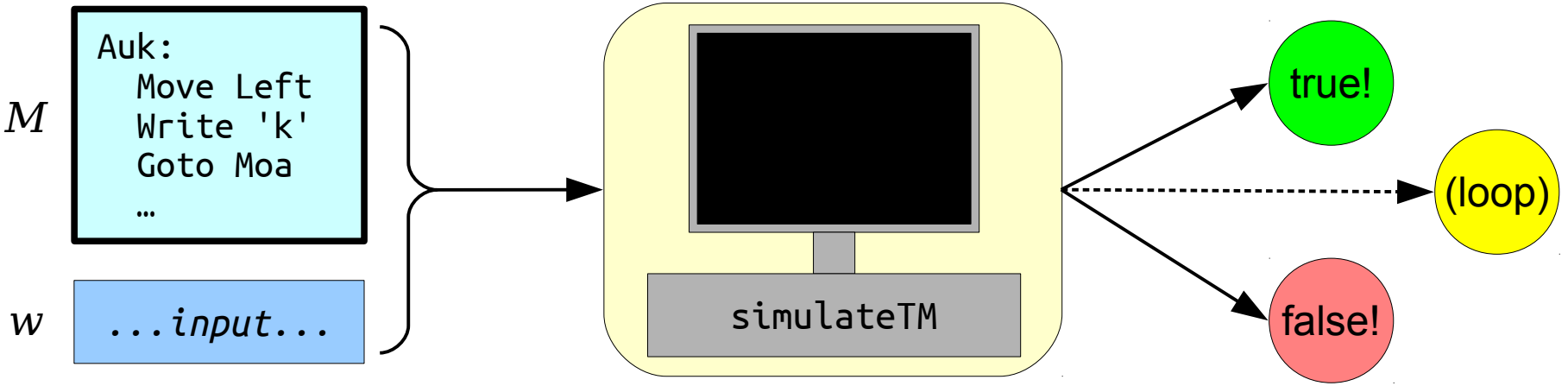
A TM Simulator

- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.



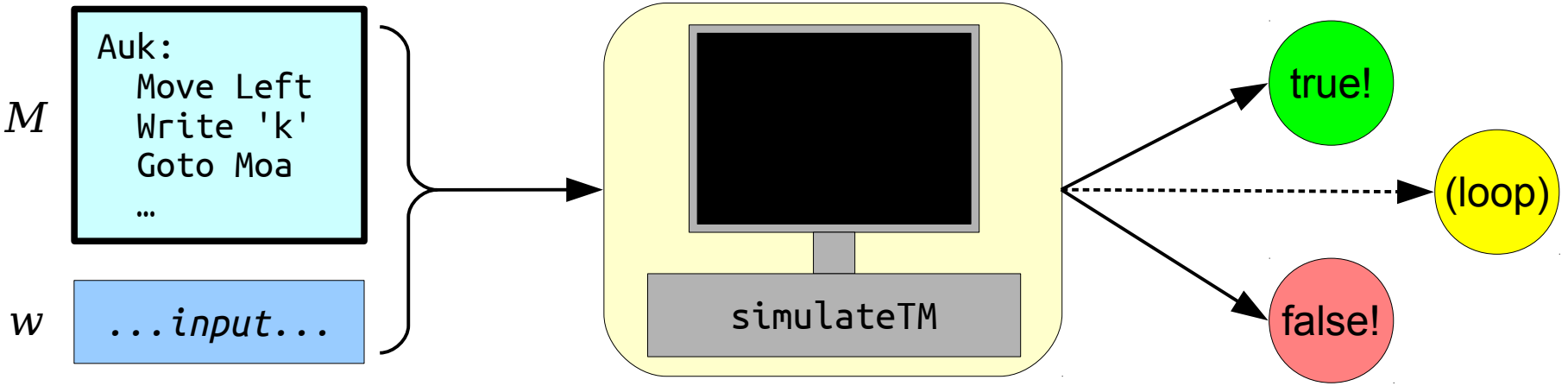
A TM Simulator

- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.
- This means that there must be some TM that has the behavior of this `simulateTM` method.



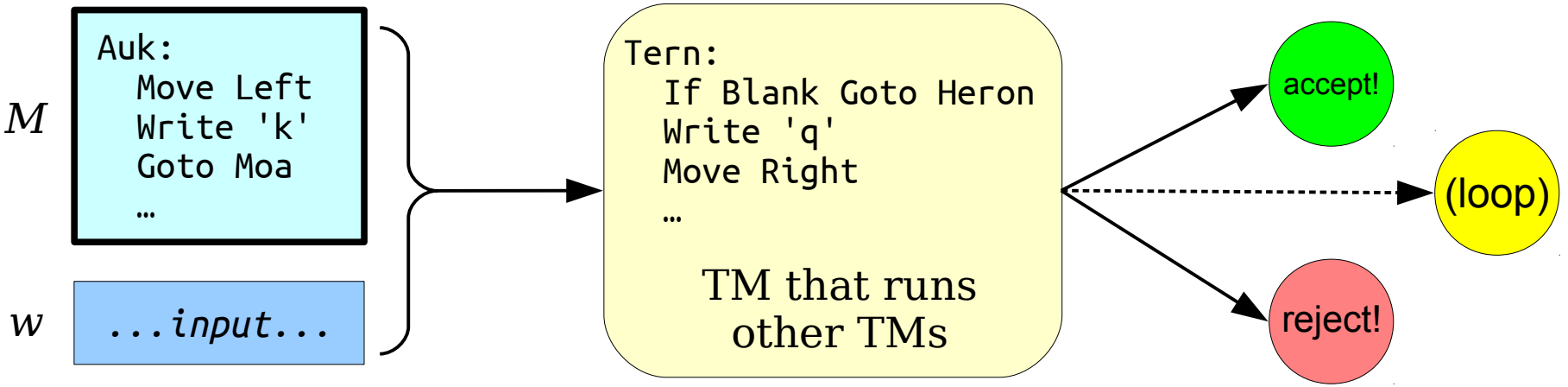
A TM Simulator

- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.
- This means that there must be some TM that has the behavior of this `simulateTM` method.
- What would that look like?



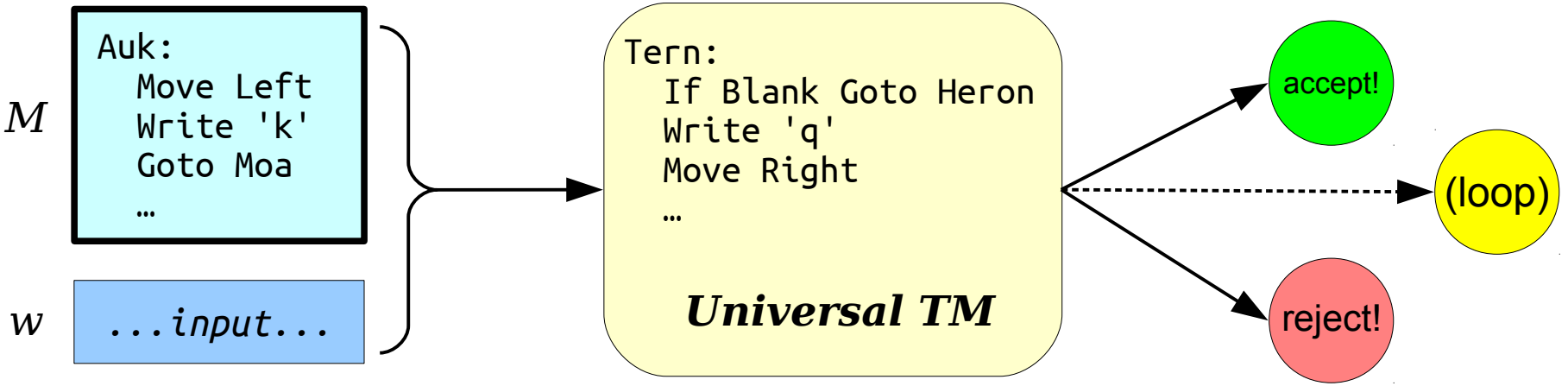
A TM Simulator

- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.
- This means that there must be some TM that has the behavior of this `simulateTM` method.
- What would that look like?



A TM Simulator

- It is known that anything that can be done with an unbounded-memory computer can be done with a TM.
- This means that there must be some TM that has the behavior of this `simulateTM` method.
- What would that look like?



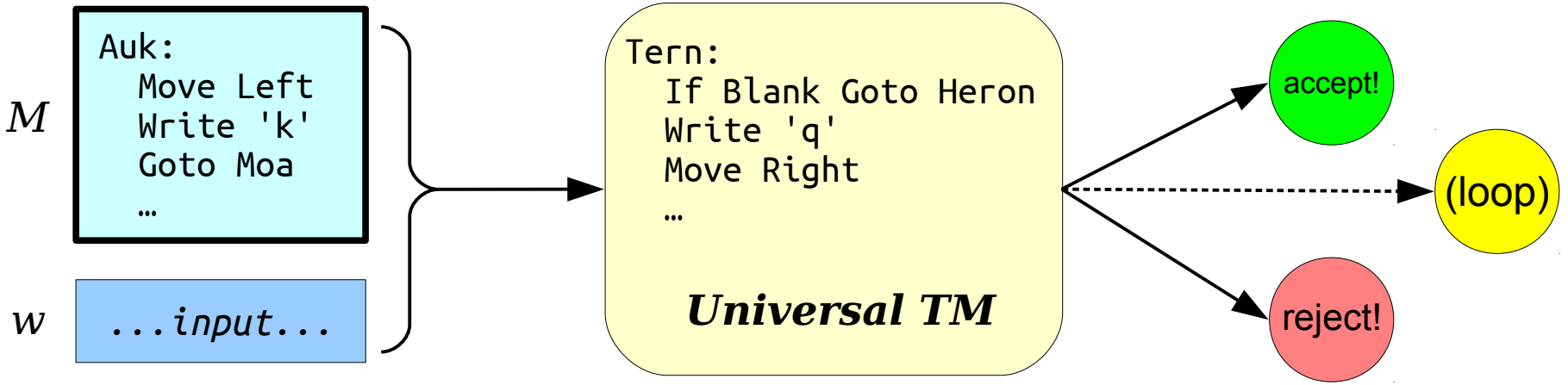
The Universal Turing Machine

- ***Theorem (Turing, 1936)***: There is a Turing machine **U_{TM}** called the ***universal Turing machine*** that, when run on an input of the form $\langle M, w \rangle$, where M is a Turing machine and w is a string, simulates M running on w and does whatever M does on w (accepts, rejects, or loops).
- The observable behavior of U_{TM} is the following:
  - If M accepts w, then U_{TM} accepts $\langle M, w \rangle$.
  - If M rejects w, then U_{TM} rejects $\langle M, w \rangle$.
  - If M loops on w, then U_{TM} loops on $\langle M, w \rangle$.

U_{TM} does to $\langle M, w \rangle$

what

M does to w.



U_{TM} as a Recognizer

- U_{TM}, when run on a string $\langle M, w \rangle$, where M is a TM and w is a string, will
  - … accept $\langle M, w \rangle$ if M accepts w,
  - … reject $\langle M, w \rangle$ if M rejects w, and
  - … loop on $\langle M, w \rangle$ if M loops on w.
- Although we didn't design U_{TM} as a recognizer, it does recognize some language.
- Which language is that?

U_{TM} as a Recognizer

- U_{TM}, when run on a string $\langle M, w \rangle$, where M is a TM and w is a string, will
  - … accept $\langle M, w \rangle$ if M accepts w,
  - … reject $\langle M, w \rangle$ if M rejects w, and
  - … loop on $\langle M, w \rangle$ if M loops on w.
- Let's let $\mathbf{A_{TM}}$ be the language recognized by the universal TM U_{TM}. This means that

$$\forall x \in \Sigma^*. (U_{TM} \text{ accepts } x \leftrightarrow x \in A_{TM})$$

U_{TM} as a Recognizer

- U_{TM}, when run on a string $\langle M, w \rangle$, where M is a TM and w is a string, will
  - … accept $\langle M, w \rangle$ if M accepts w,
  - … reject $\langle M, w \rangle$ if M rejects w, and
  - … loop on $\langle M, w \rangle$ if M loops on w.
- Let's let $\mathbf{A_{TM}}$ be the language recognized by the universal TM U_{TM}. This means that

$$\forall M. \forall w \in \Sigma^*. (U_{TM} \text{ accepts } \langle M, w \rangle \leftrightarrow \langle M, w \rangle \in A_{TM})$$

U_{TM} as a Recognizer

- U_{TM}, when run on a string $\langle M, w \rangle$, where M is a TM and w is a string, will
  - … accept $\langle M, w \rangle$ if M accepts w,
  - … reject $\langle M, w \rangle$ if M rejects w, and
  - … loop on $\langle M, w \rangle$ if M loops on w.
- Let's let $\mathbf{A_{TM}}$ be the language recognized by the universal TM U_{TM}. This means that

$$\forall M. \forall w \in \Sigma^*. (M \text{ accepts } w \leftrightarrow \langle M, w \rangle \in A_{TM})$$

- So we have

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

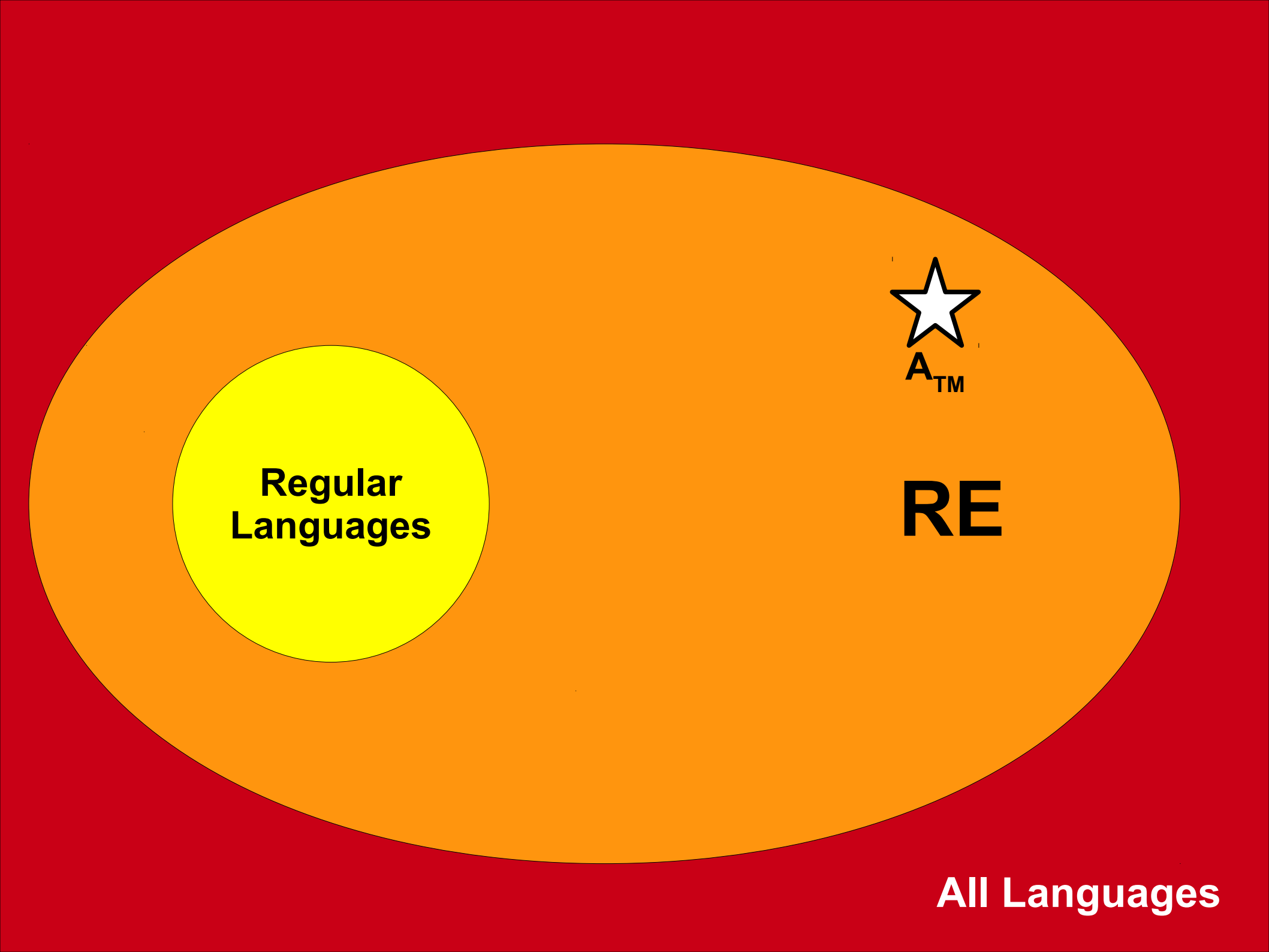
The Language A_{TM}

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

- Here's a complicated expression. Can you simplify it?

$$\langle U_{TM}, \langle M, w \rangle \rangle \in A_{TM}.$$

- Given the definition of A_{TM} and U_{TM}, the following statements are all equivalent to one another.
  - M accepts w.
  - U_{TM} accepts $\langle M, w \rangle$.
  - $\langle M, w \rangle \in A_{TM}$.

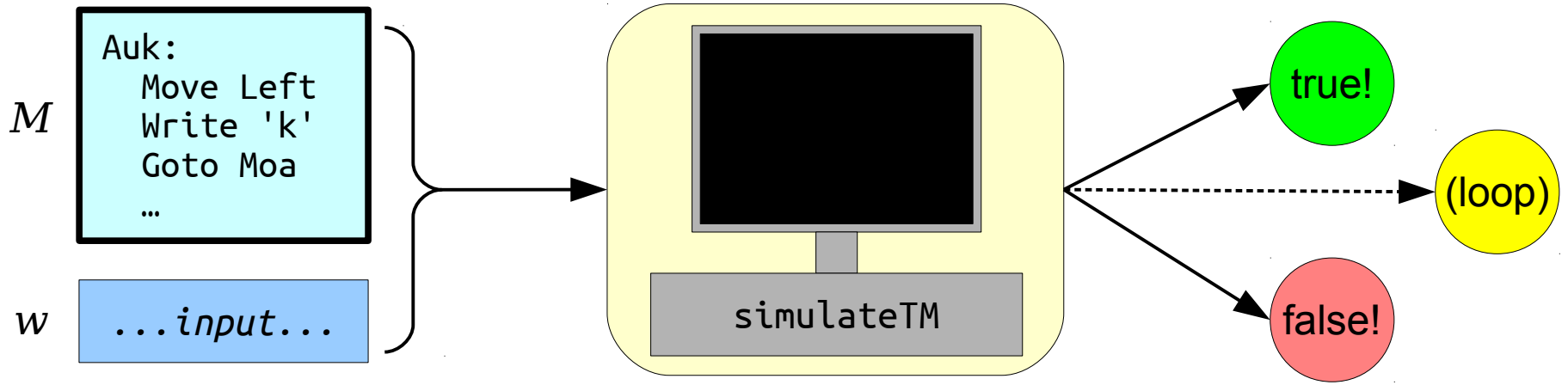

Regular
Languages
A_{TM}
RE
All Languages

Uh… so what?

Reason 1: ***It has practical consequences.***

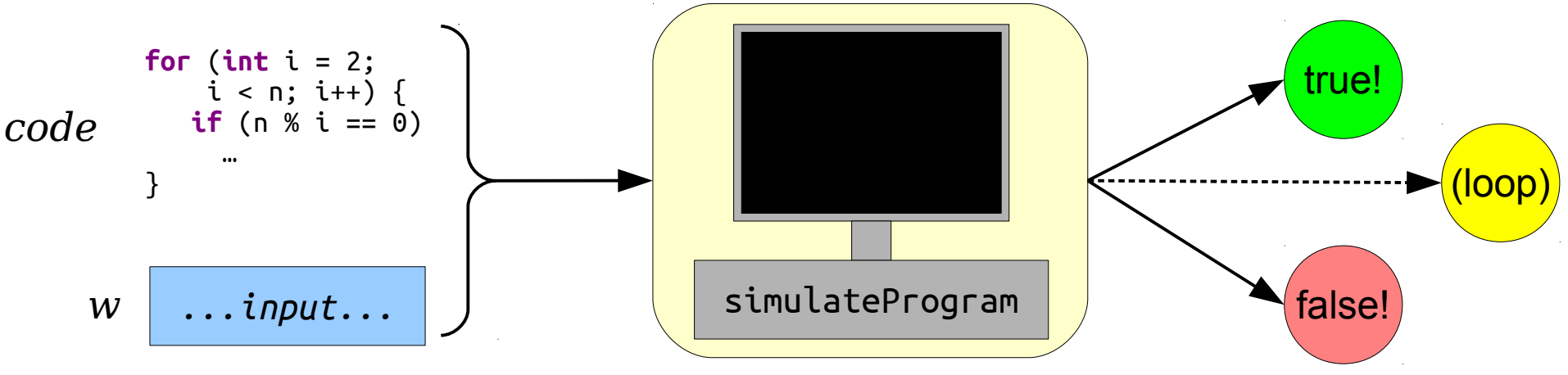
Why Does This Matter?

- The existence of a universal Turing machine has both theoretical and practical significance.
- For a practical example, let's review this diagram from before.
- Previously we replaced the *computer* with a TM. (This gave us the universal TM.)
- What happens if we replace the *TM* with a computer program?



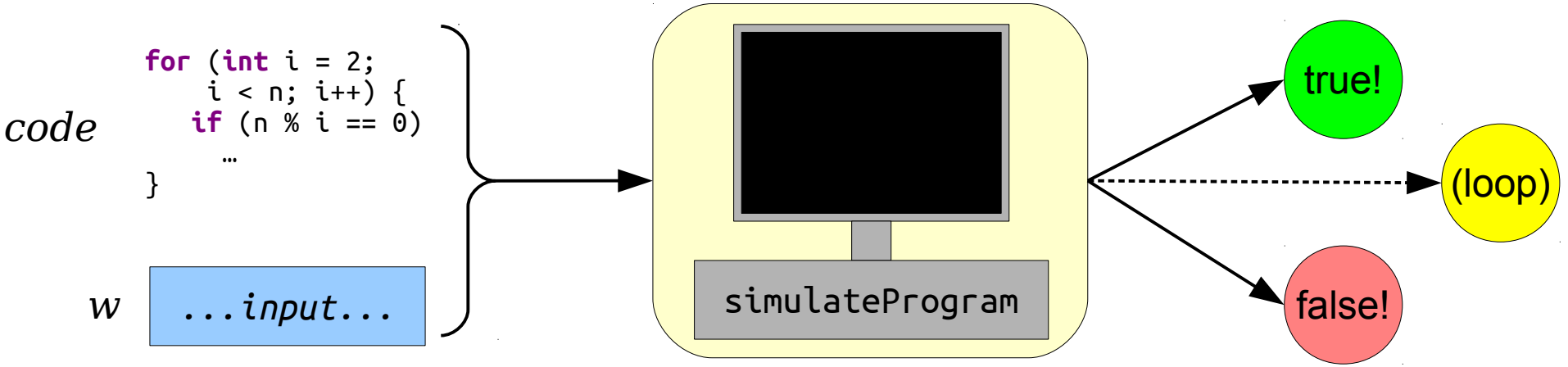
Why Does This Matter?

- The existence of a universal Turing machine has both theoretical and practical significance.
- For a practical example, let's review this diagram from before.
- Previously we replaced the *computer* with a TM. (This gave us the universal TM.)
- What happens if we replace the *TM* with a computer program?



Why Does This Matter?

- We now have a computer program that runs other computer programs!
  - An ***interpreter*** is a program that simulates other programs. Python programs are usually executed by interpreters. Your web browser interprets JavaScript code when it visits websites.
  - A ***virtual machine*** is a program that simulates an entire operating system. Virtual machines are used in computer security, cloud computing, and even by individual end users.
- It's not a coincidence that this is possible – Turing's 1936 paper says that any general-purpose computing system must be able to do this!



Why Does This Matter?

- The key idea behind the universal TM is that idea that TMs can be fed as inputs into other TMs.
  - Similarly, an interpreter is a program that takes other programs as inputs.
  - Similarly, an emulator is a program that takes entire computers as inputs.
- This hits at the core idea that ***computing devices can perform computations on other computing devices.***

Reason 2: ***It's philosophically interesting.***

Can Computers Think?

- On May 15, 1951, Alan Turing delivered **a radio lecture on the BBC** on the topic of whether computers can think.
- He had the following to say about whether a computer can be thought of as an electric brain...

“In fact I think [computers] could be used in such a manner that they could be appropriately described as brains. I should also say that

‘If any machine can be appropriately described as a brain, then any digital computer can be so described.’

This last statement needs some explanation. It may appear rather startling, but with some reservations it appears to be an inescapable fact.

It can be shown to follow from a characteristic property of digital computers, which I will call their ***universality***. A digital computer is a universal machine in the sense that it can be made to replace any machine of a certain very wide class. It will not replace a bulldozer or a steam-engine or a telescope, but it will replace any rival design of calculating machine, that is to say any machine into which one can feed data and which will later print out results. In order to arrange for our computer to imitate a given machine it is only necessary to program the computer to calculate what the machine in question would do under given circumstances, and in particular what answers it would print out. The computer can then be made to print out the same answers.

If now some machine can be described as a brain we have only to program our digital computer to imitate it and it will also be a brain.”

Self-Referential Software

Self-Referential Programs

- If TMs can take other TMs as input, could they take themselves as input?

YES.

- TMs can take their own code as input, and ask questions about (or even execute!) their own code.
- In fact, any computing system that's equal in power to a Turing machine possesses some mechanism for self-reference.
- Want to see how deep the rabbit hole goes? Take CS154!

Quines

- A ***Quine*** is a special kind of self-referential program that, when run, prints its own source code.
- Believe it or not, it is possible to write such a program!
- *See zip file with lecture slides for code.*

Self-Referential Programs

- ***Claim:*** Going forward, assume that any function has the ability to get access to its own source code.
- This means we can write programs like the one shown here:

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

    return input == me;
}
```

Next Time

- ***Self-Defeating Objects***
  - Objects “too powerful” to exist.
- ***Undecidable Problems***
  - Problems truly beyond the limits of algorithmic problem-solving!
- ***Consequences of Undecidability***
  - Why does any of this matter outside of Theoryland?