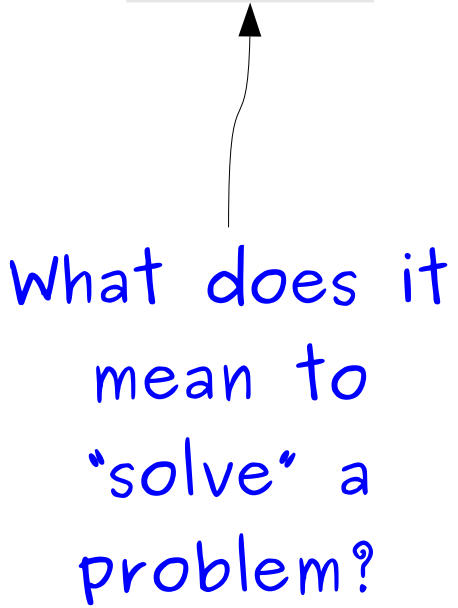
Turing Machines

Part Two

Recap from Last Time

What problems can we solve with a computer?



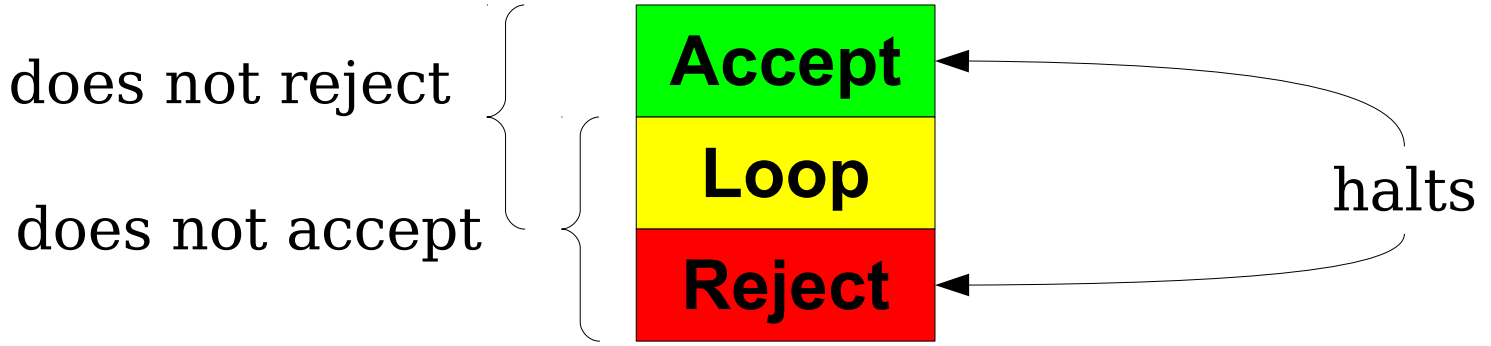
An Important Observation

- 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?

New Stuff!

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

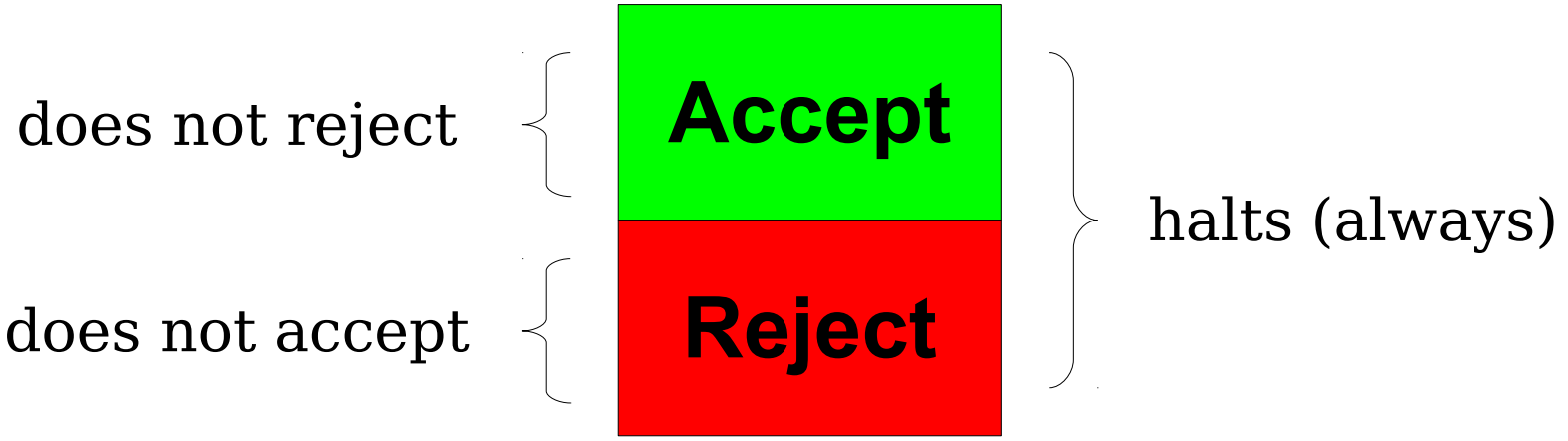
- The class **RE** consists of all recognizable languages.
- Formally speaking:

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

- 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 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.

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.

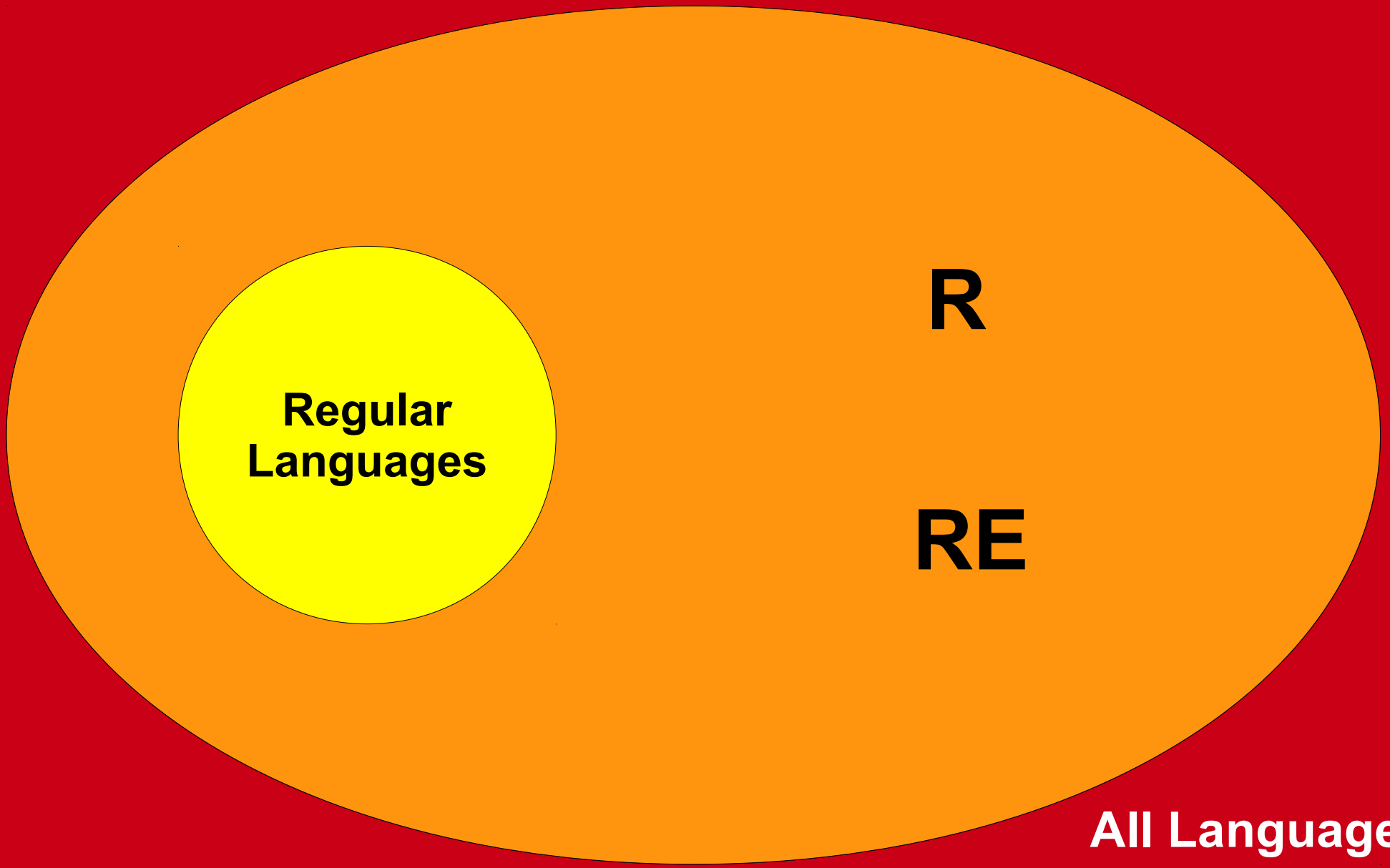
R and **RE** Languages

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

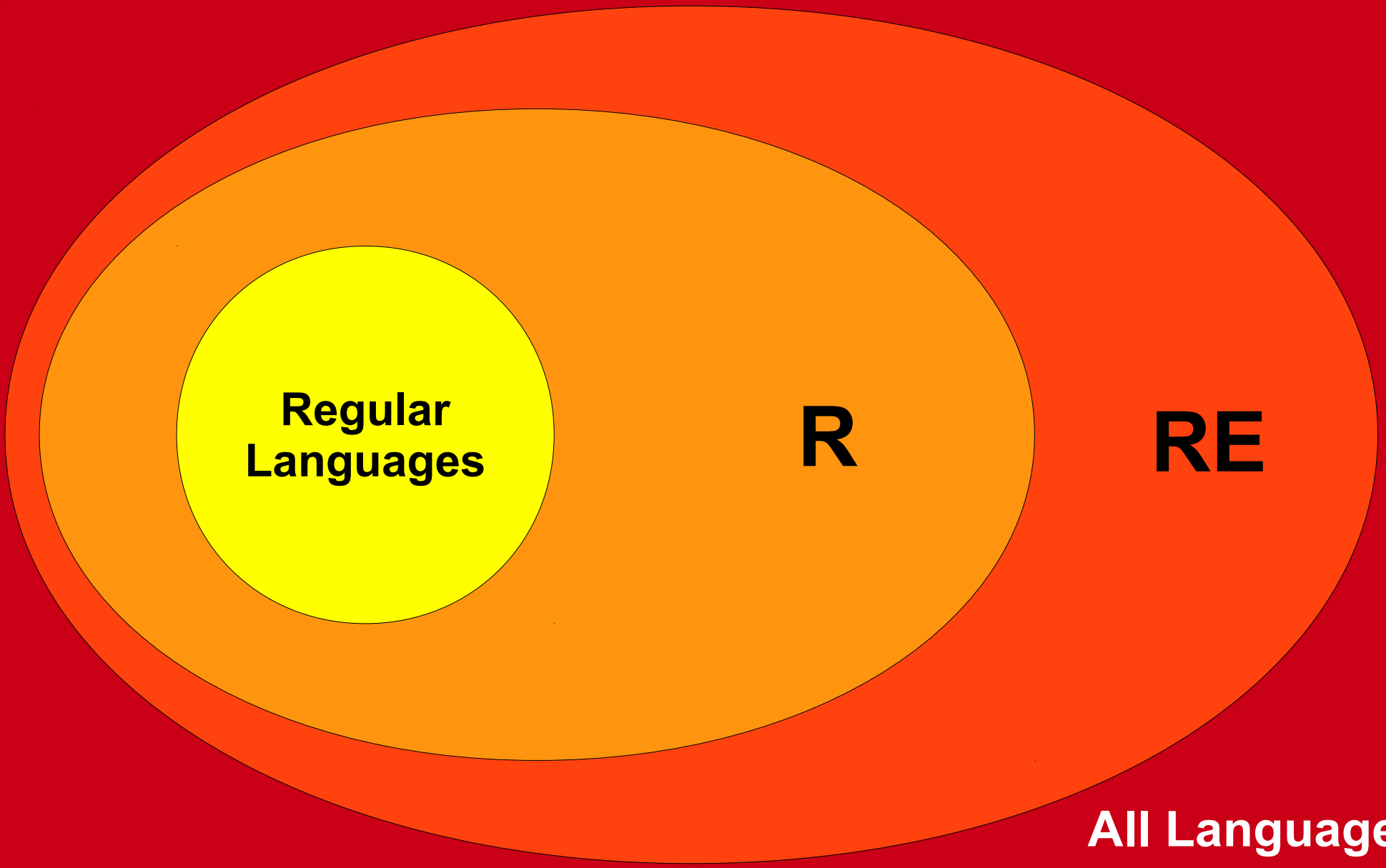
$$\mathbf{R} \stackrel{?}{=} \mathbf{RE}$$

- 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?

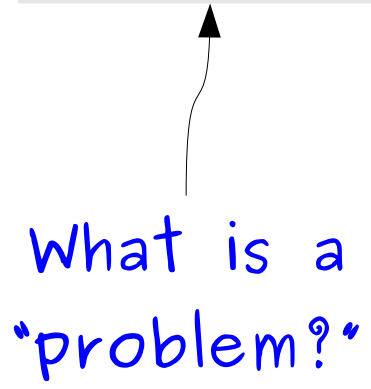


Unanswered Questions

- 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.

Strings, Languages, and Encodings

What problems can we solve with a computer?

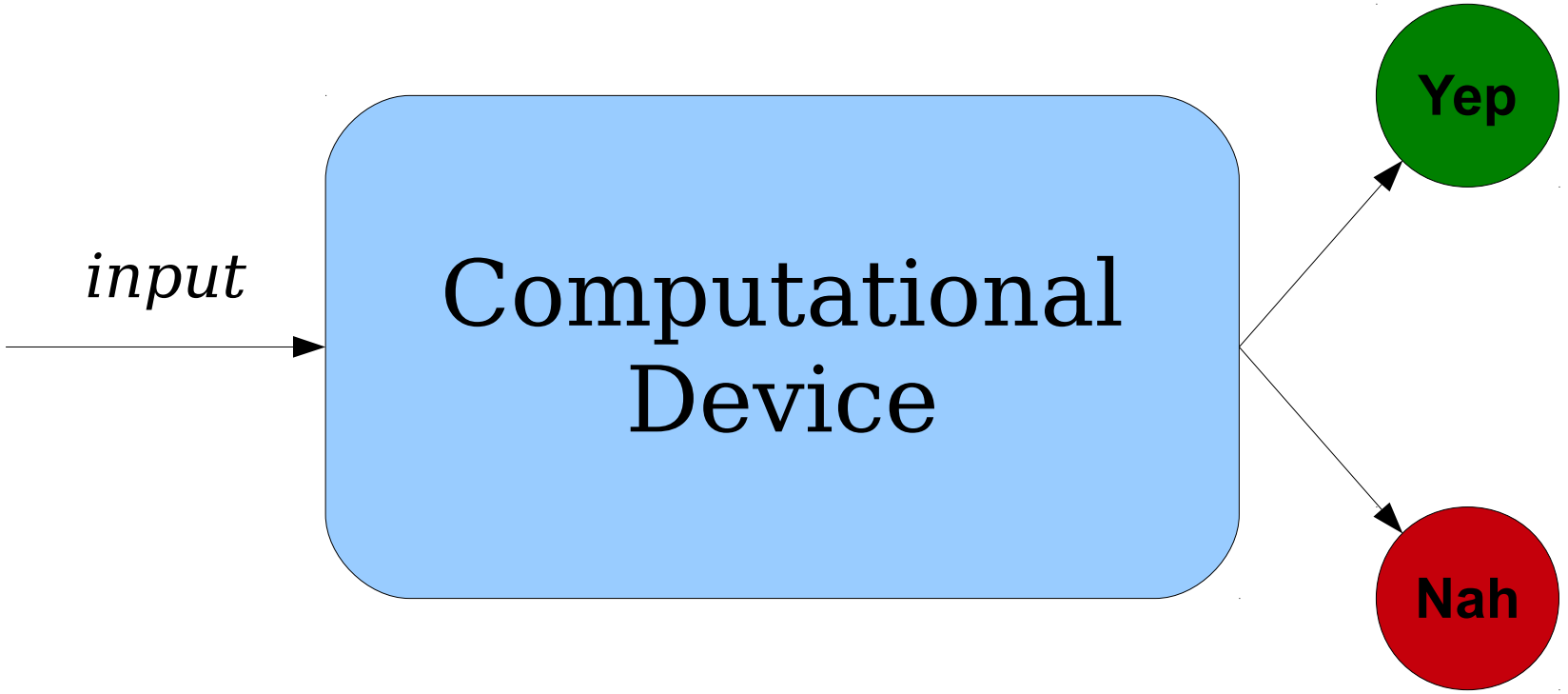


Decision Problems

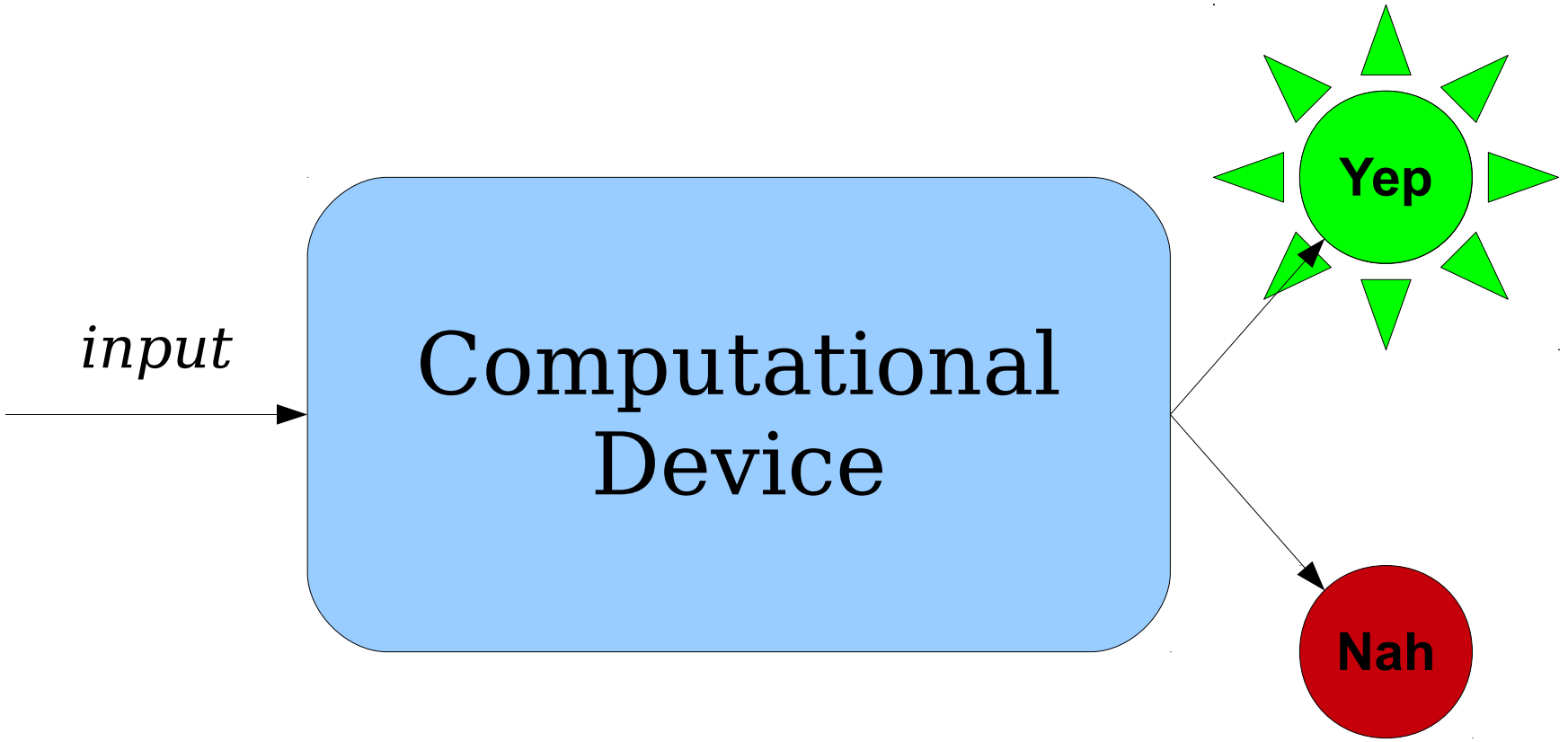
- A ***decision problem*** is a type of problem where the goal is to provide a yes or no answer.
- Example: Dominating Set Problem

  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?

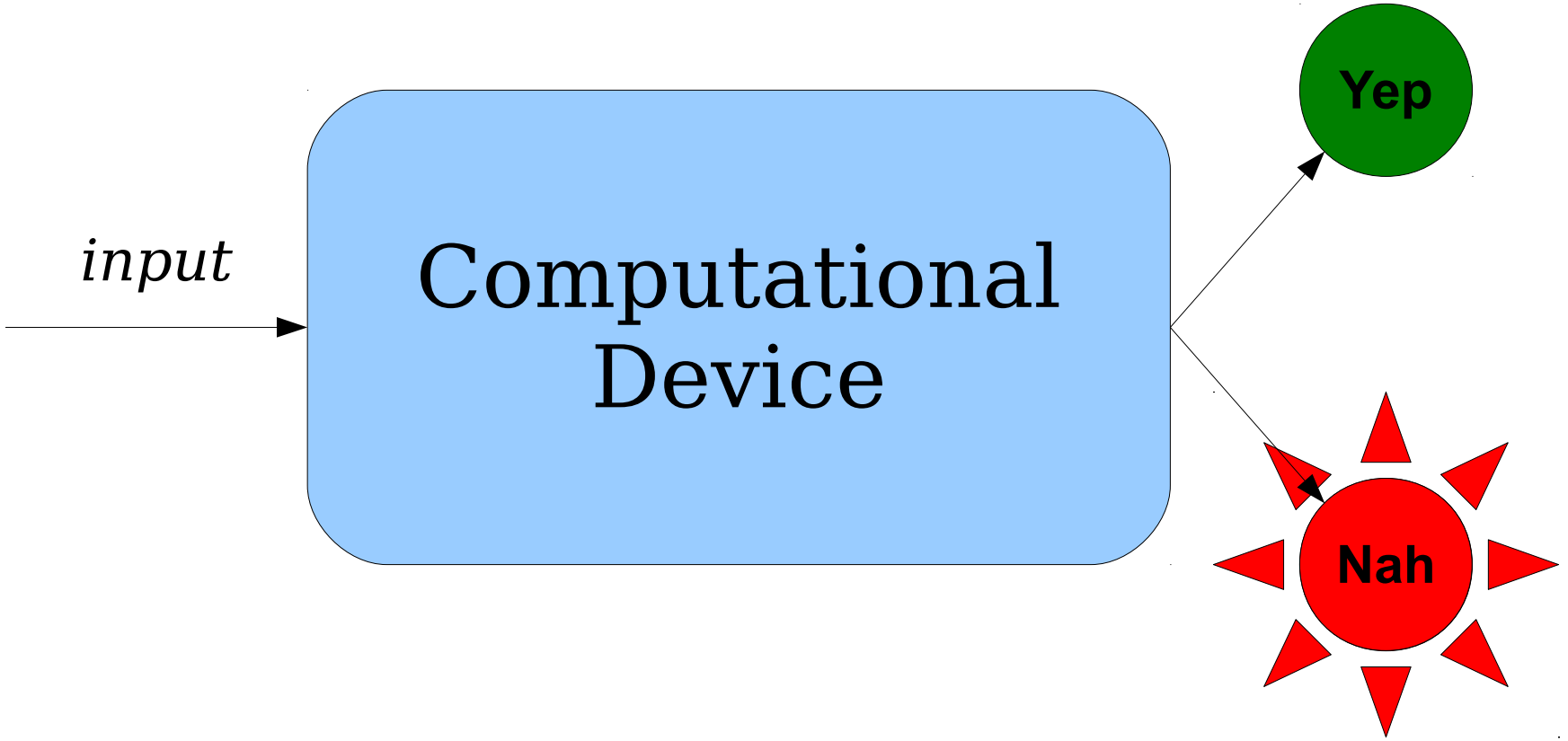
A Model for Solving Problems



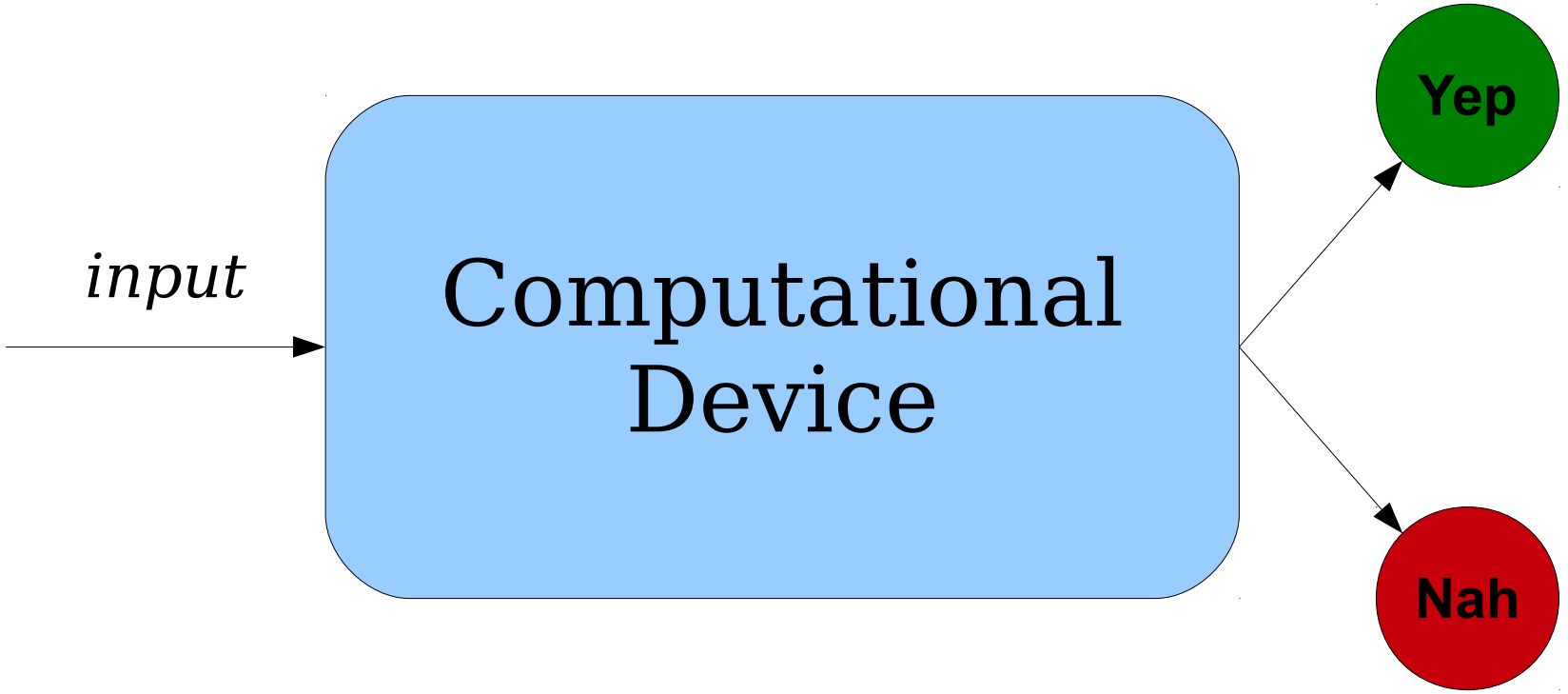
A Model for Solving Problems



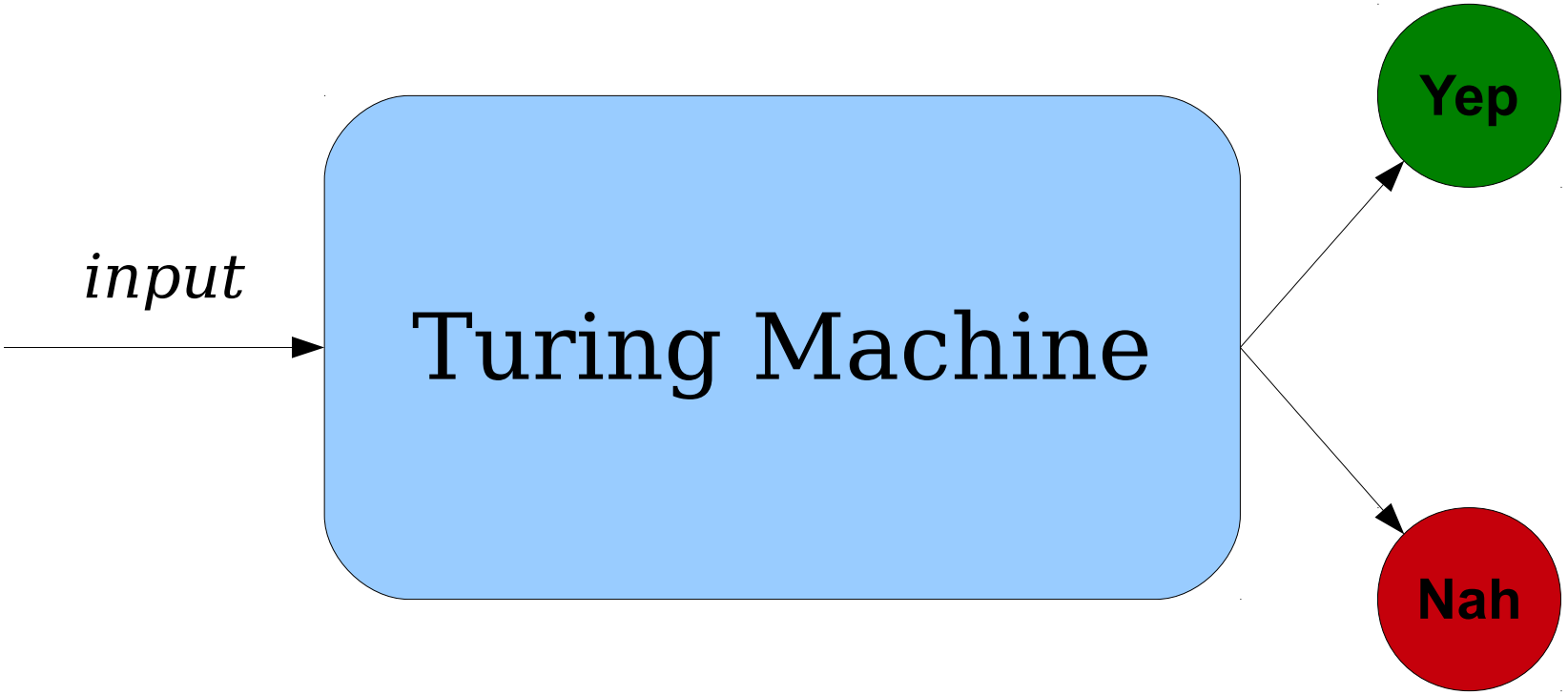
A Model for Solving Problems



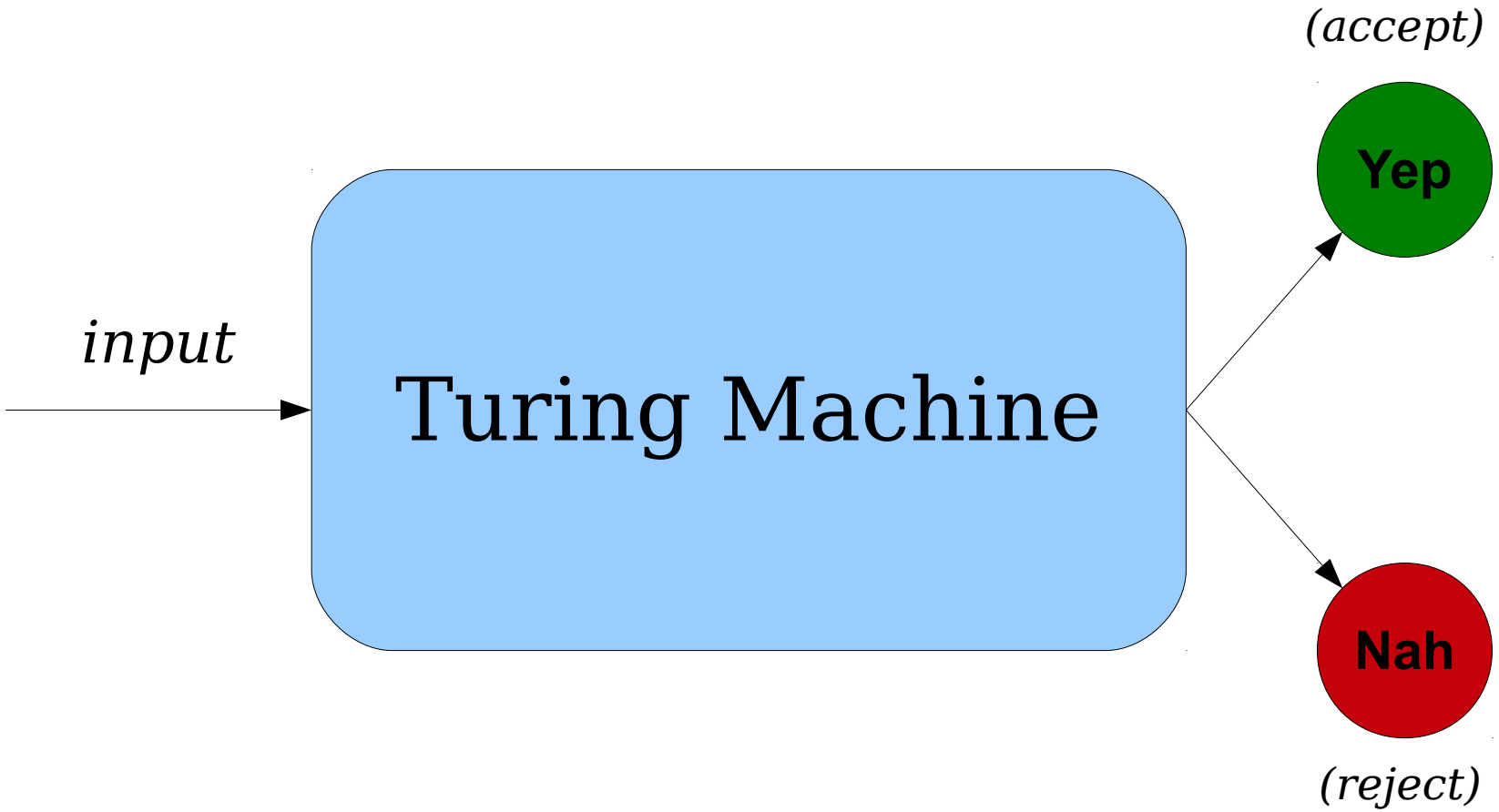
A Model for Solving Problems



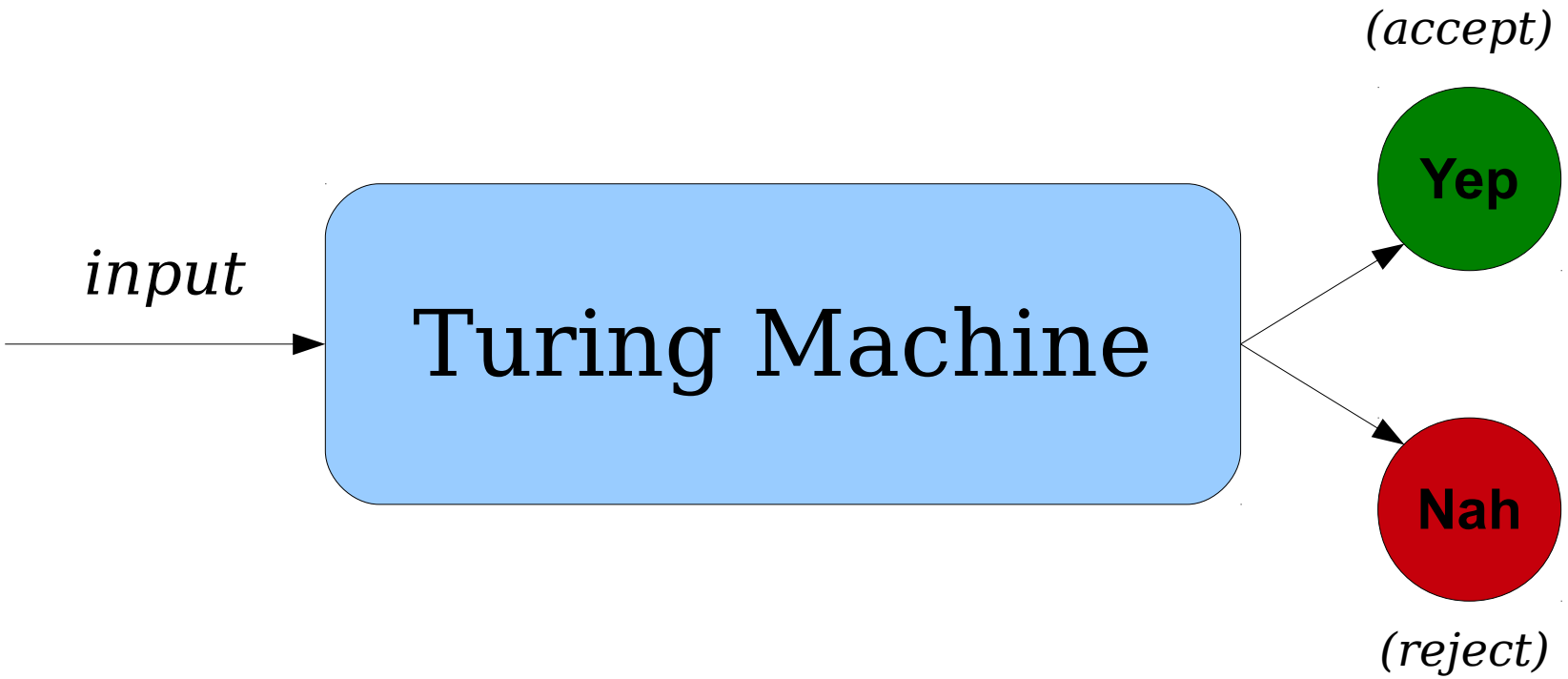
A Model for Solving Problems



A Model for Solving Problems

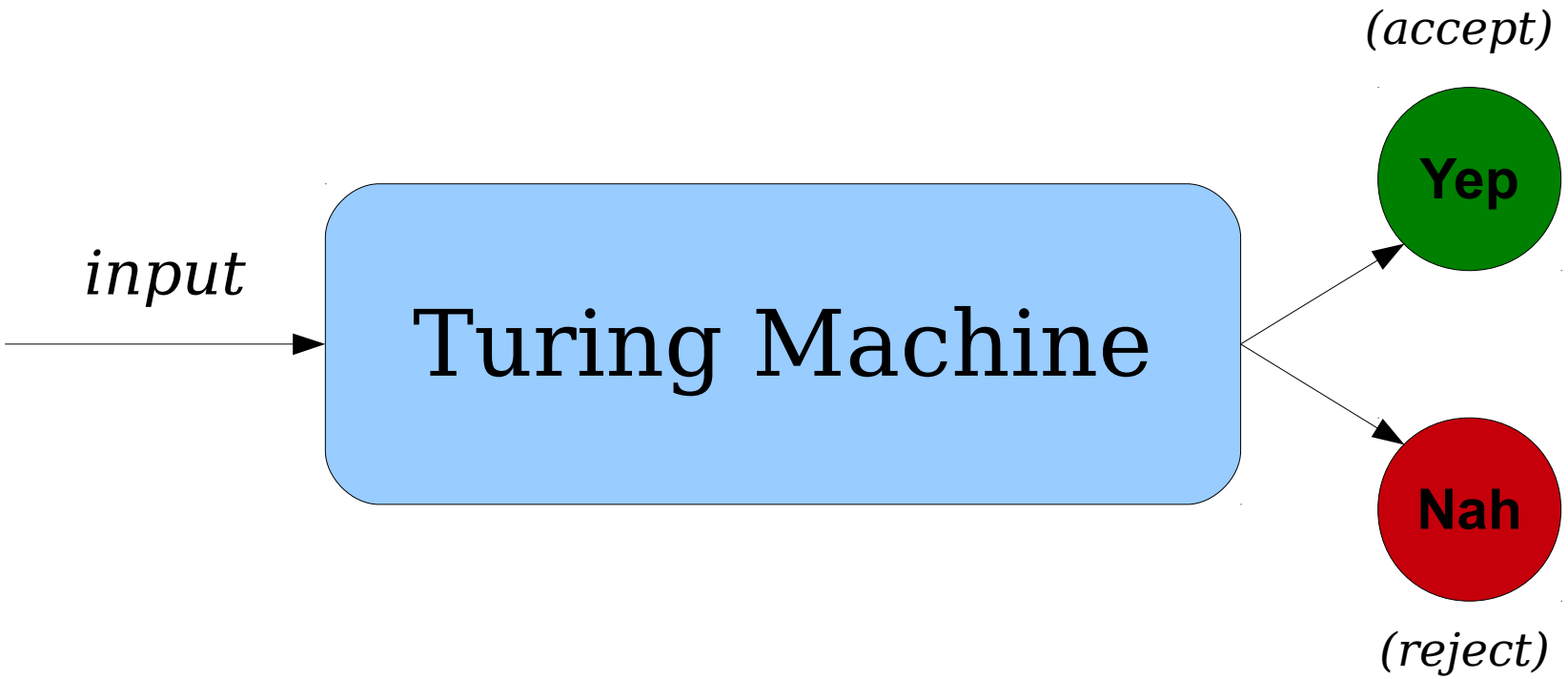


A Model for Solving Problems



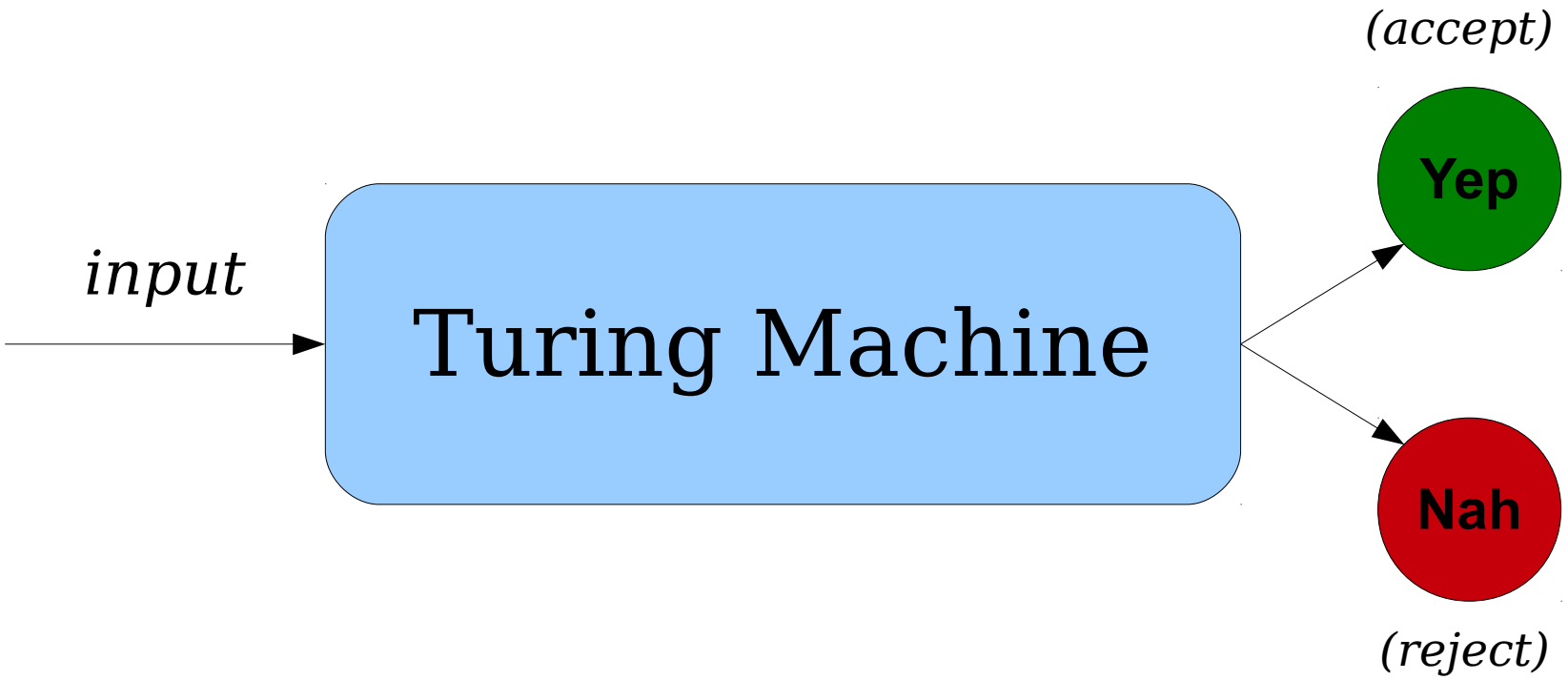

```
bool someFunctionName(string input) {

    // … do something …

}
```

A Model for Solving Problems




```
bool isAnBn(string input) {

    // … do something …

}
```

A Model for Solving Problems




```
bool 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

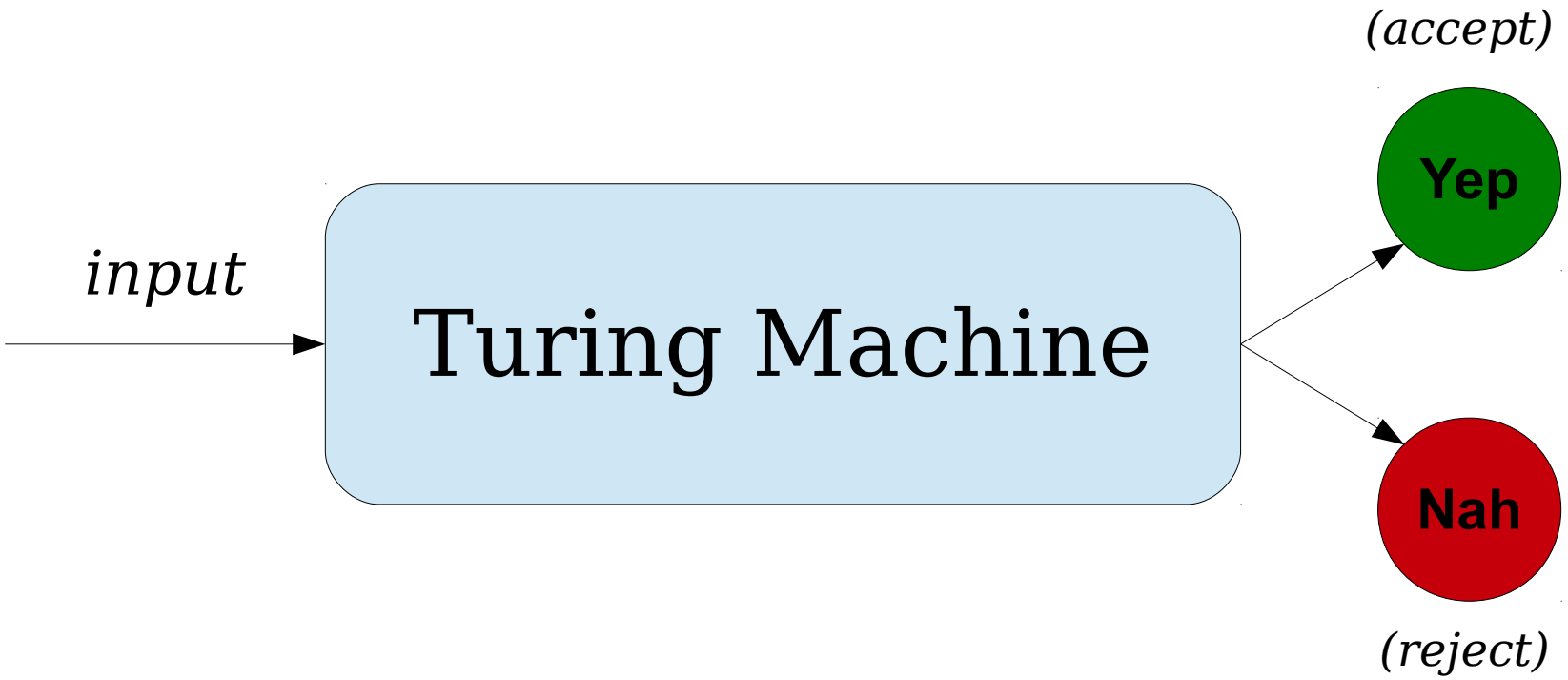
- If *Obj* is some mathematical object that is *discrete* and *finite*, then we'll use the notation ***⟨Obj⟩*** to refer to some way of encoding that object as a string.
- Think of ⟨*Obj*⟩ like a file on disk – it encodes some high-level object as a series of characters.

⟨  ⟩ = 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 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.

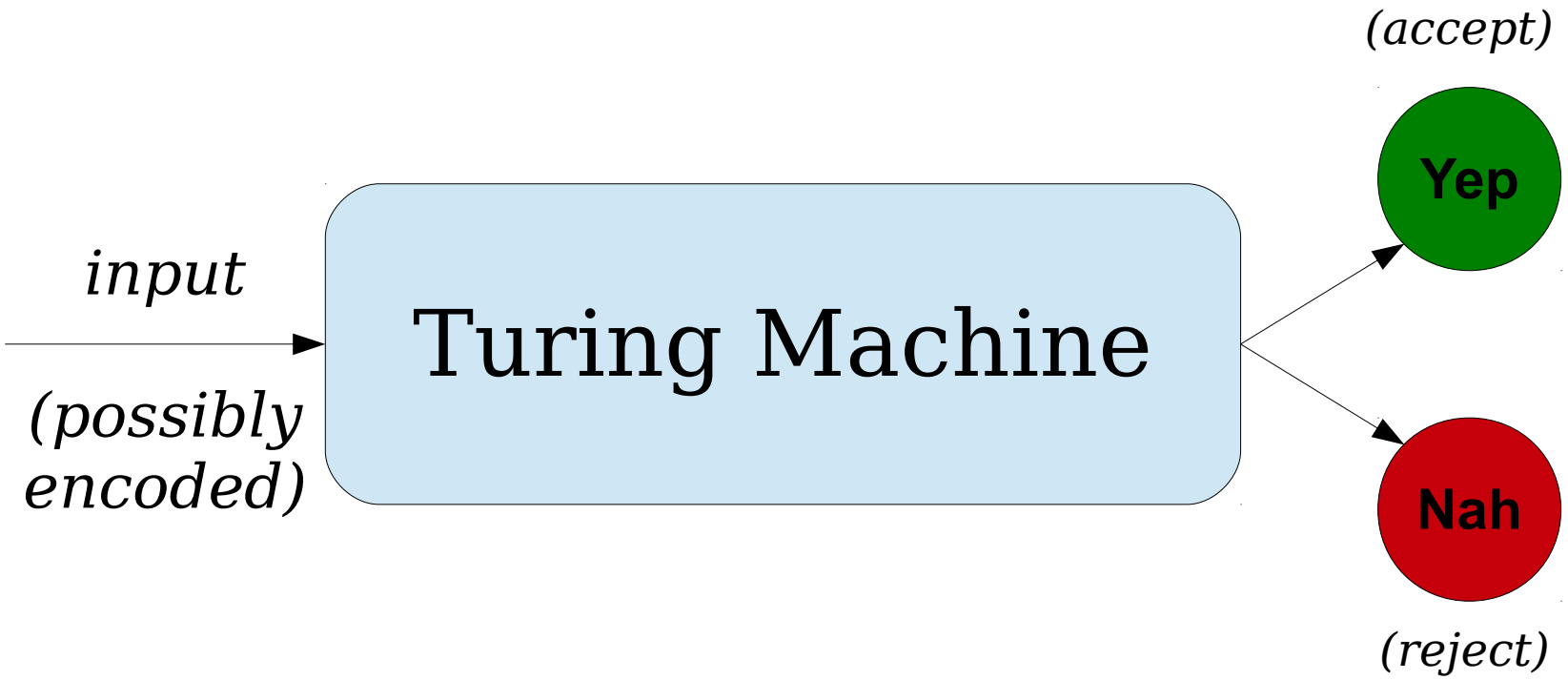
A Model for Solving Problems




```
bool containsCat(Picture P) {
    // … do something …
}
```

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

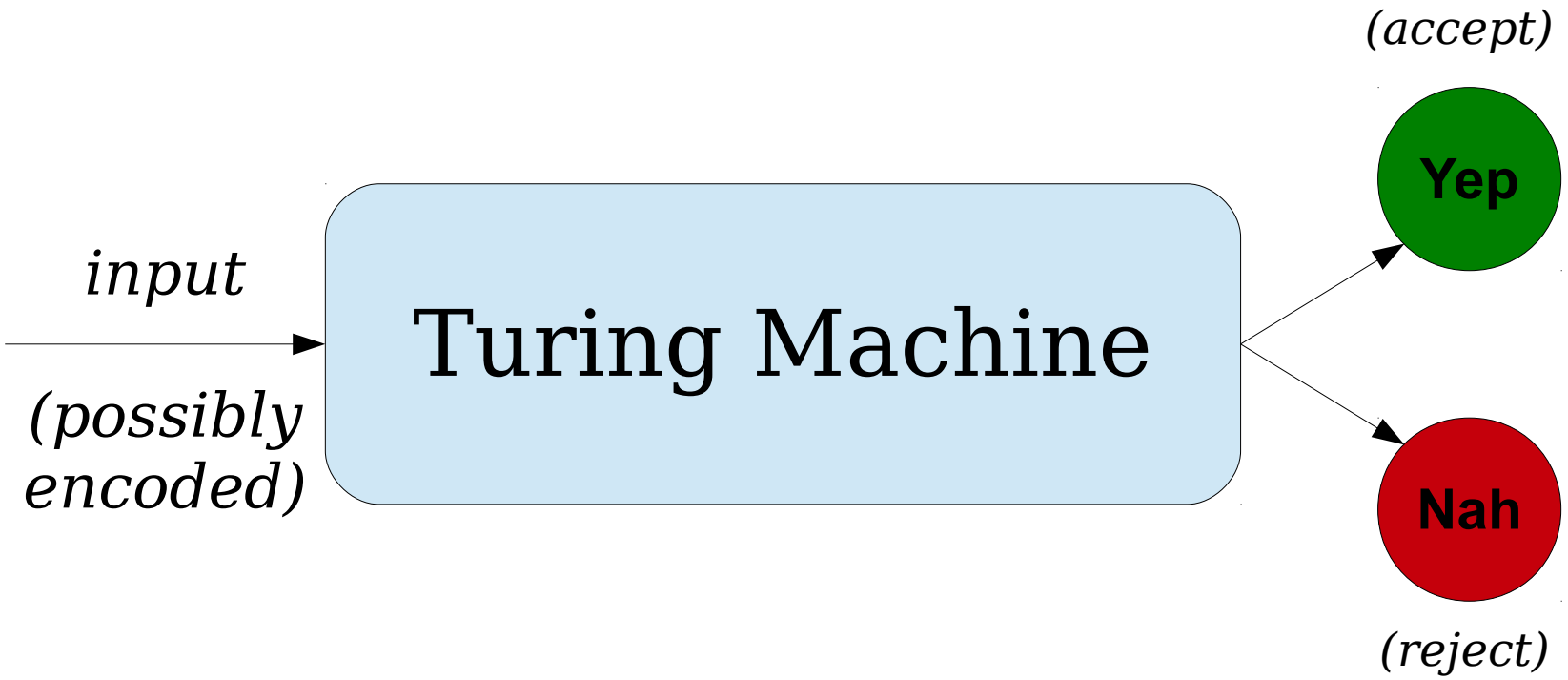
A Model for Solving Problems




```
bool containsCat(Picture P) {

    // … do something …

}
```

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

A Model for Solving Problems



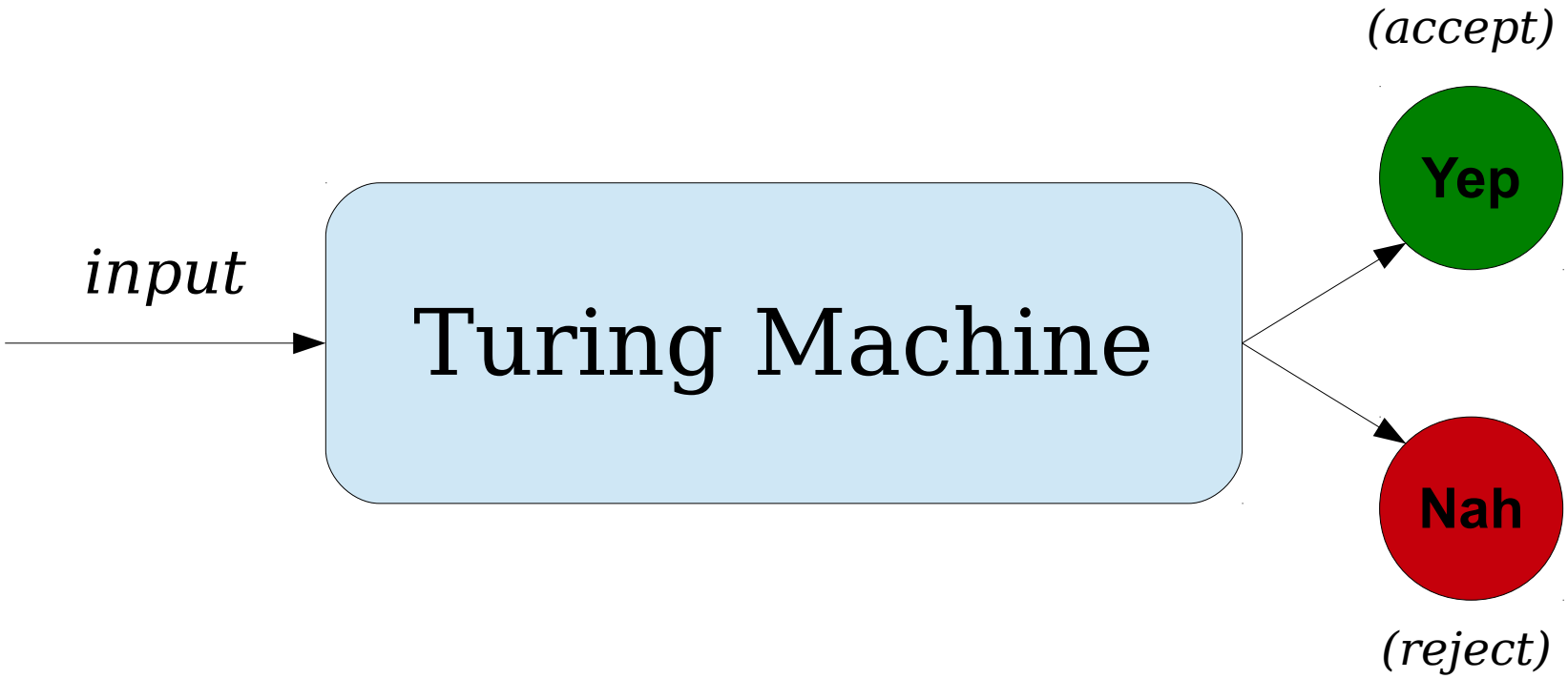

```
bool matchesRegex(string w, Regex R) {

    // … 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 $\langle Obj_1, \ldots, Obj_n \rangle$.

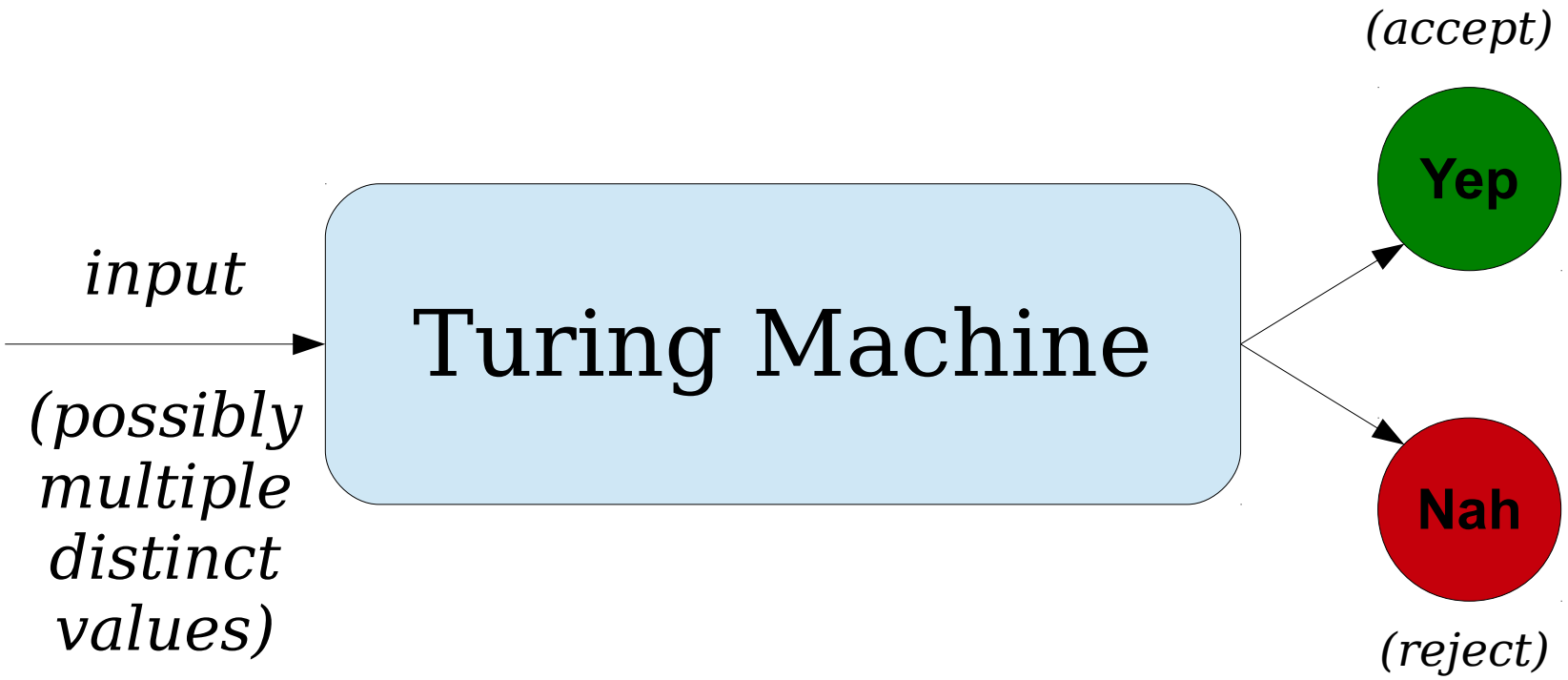
A Model for Solving Problems




```
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 $\{ \mathbf{a}^n\mathbf{b}^n \mid n \in \mathbb{N} \}$. 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?

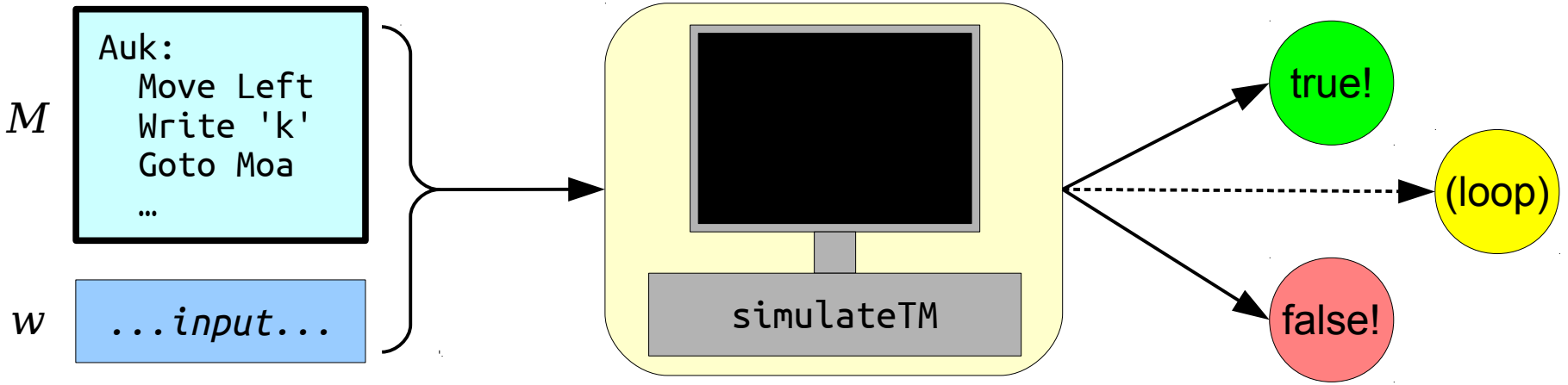
Can we make a “reprogrammable Turing machine?”

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

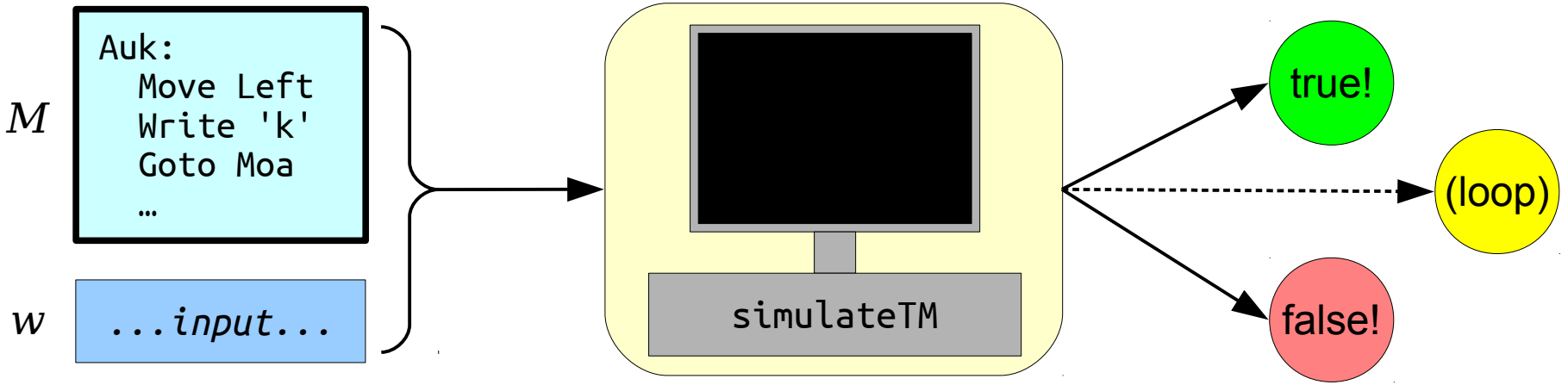
```
bool simulateTM(TM M, string w)
```

  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.



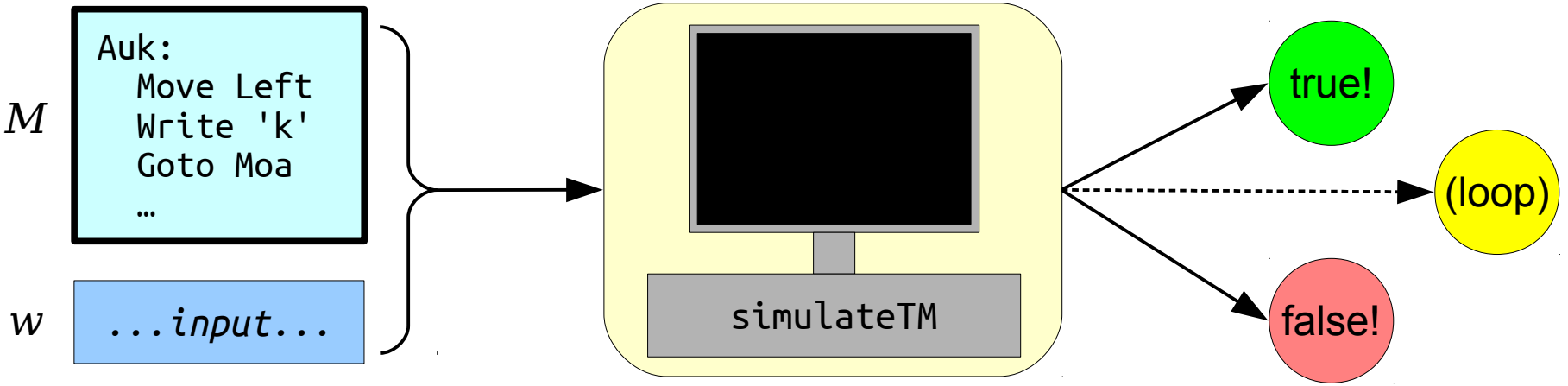
A TM Simulator

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



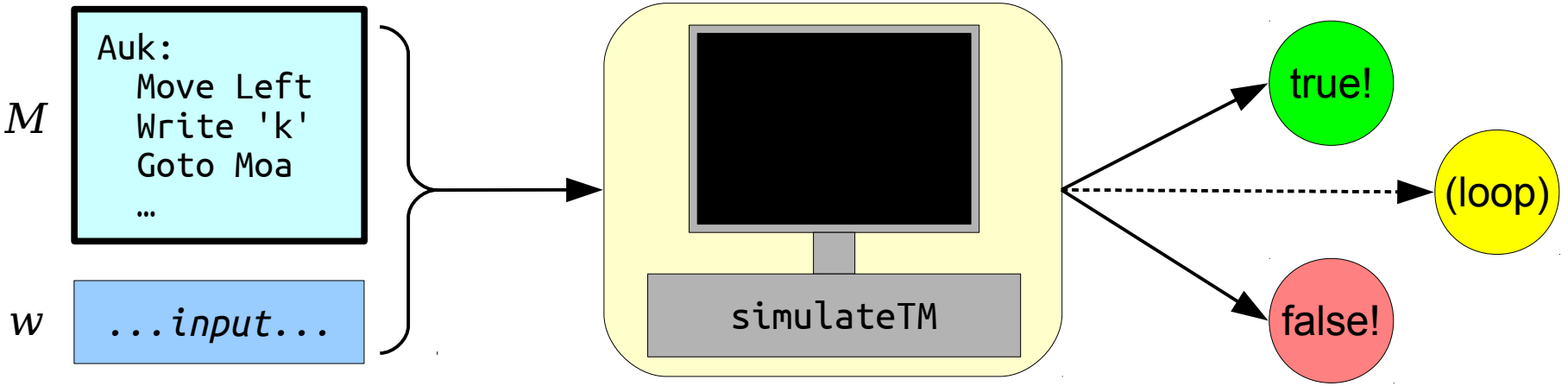
A TM Simulator

- 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.



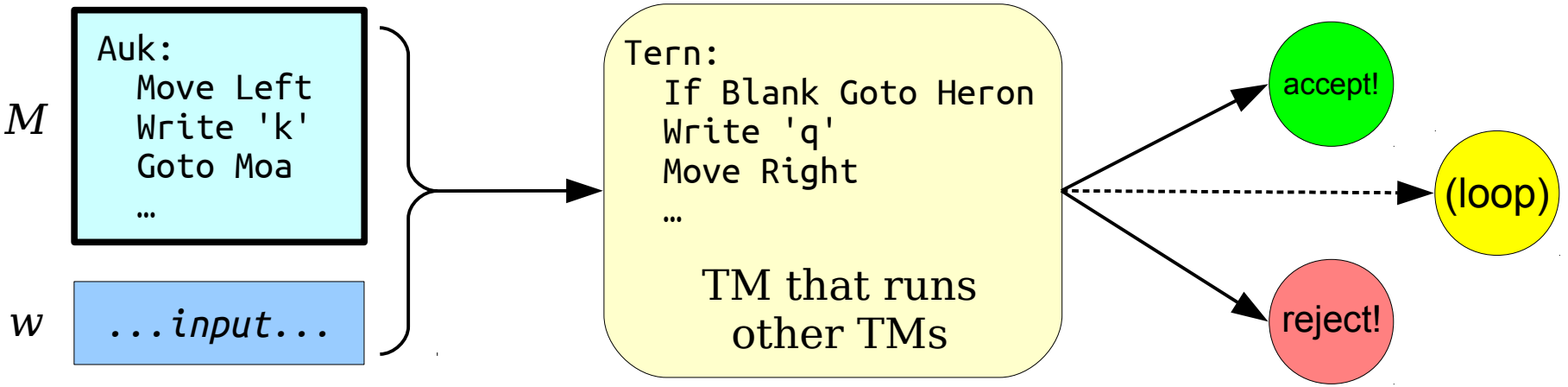
A TM Simulator

- 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?



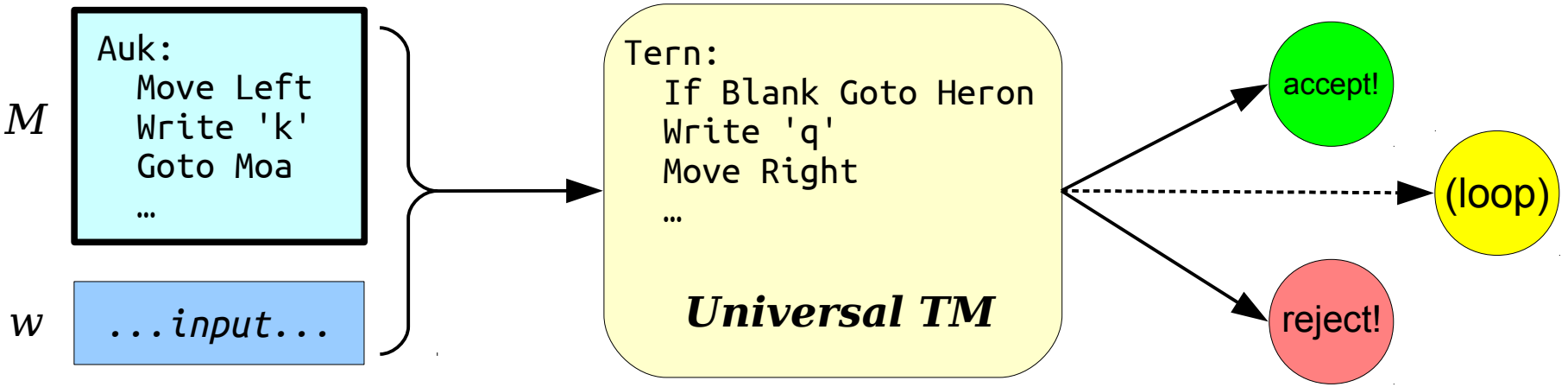
A TM Simulator

- 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?



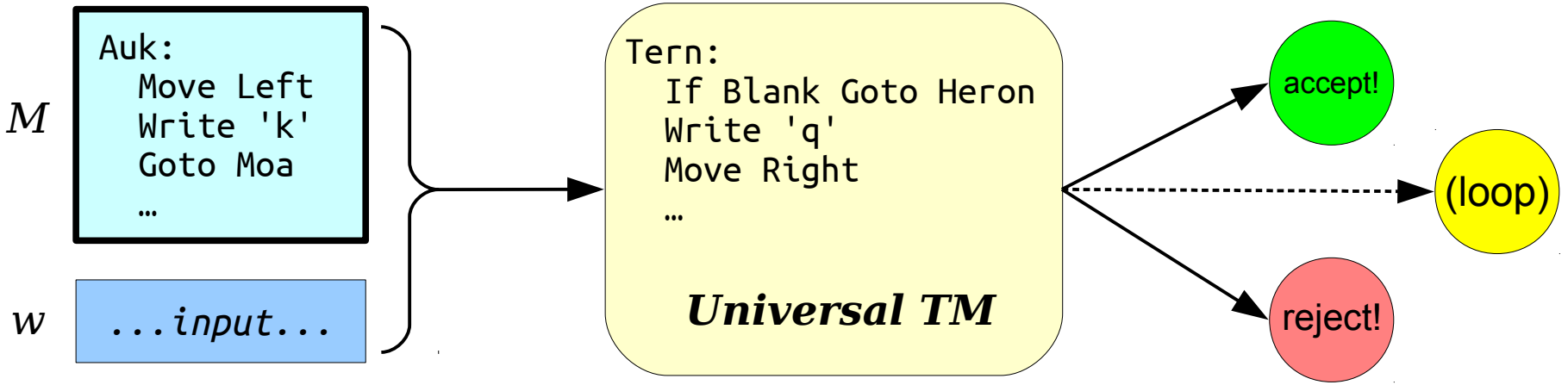
The Universal Turing Machine

- ***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 }

Let M be a TM where $\mathcal{L}(M) = \{ \mathbf{a}^n\mathbf{b}^n \mid n \in \mathbb{N} \}$.
Fill in the following blanks:

_____ accepts **aabb**
_____ accepts $\langle M,$ **aabb**$\rangle$
_____ $\in A_{TM}$

Respond at pollev.com/cs103

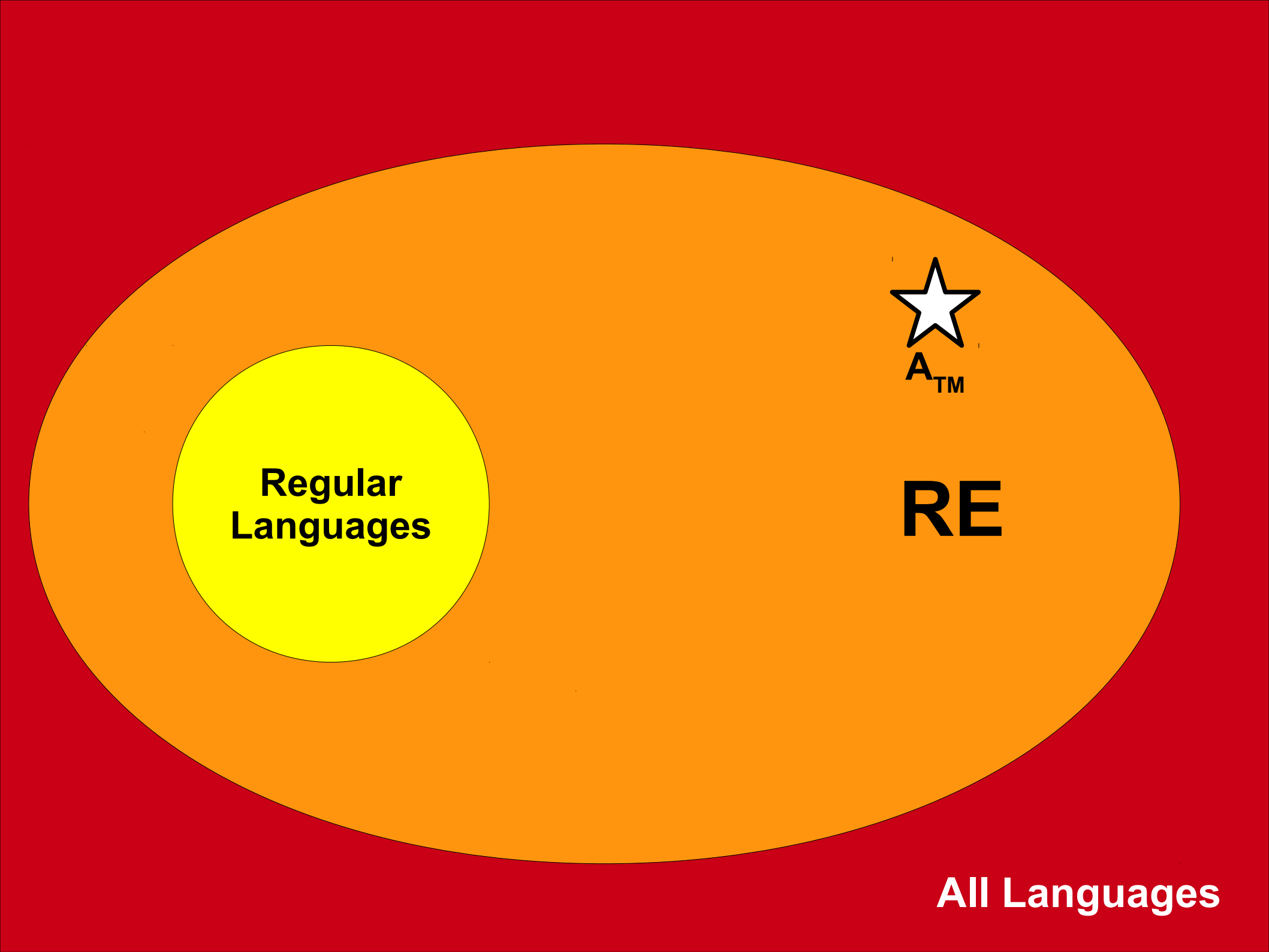
The Language A_{TM}

$A_{TM} = \{ \langle M, w \rangle \mid M$ 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 $\langle M, w \rangle$.
  - $\langle M, w \rangle \in A_{TM}$.

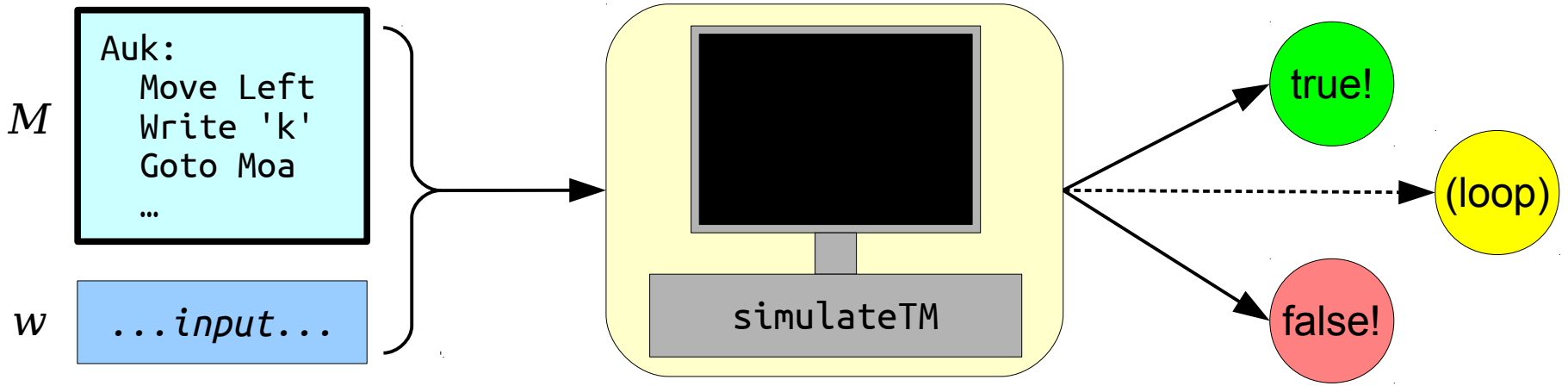

Regular
Languages
A_{TM}
RE
All Languages

Uh… so what?

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

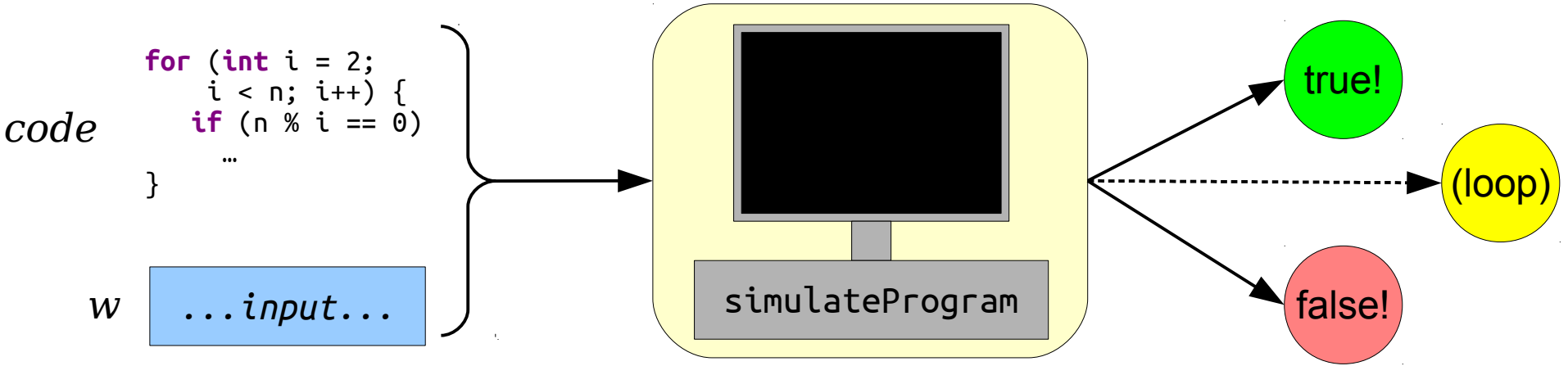
Why Does This Matter?

- 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?



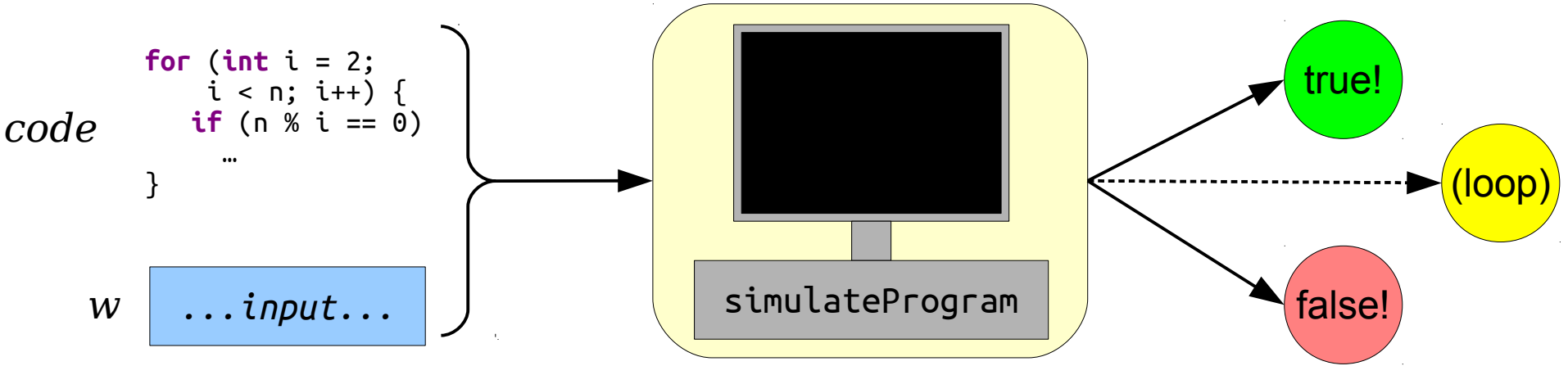
Why Does This Matter?

- 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 programme the 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 programme our digital computer to imitate it and it will also be a brain.”

Time-Out for Announcements!

Back to CS103!

Things are about to get weird...

Part One: Self-Defeating Objects

A ***self-defeating object*** is an object whose essential properties ensure it doesn’t exist.

Question: Why is there no largest integer?

Answer: Because if n is the largest integer, what happens when we look at $n+1$?

Self-Defeating Objects

Theorem: There is no largest integer.

Proof sketch: Suppose for the sake of contradiction that there is a largest integer. Call that integer n.

Consider the integer $n+1$.

Notice that $n < n+1$.

But then n isn't the largest integer.

Contradiction! ■-*ish*

Self-Defeating Objects

Theorem: There is no largest integer.

Proof sketch: Suppose for the sake of contradiction that there is a largest integer. Call that integer n.

Consider the integer $n+1$.

Notice that $n < n+1$.

But then n isn't the larg

Contradiction! ■-*ish*

We're using n to construct something that undermines n, hence the term "self-defeating."

An Important Detail

Careful – we're assuming what we're trying to prove!

Claim: There is a largest integer.

Proof: Assume x is the largest integer.

Notice that $x > x - 1$.

So there's no contradiction. ■-*ish*

How do we know there's no contradiction? We just checked one case.

Self-Defeating Objects

- If you can show

$$x \text{ exists} \rightarrow \bot$$

  then you know that *x* doesn't exist. (This is a proof by contradiction.)

- If you can show

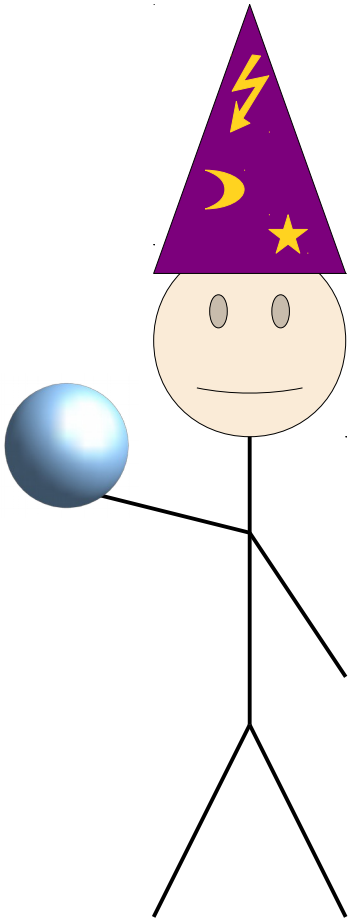
$$x \text{ exists} \rightarrow \top$$

  you cannot conclude that *x* exists. (This is not a valid proof technique.)

Part Two: The Fortune Teller

The Fortune Teller

- A fortune teller appears who claims they can see into anyone’s future.
- For a nominal fee, the fortune teller will tell you anything you want to know about the future.



The Fortune Teller

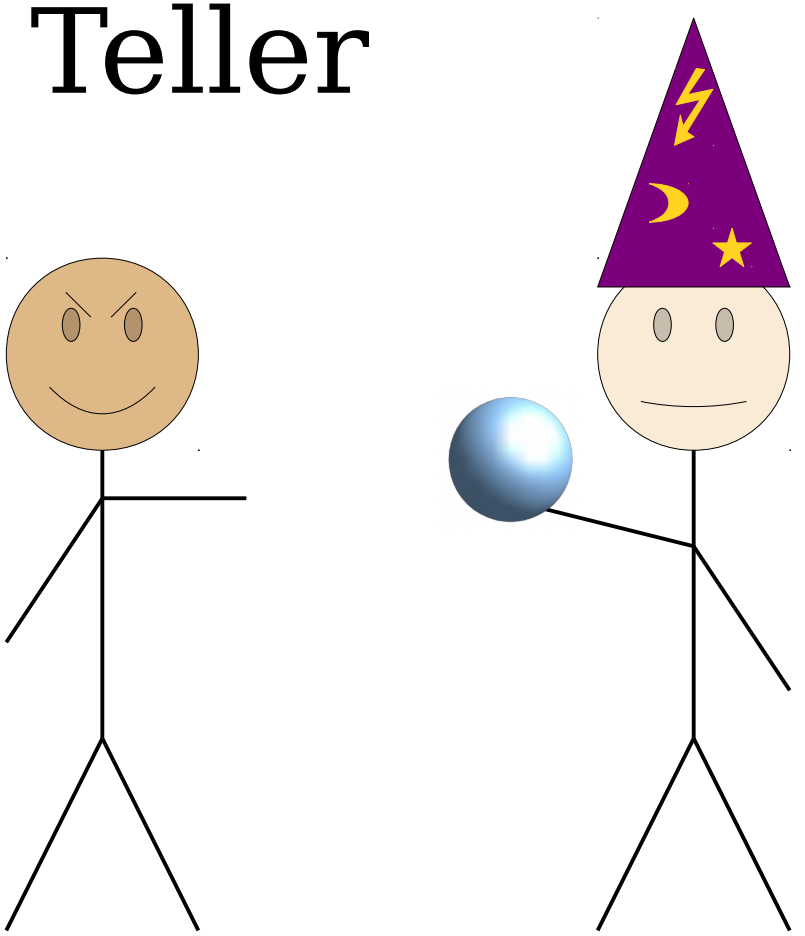
- One day, a trickster arrives. The trickster thinks the fortune teller is lying and can't really see the future.
- The trickster says the following:

"I have a yes/no question about the future. But before I ask my question, let's talk payment.

If you answer yes, then I'll pay you $137.

If you answer no, then I'll pay you $42.

- The fortune teller thinks for a moment, then agrees.



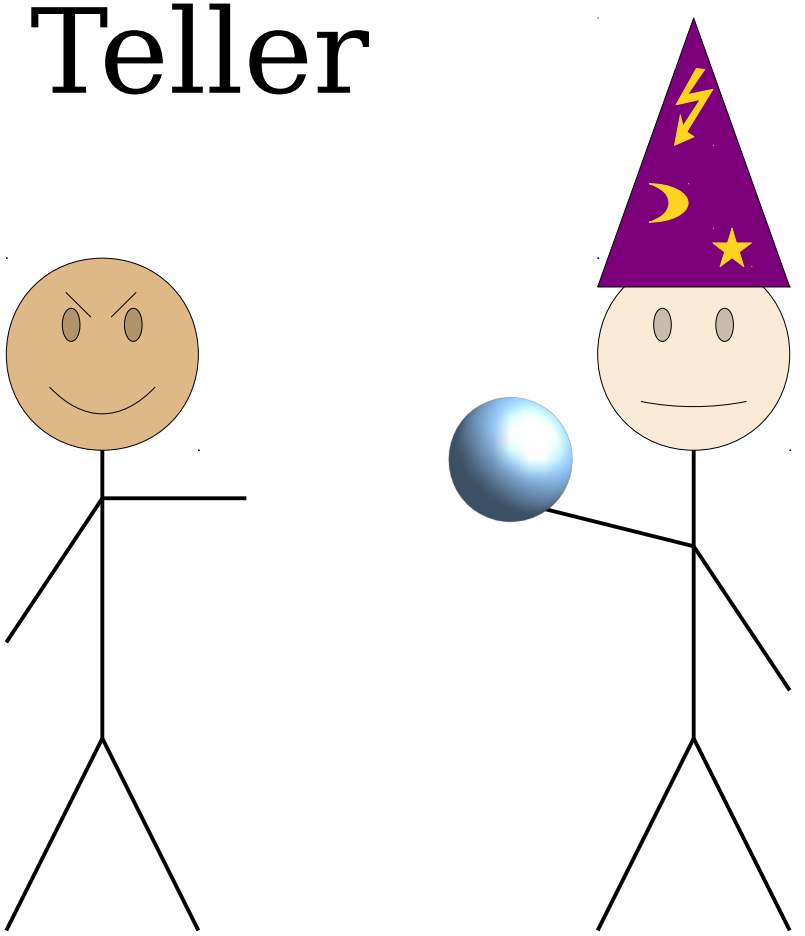
Trickster pays $137 if the fortune teller answers "yes."

Trickster pays $42 if the fortune teller answers "no."

The Fortune Teller

- The trickster then asks this question:

"Am I going to pay you $42?"



The fortune teller is trapped! Why? Discuss with your neighbors.

Respond at pollev.com/cs103

Trickster pays $137 if the fortune teller answers "yes."

Trickster pays $42 if the fortune teller answers "no."

The Fortune Teller

- The payment scheme the fortune teller agreed to means

  Fortune Teller Says Yes ↔ ***Trickster Pays $137***.

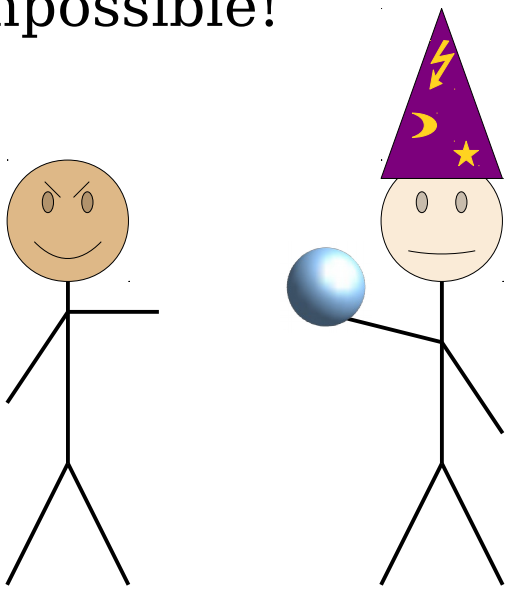
- The trickster's question to the fortune teller means

  Fortune Teller Says Yes ↔ ***Trickster Pays $42***.

- Putting this together, we get

  Trickster Pays $42 ↔ ***Trickster Pays $137***.

- This is impossible!

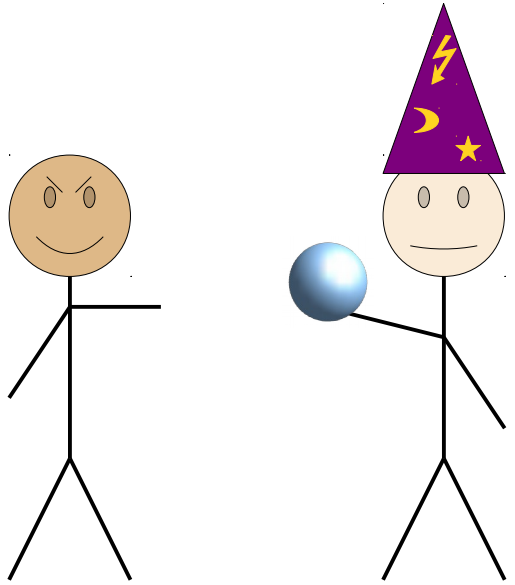


Trickster pays $137 if the fortune teller answers "yes."

Trickster pays $42 if the fortune teller answers "no."

The Fortune Teller

- The fortune teller is a self-defeating object.
- The trickster's strategy is to couple the fortune teller's behavior to what the future holds.
  - The trickster's behavior is chosen in advance to make the fortune teller's answer wrong.
- Therefore, the fortune teller can't answer all questions about all people in the future.



Trickster pays $137 if the fortune teller answers "yes."

Trickster pays $42 if the fortune teller answers "no."

Part Three: Why Do Programs Loop?

Thoughts on Loops

- In practice, the programs we write sometimes go into infinite loops.
- In Theoryland, Turing machines are allowed to loop. This happens if they don’t accept and don’t reject.
- ***Question:*** Why are infinite loops possible?
- Or rather: are infinite loops an inherent part of computation, or are they some weird sort of “accident” in how we program computers?

Thoughts on Loops

- ***Theorem:*** The language A_{TM} is recognizable, but undecidable.
  - There's a *recognizer* for A_{TM} (specifically, the universal Turing machine U_{TM}).
  - It is impossible to build a *decider* for this language.
- Stated differently, there's a program we can write (a universal TM) that *has* to loop infinitely on some inputs.
- ***Goal:*** Prove this theorem, and explore its theoretical and philosophical implications.

A_{TM} Revisited

- As a refresher, the language A_{TM} is

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

- The universal TM U_{TM} has the following behavior when given as input a TM M and a string w:
  - 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} is a recognizer for A_{TM}, but because of that last case it's not a decider for A_{TM}.

A_{TM} Revisited

- As a refresher, the language A_{TM} is

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

- Given a TM M and a string w, a decider D for A_{TM} would need to have this behavior:
  - If M accepts w, then D **?** $\langle M, w \rangle$.
  - If M rejects w, then D **?** $\langle M, w \rangle$.
  - If M loops on w, then D **?** $\langle M, w \rangle$.
- This is basically the same set of requirements as U_{TM}, except for what happens if M loops on w.
- Our goal is to prove that there is no way to build a program that meets these requirements.

Why is A_{TM} Hard?

- ***Intuition:*** A decider for A_{TM} would be able to...
  - ... determine whether the hailstone sequence terminates for any input. (Write a recognizer that runs the hailstone sequence, then feed it into the decider for A_{TM}.)
  - ... solve other open math problems.
  - ... and much, much more.
- In other words, this seemingly simple problem of "is this program going to terminate?" accidentally scoops up a bunch of other seemingly harder problems.

Part Four: 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;
}
```

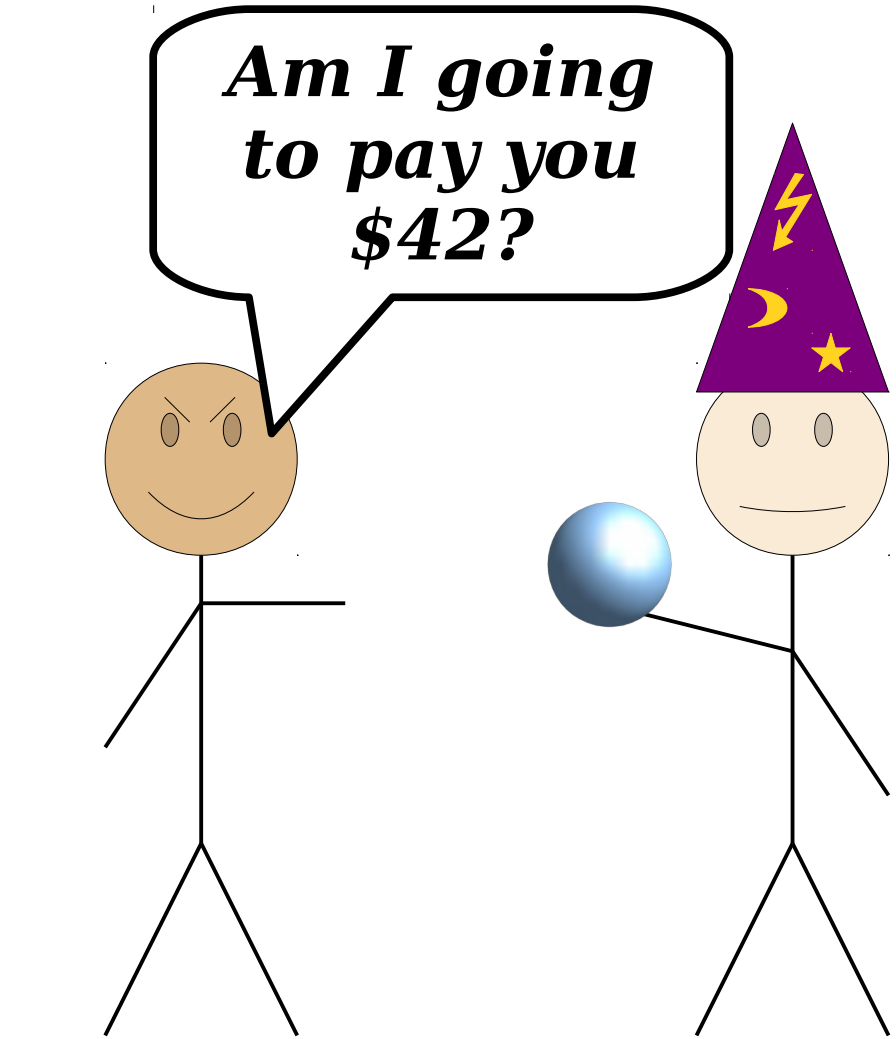
Part Five: Putting It All Together

To Recap

- We're assuming that, somehow, someone wrote a function

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

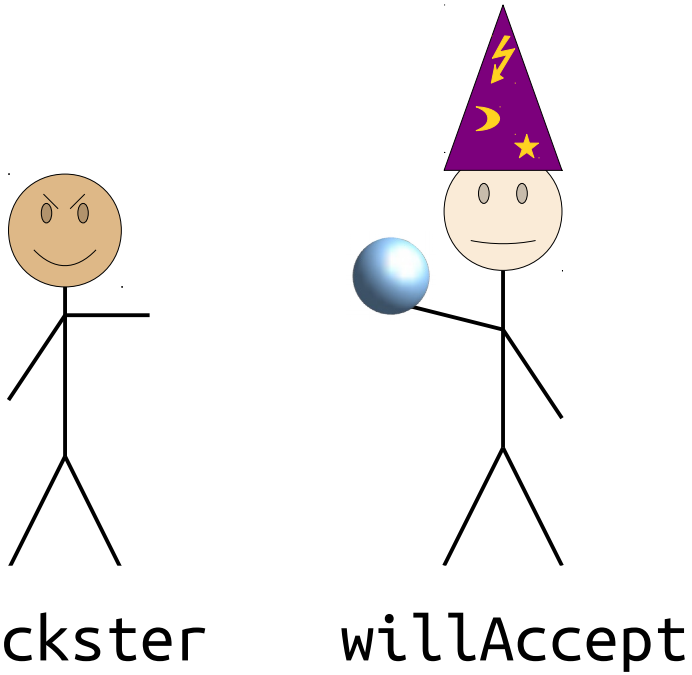
  that takes the code of a function and an input to that function, then
  - returns true if `function(input)` returns true, and
  - returns false if `function(input)` doesn't return true.
- ***Goal:*** Show that this decider is "self-defeating;" its power is so great that it undermines itself.
- ***Idea:*** Convert the fortune teller story into a program.



Trickster pays $137 if the fortune teller answers “yes.”

Trickster pays $42 if the fortune teller answers “no.”

```
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 willAccept

If `willAccept` says `trickster` will return true, then `trickster` returns false.

If `willAccept` says `trickster` will not return true, then `trickster` returns true.

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

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

A self-defeating object.

Using that object against itself.

```
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);
}
```

"The largest integer *n*."

"The integer *n* + 1."

Theorem: There is no largest integer.

Proof sketch: Suppose for the sake of contradiction that there is a largest integer. Call that integer n.

Consider the integer $n+1$.

Notice that $n < n+1$.

But then n isn't the largest integer.

Contradiction! ■-*ish*

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

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

Proof:

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

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$.

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}.

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.

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);
}
```

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);
}
```

Choose a string w.

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);
}
```

Choose a string w. We consider two cases:

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);
}
```

Choose a string w. We consider two cases:

Case 1: `willAccept(me, input)` returns true.

Case 2: `willAccept(me, 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);
}
```

Choose a string w. We consider two cases:

Case 1: `willAccept(me, input)` returns true. Since `willAccept` decides A_{TM}, this means `trickster(w)` returns true.

Case 2: `willAccept(me, 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);
}
```

Choose a string w. We consider two cases:

Case 1: `willAccept(me, input)` returns true. Since `willAccept` decides A_{TM}, this means `trickster(w)` returns true. However, given how `trickster` is written, in this case `trickster(w)` returns false.

Case 2: `willAccept(me, 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);
}
```

Choose a string w. We consider two cases:

Case 1: `willAccept(me, input)` returns true. Since `willAccept` decides A_{TM}, this means `trickster(w)` returns true. However, given how `trickster` is written, in this case `trickster(w)` returns false.

Case 2: `willAccept(me, input)` returns false. Since `willAccept` decides A_{TM}, this means `trickster(w)` doesn't return true.

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);
}
```

Choose a string w. We consider two cases:

Case 1: `willAccept(me, input)` returns true. Since `willAccept` decides A_{TM}, this means `trickster(w)` returns true. However, given how `trickster` is written, in this case `trickster(w)` returns false.

Case 2: `willAccept(me, input)` returns false. Since `willAccept` decides A_{TM}, this means `trickster(w)` doesn't return true. However, given how `trickster` is written, in this case `trickster(w)` returns true.

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);
}
```

Choose a string w. We consider two cases:

Case 1: `willAccept(me, input)` returns true. Since `willAccept` decides A_{TM}, this means `trickster(w)` returns true. However, given how `trickster` is written, in this case `trickster(w)` returns false.

Case 2: `willAccept(me, input)` returns false. Since `willAccept` decides A_{TM}, this means `trickster(w)` doesn't return true. However, given how `trickster` is written, in this case `trickster(w)` returns true.

In both cases we reach a contradiction, so our assumption must have been wrong.

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);
}
```

Choose a string w. We consider two cases:

Case 1: `willAccept(me, input)` returns true. Since `willAccept` decides A_{TM}, this means `trickster(w)` returns true. However, given how `trickster` is written, in this case `trickster(w)` returns false.

Case 2: `willAccept(me, input)` returns false. Since `willAccept` decides A_{TM}, this means `trickster(w)` doesn't return true. However, given how `trickster` is written, in this case `trickster(w)` returns true.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \notin \mathbf{R}$.

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`:

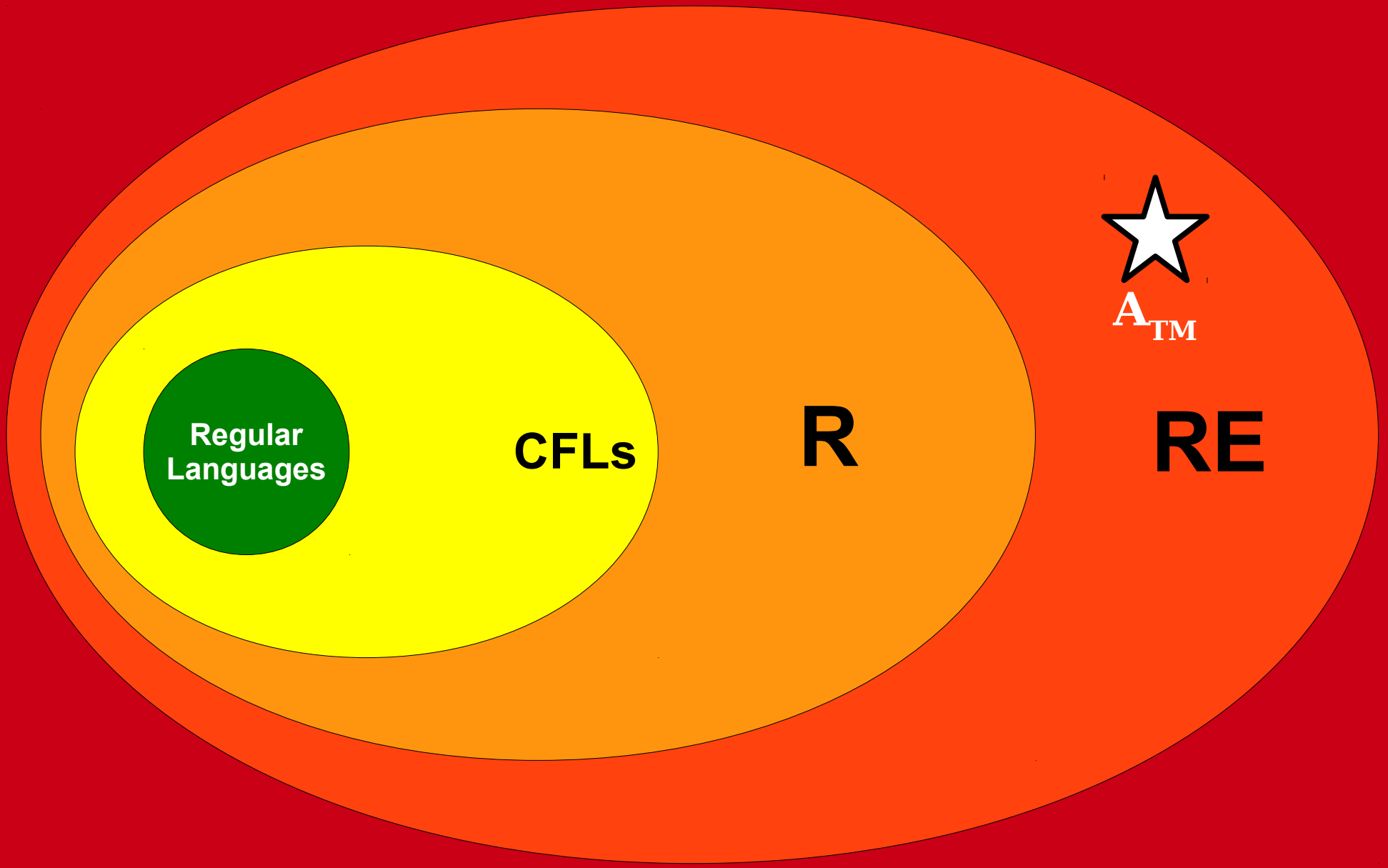
```
bool trickster(string input) {
    string me = /* source code of trickster */;
    return !willAccept(me, input);
}
```

Choose a string w. We consider two cases:

Case 1: `willAccept(me, input)` returns true. Since `willAccept` decides A_{TM}, this means `trickster(w)` returns true. However, given how `trickster` is written, in this case `trickster(w)` returns false.

Case 2: `willAccept(me, input)` returns false. Since `willAccept` decides A_{TM}, this means `trickster(w)` doesn't return true. However, given how `trickster` is written, in this case `trickster(w)` returns true.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \notin \mathbf{R}$. ■


Regular Languages
CFLs
R
RE
A_{TM}
All Languages

What Does This Mean?

- In one fell swoop, we've proven that
  - A_{TM} is ***undecidable***; there is no general algorithm that can determine whether a TM will accept a string.
  - $\mathbf{R} \neq \mathbf{RE}$, because $A_{TM} \notin \mathbf{R}$ but $A_{TM} \in \mathbf{RE}$.
- What do these three statements really mean? As in, why should you care?

$A_{TM} \notin \mathbf{R}$

- What exactly does it mean for A_{TM} to be undecidable?

  Intuition: The only general way to find out what a program will do is to run it.

- As you'll see, this means that it's provably impossible for computers to be able to answer most questions about what a program will do.

$A_{TM} \notin \mathbf{R}$

- At a more fundamental level, the existence of undecidable problems tells us the following:

  There is a difference between what is true and what we can discover is true.

- Given a TM *M* and a string *w*, one of these two statements is true:

  M* accepts *w ***M* does not accept *w***

  But since A_{TM} is undecidable, there is no algorithm that can always determine which of these statements is true!

R ≠ RE

- Because **R ≠ RE**, there is a difference between decidability and recognizability:

In some sense, it is fundamentally harder to solve a problem than it is to check an answer.

- There are problems where, when the answer is “yes,” you can confirm it (run a recognizer), but where if you don’t have the answer, you can’t come up with it in a mechanical way (build a decider).

Next Time

- ***Why All This Matters***
  - Important, practical, undecidable problems.
- ***Intuiting RE***
  - What exactly is the class **RE** all about?
- ***Verifiers***
  - A totally different perspective on problem solving.
- ***Beyond RE***
  - Finding an impossible problem using very familiar techniques.