Turing Machines

Part Two

Outline for Today

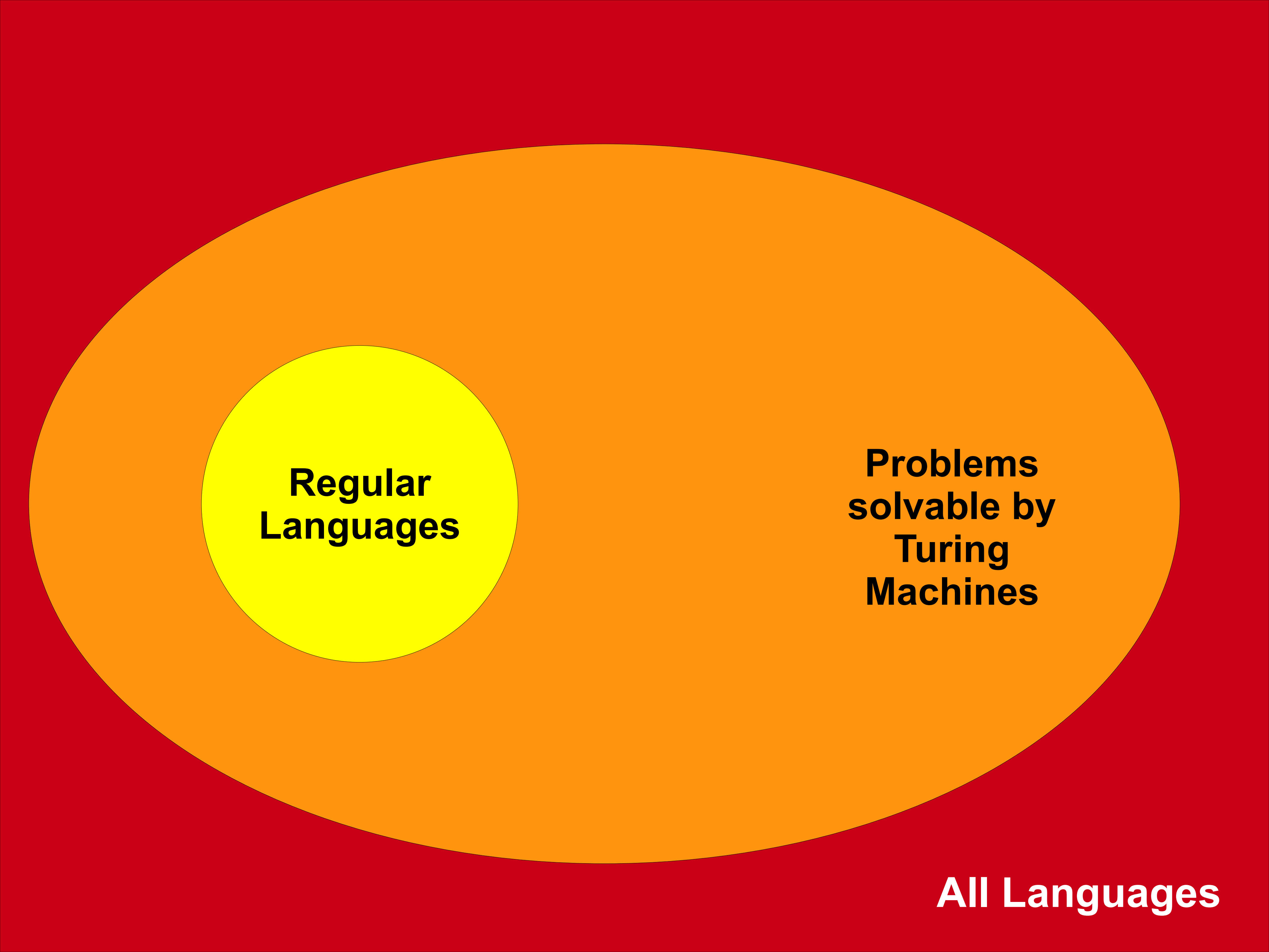
- ***The Church-Turing Thesis***
  - How powerful are Turing machines?
- ***Decidability and Recognizability***
  - Two notions of “solving a problem.”
- ***Universal Machines***
  - A single computer that can compute anything computable anywhere.
- ***Self-Referential Software***
  - Programs that compute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

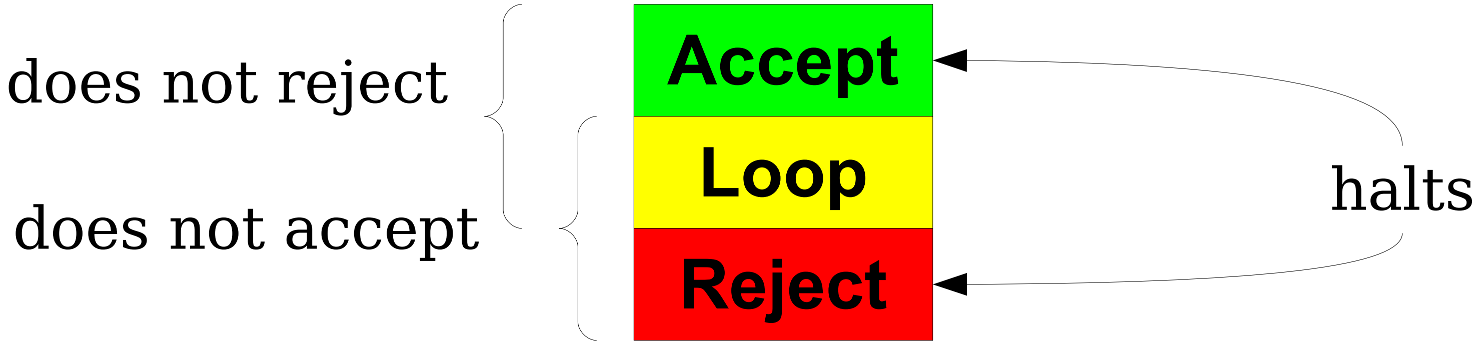
- Ryan Williams


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

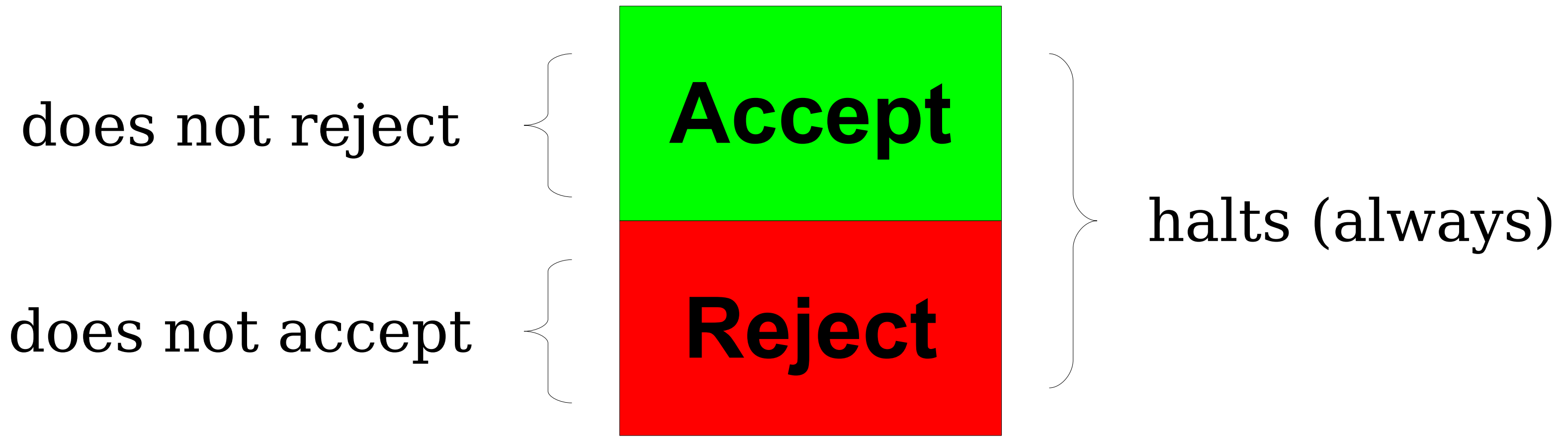
- The class **RE** consists of all recognizable languages.
- Formally speaking:

  RE = { L | L 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:

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

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

- 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.

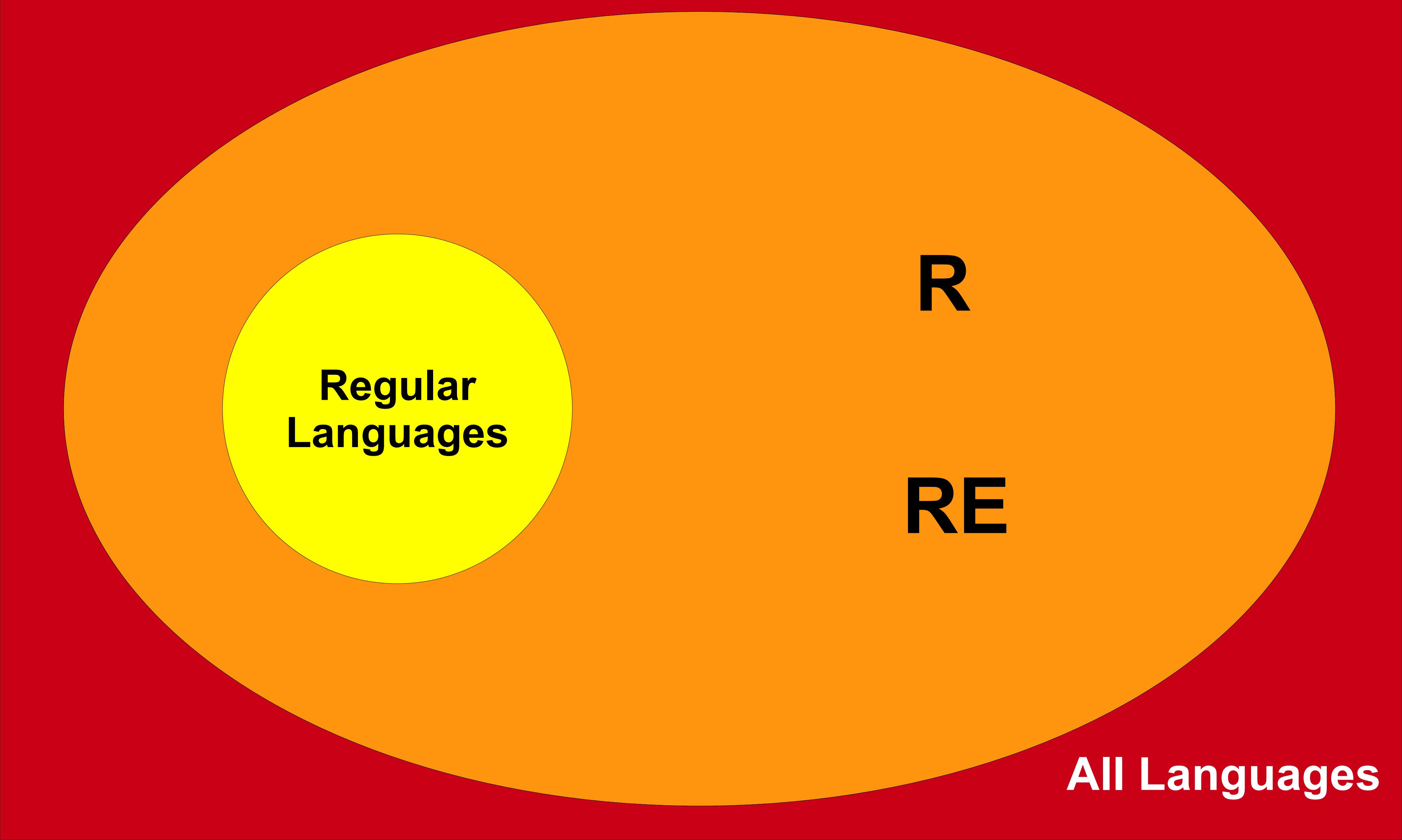
R and **RE** Languages

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

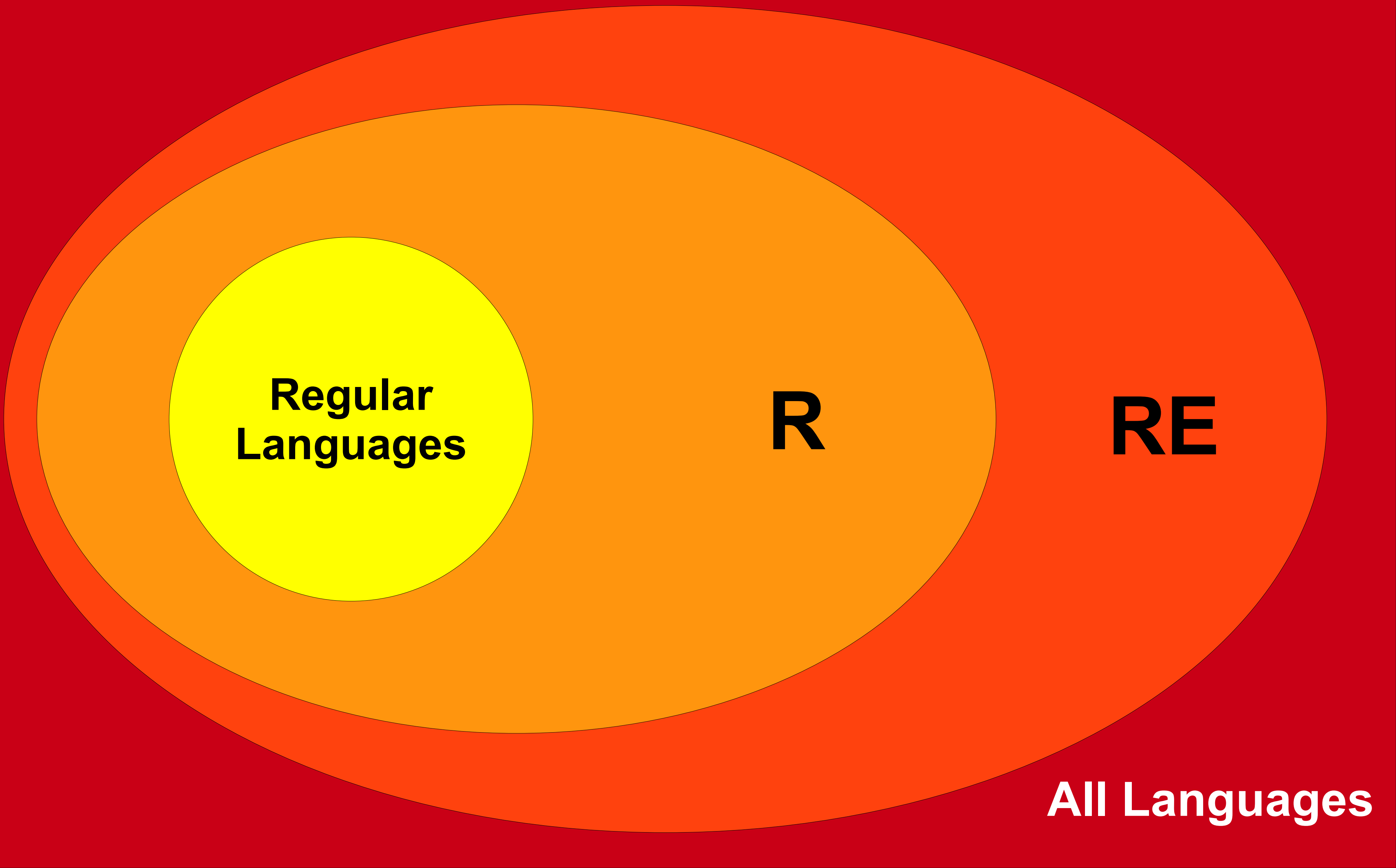
$$\mathbf{R} \stackrel{?}{=} \mathbf{RE}$$

- 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

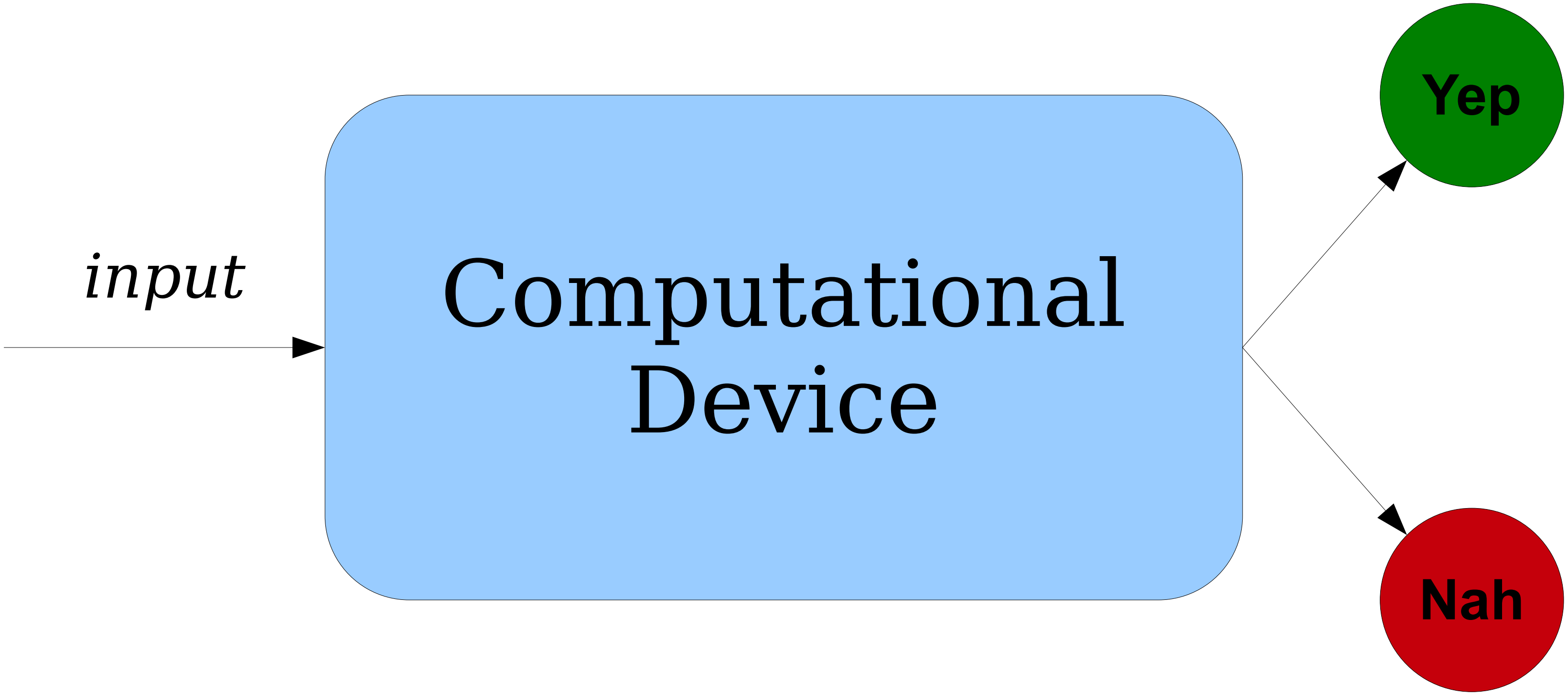
- A ***decision problem*** is a type of problem where the goal is to provide a yes or no answer.
- Example: Bin Packing

  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?

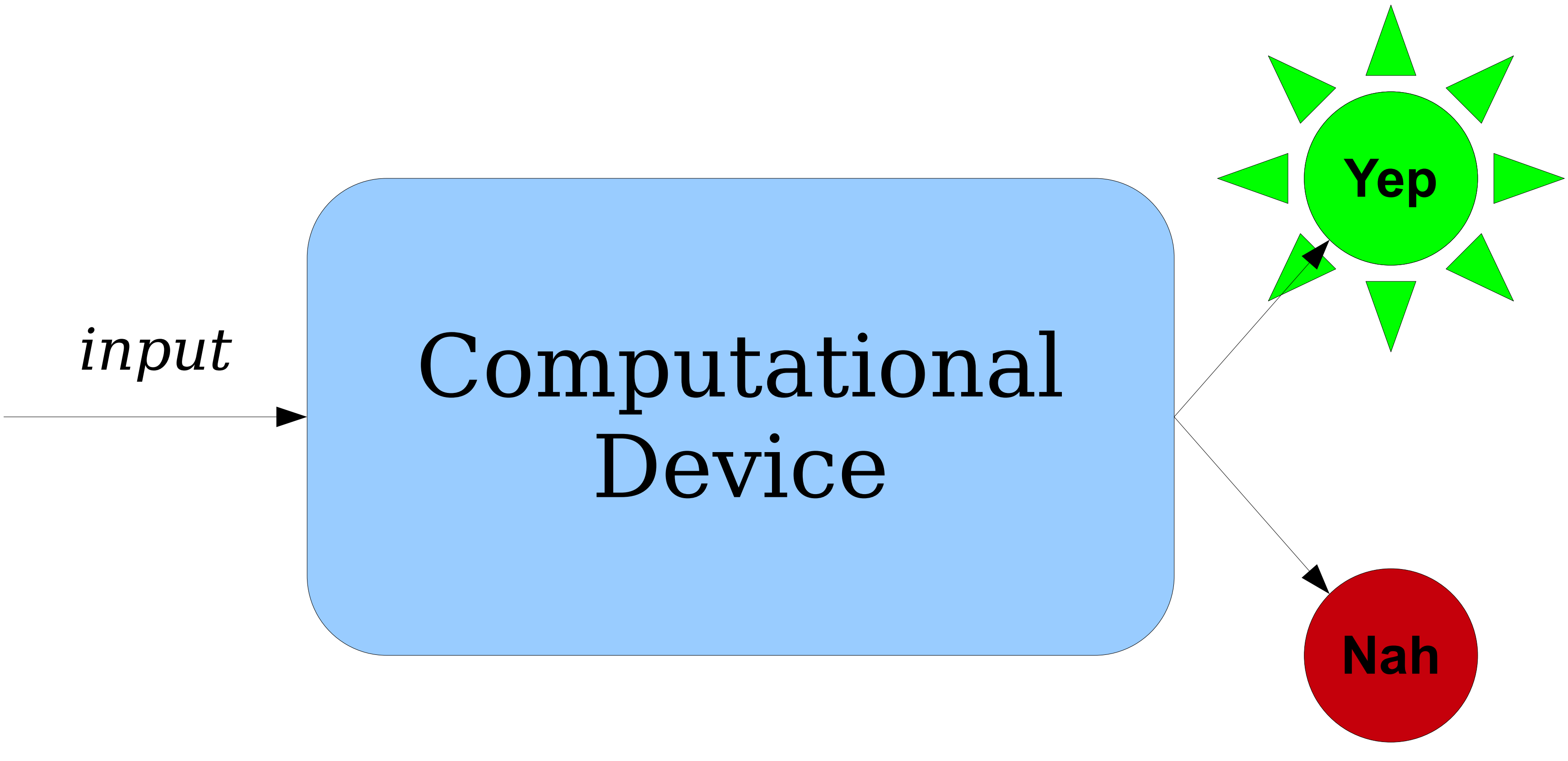
- Example: Dominating Set Problem

  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?

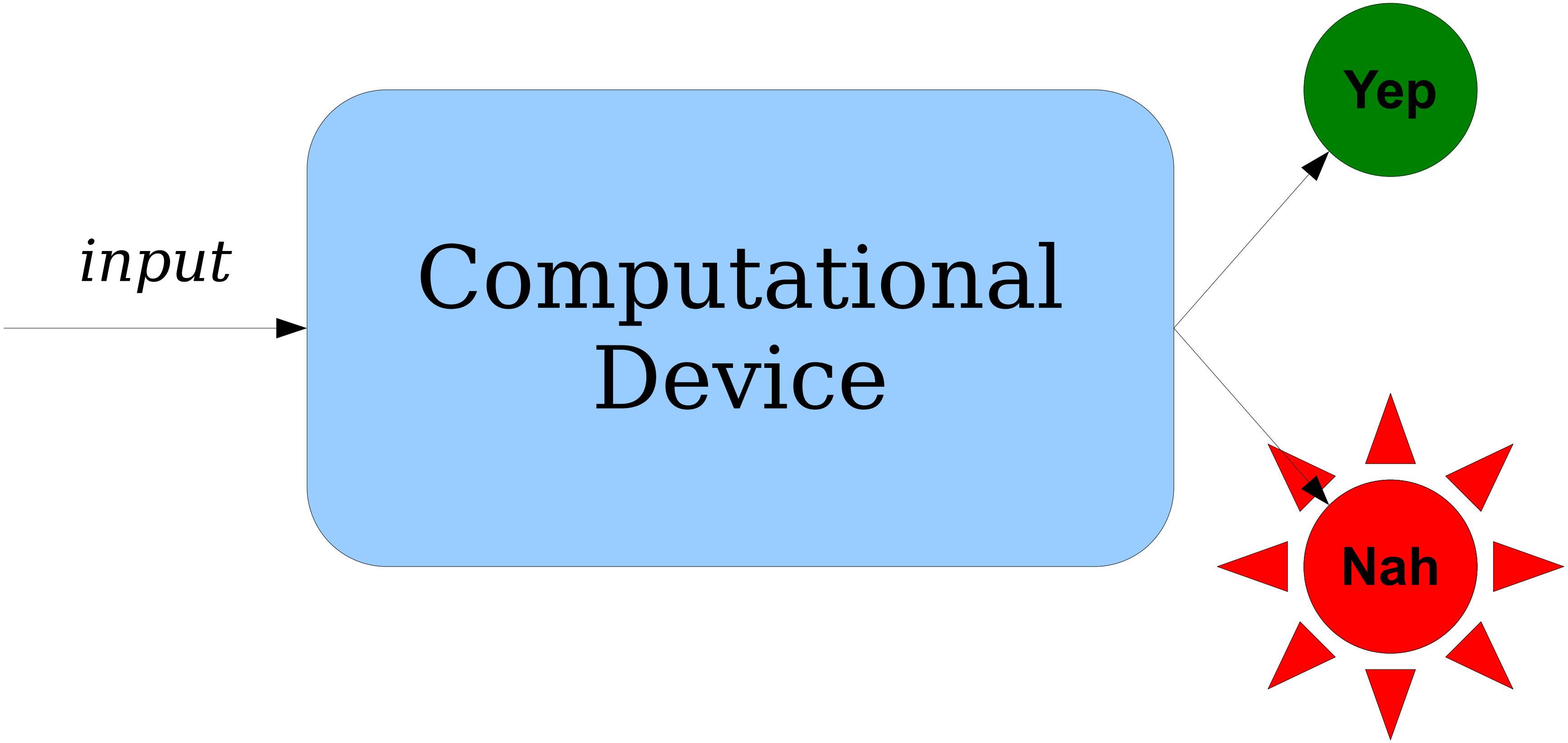
A Model for Solving Problems



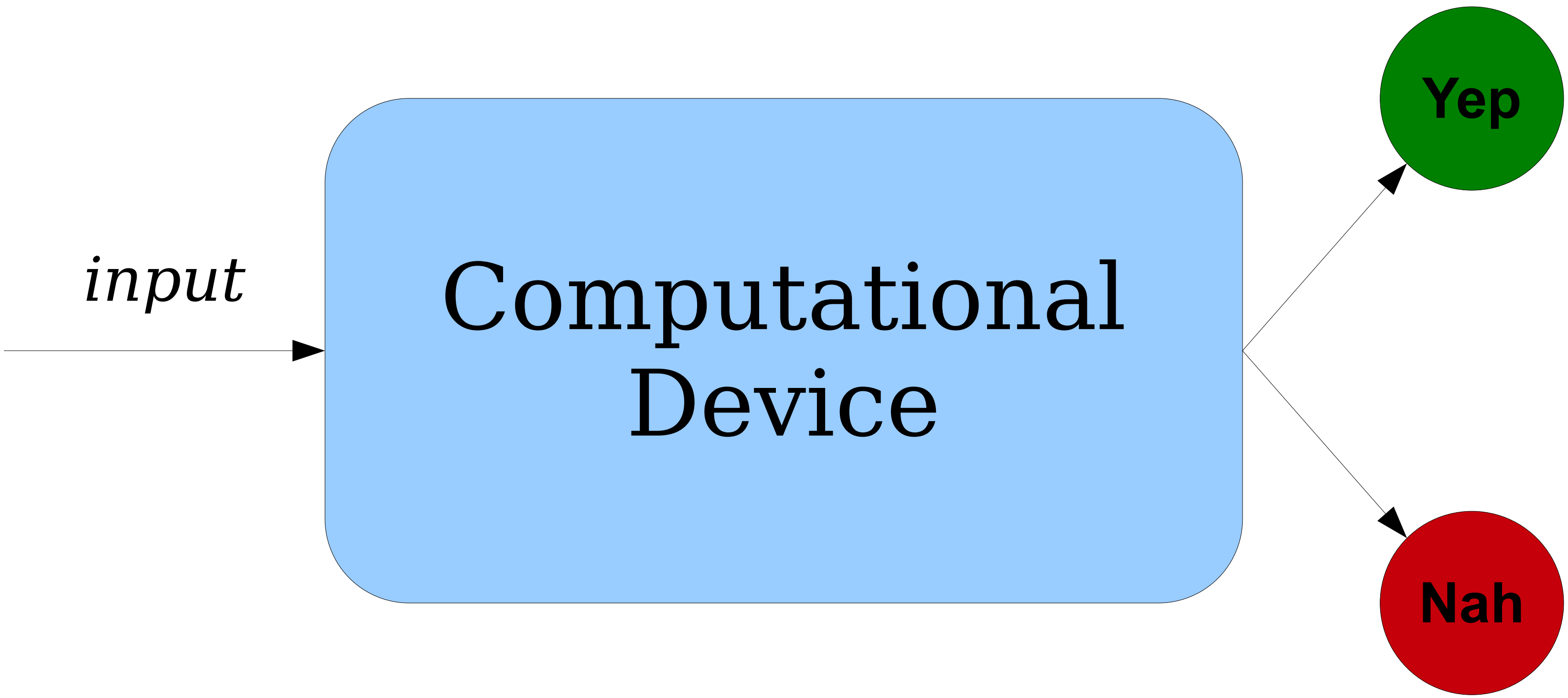
A Model for Solving Problems



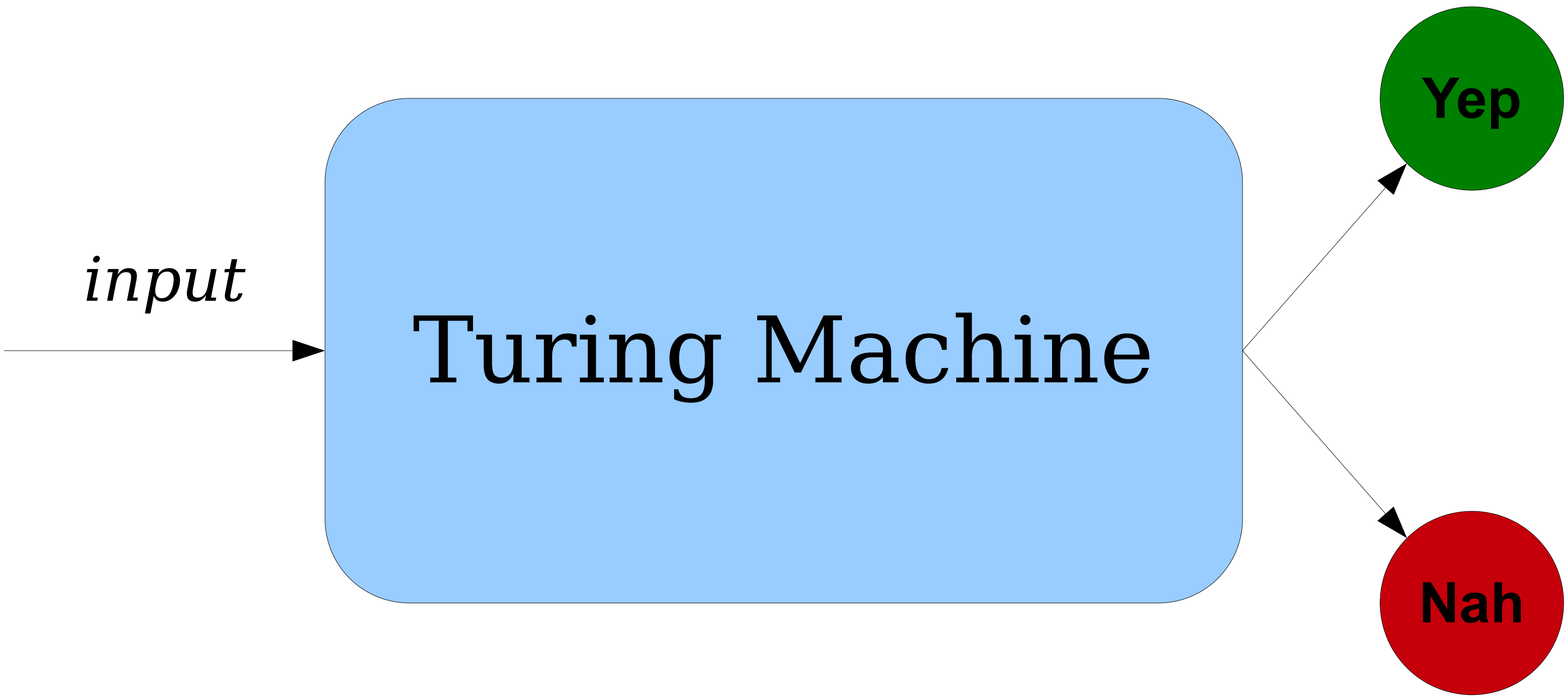
A Model for Solving Problems



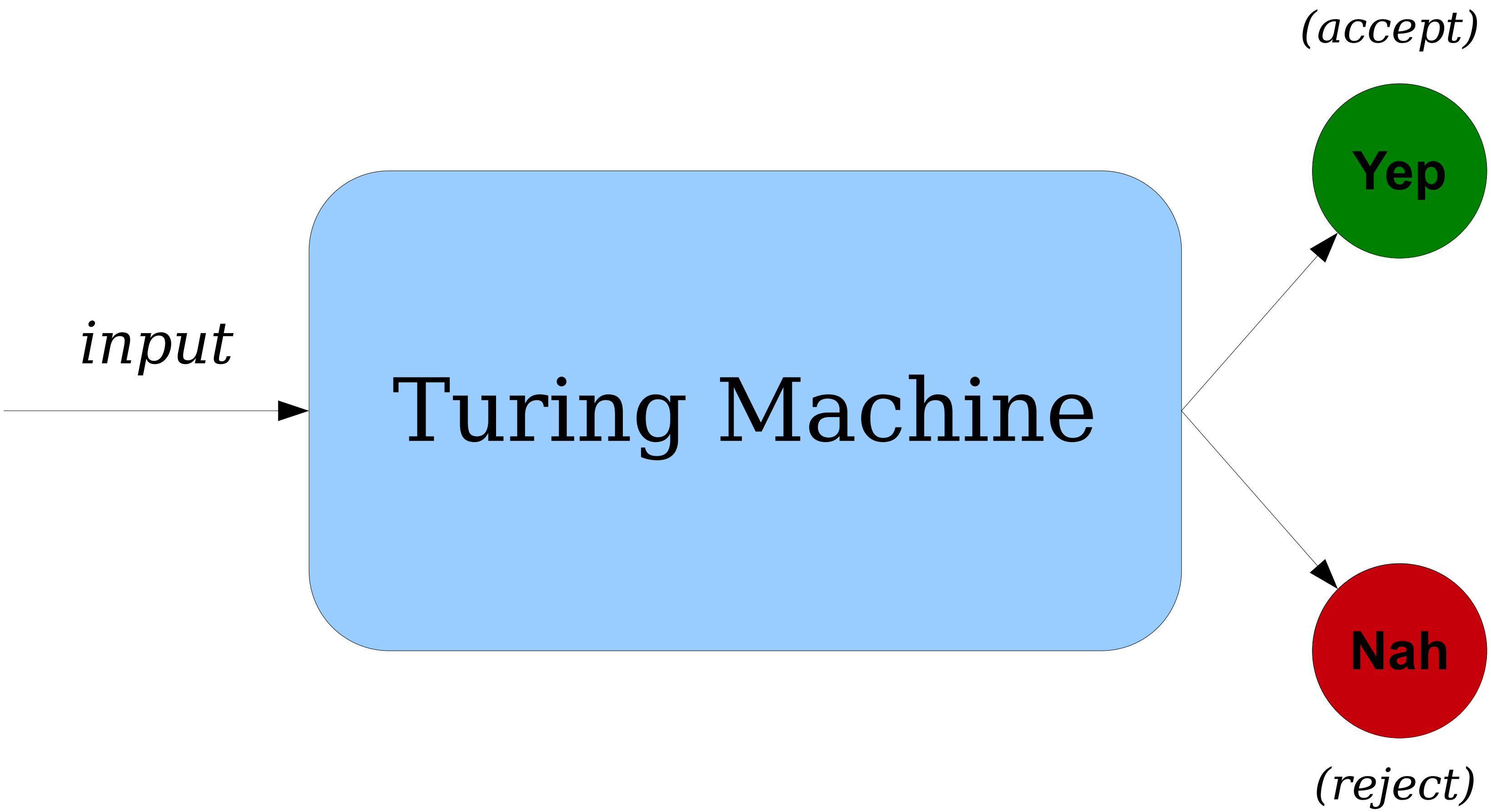
A Model for Solving Problems



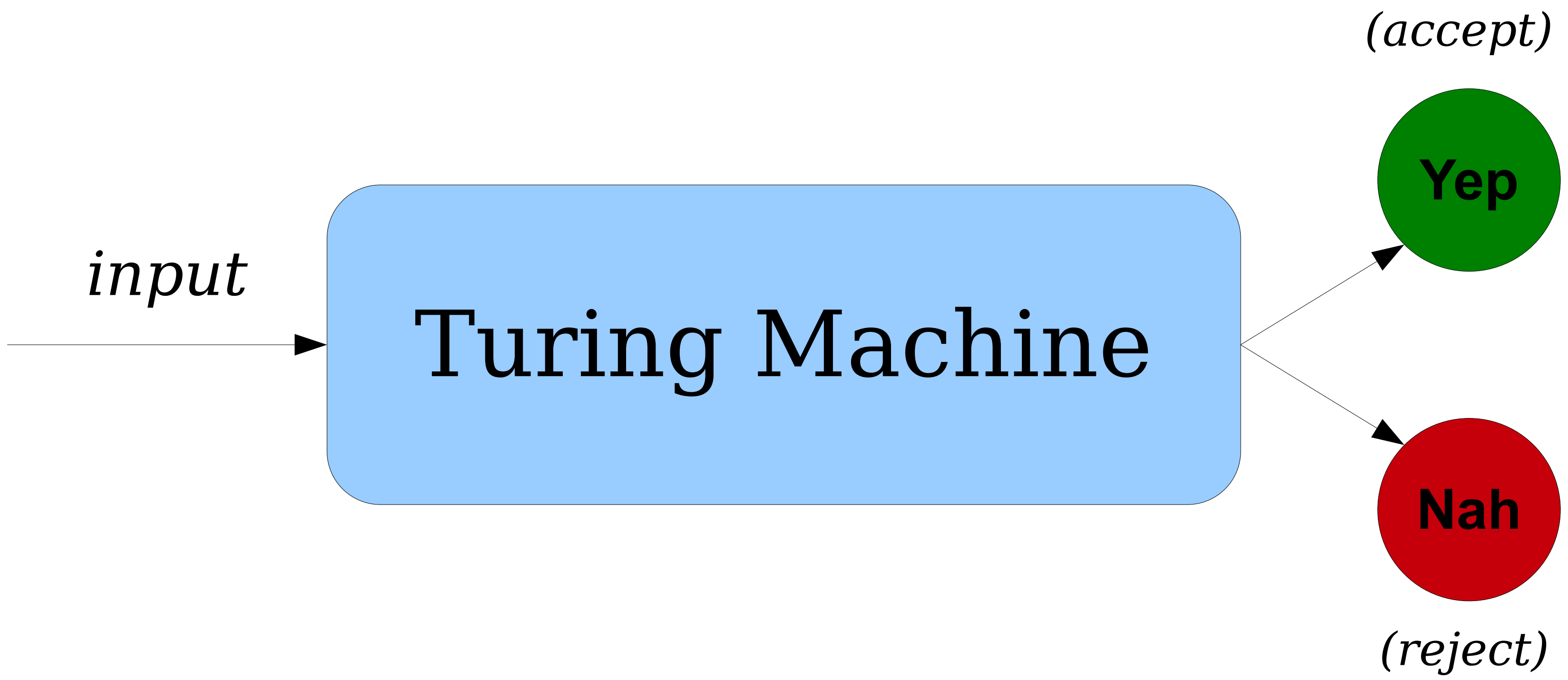
A Model for Solving Problems



A Model for Solving Problems

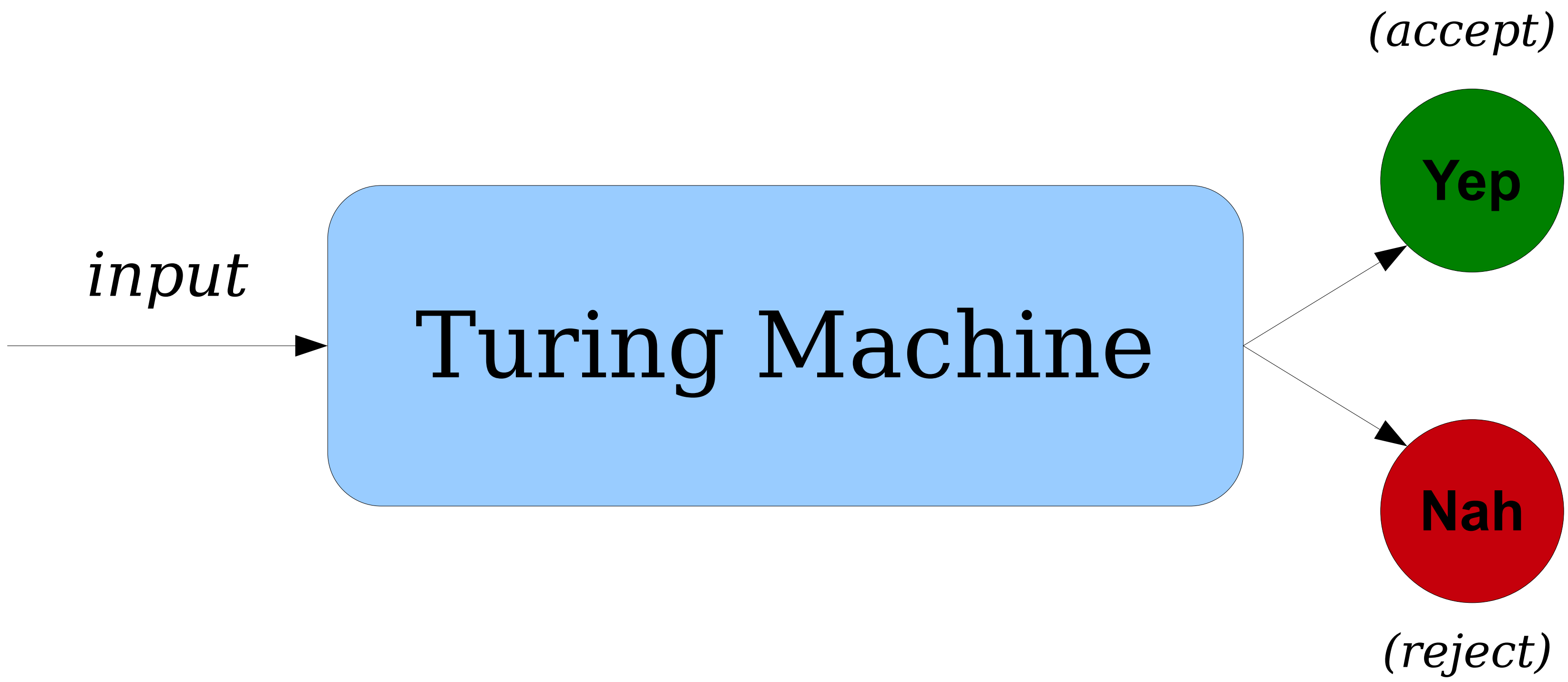


A Model for Solving Problems



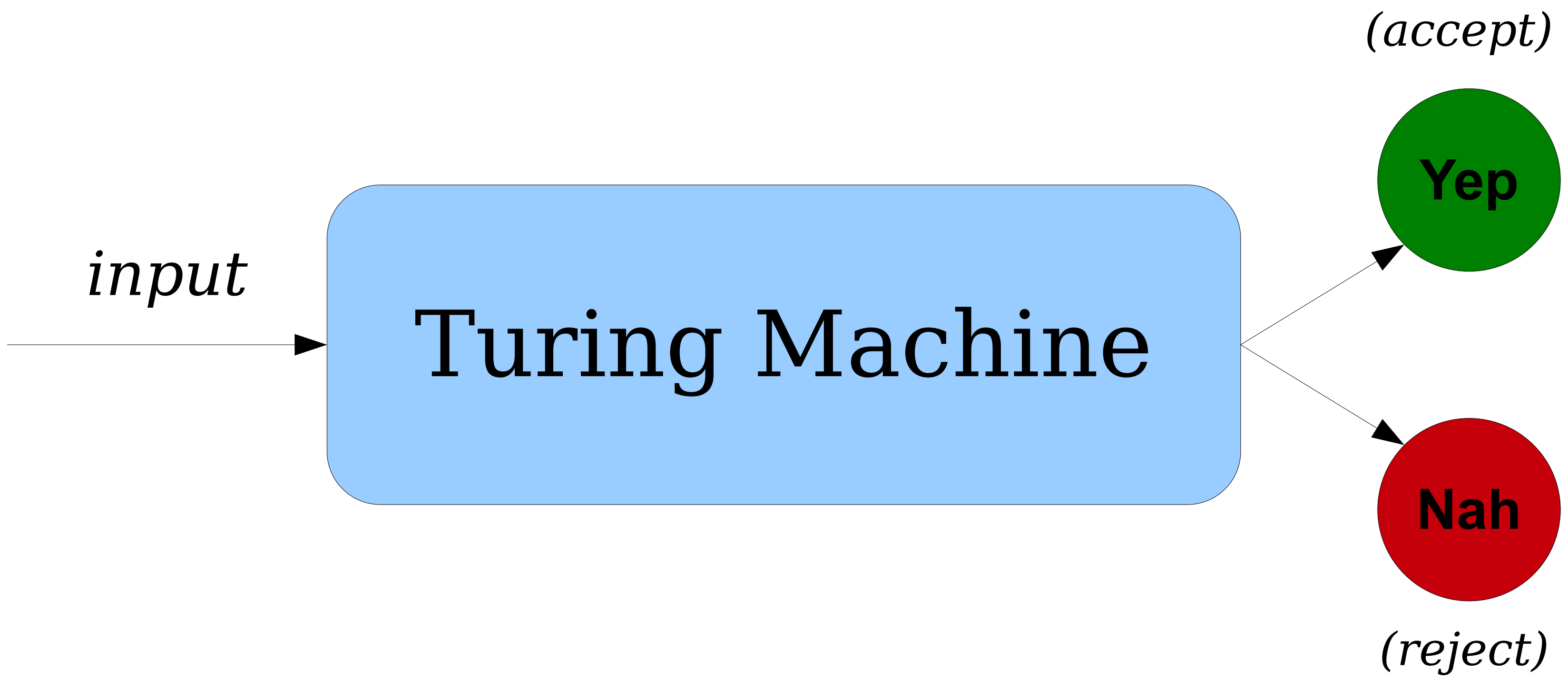

```
bool someFunctionName(string input) {
    // … do something …
}
```

A Model for Solving Problems



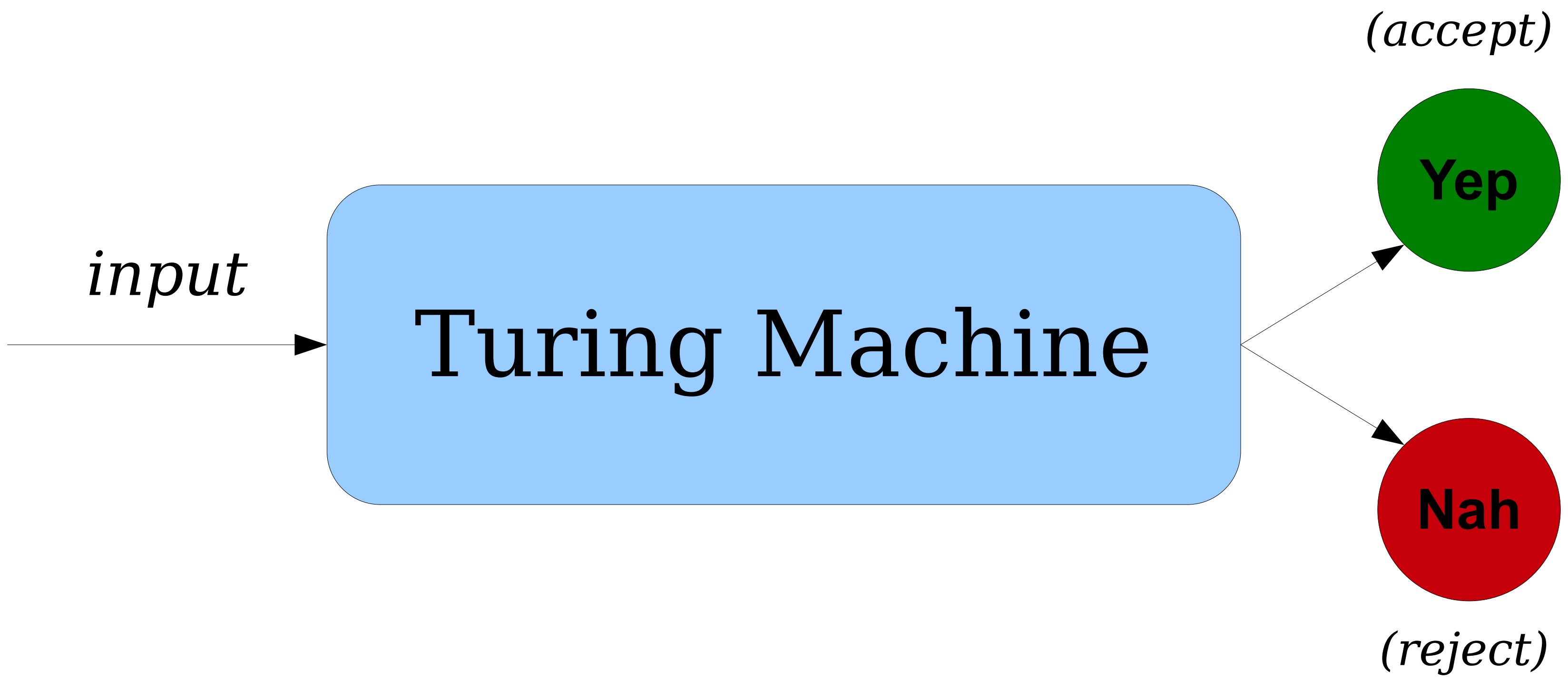

```
bool isAnBn(string input) {
    // … do something …
}
```

A Model for Solving Problems




```
bool isPalindrome(string input) {
    // … do something …
}
```

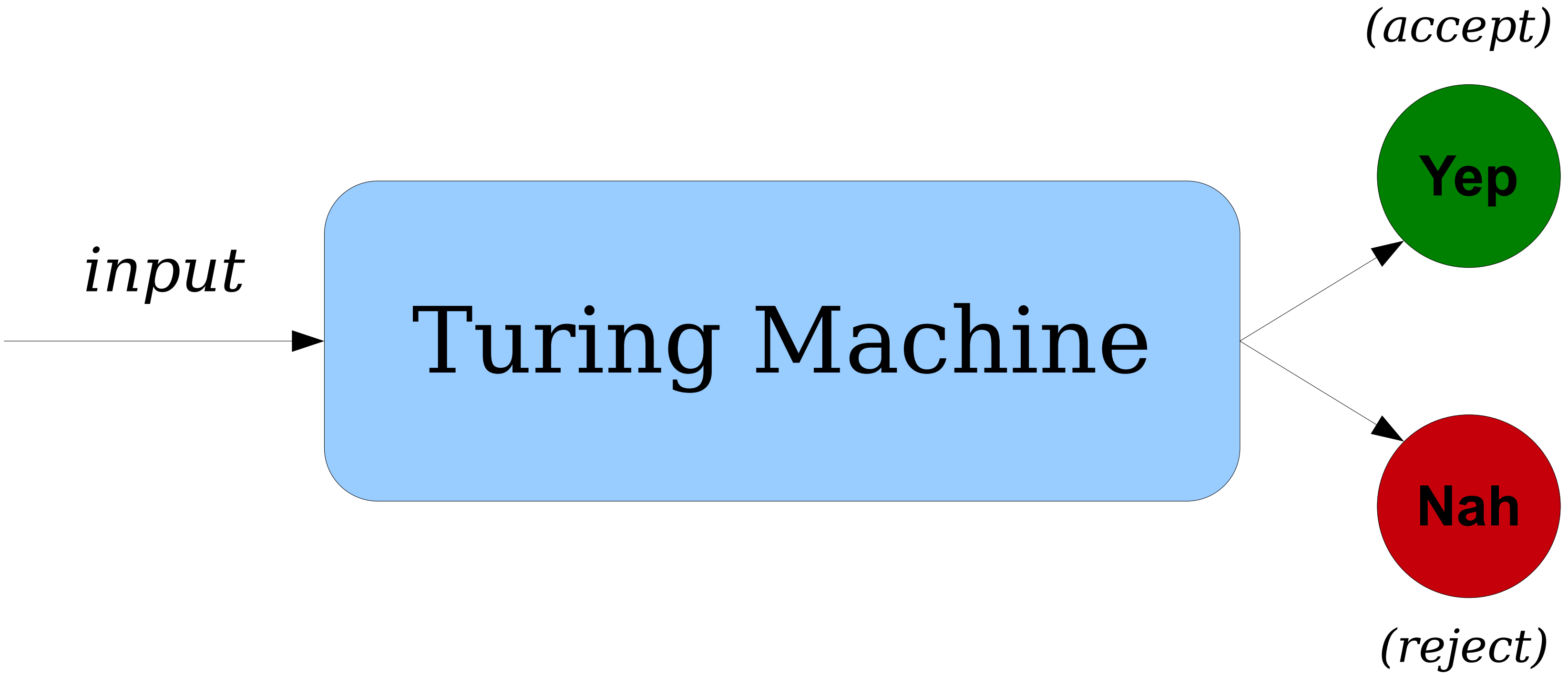
A Model for Solving Problems




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

⟨  ⟩ = 11011100101l…110

Object Encodings

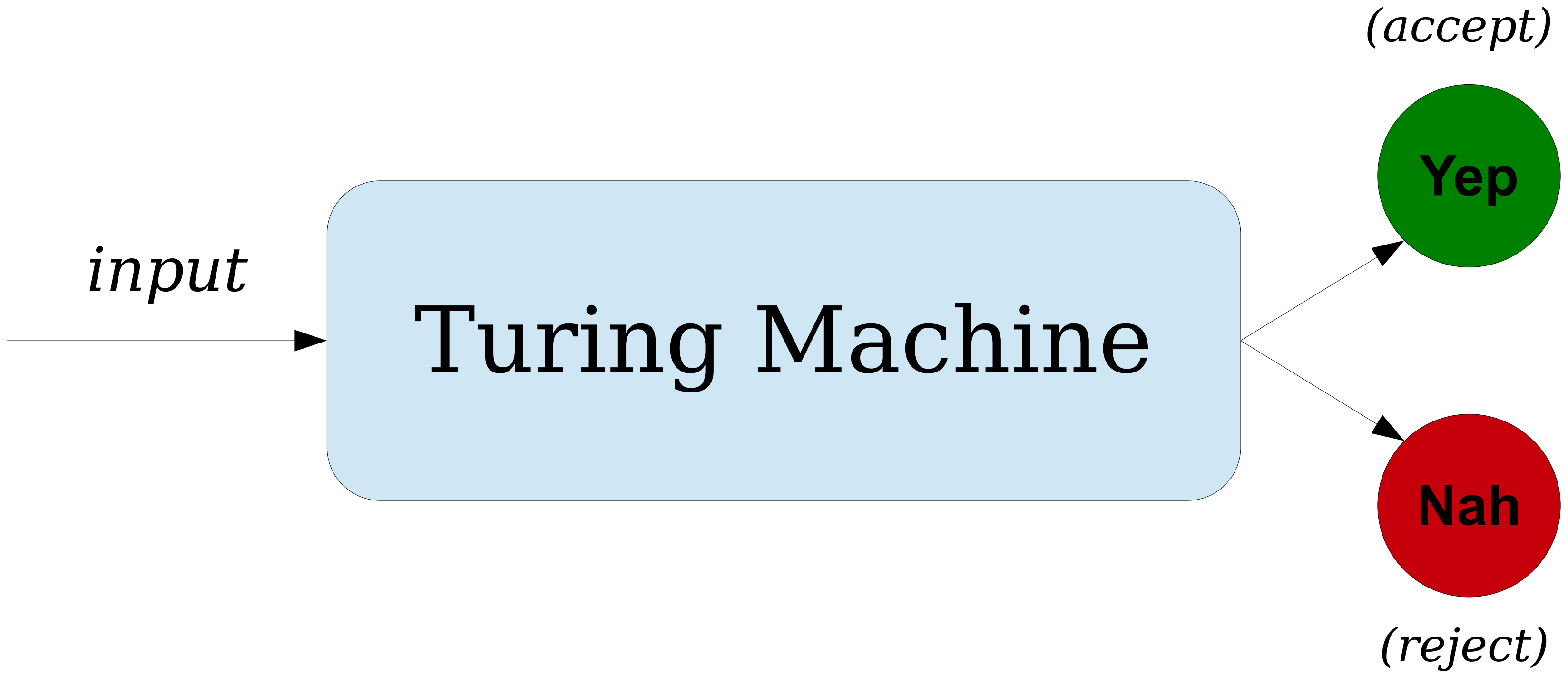
- If *Obj* is some mathematical object that is *discrete* and *finite*, then we'll use the notation **⟨*Obj*⟩** to refer to some way of encoding that object as a string.
- Think of ⟨*Obj*⟩ like a file on disk – it encodes some high-level object as a series of characters.

⟨  ⟩ = 00110101001…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.

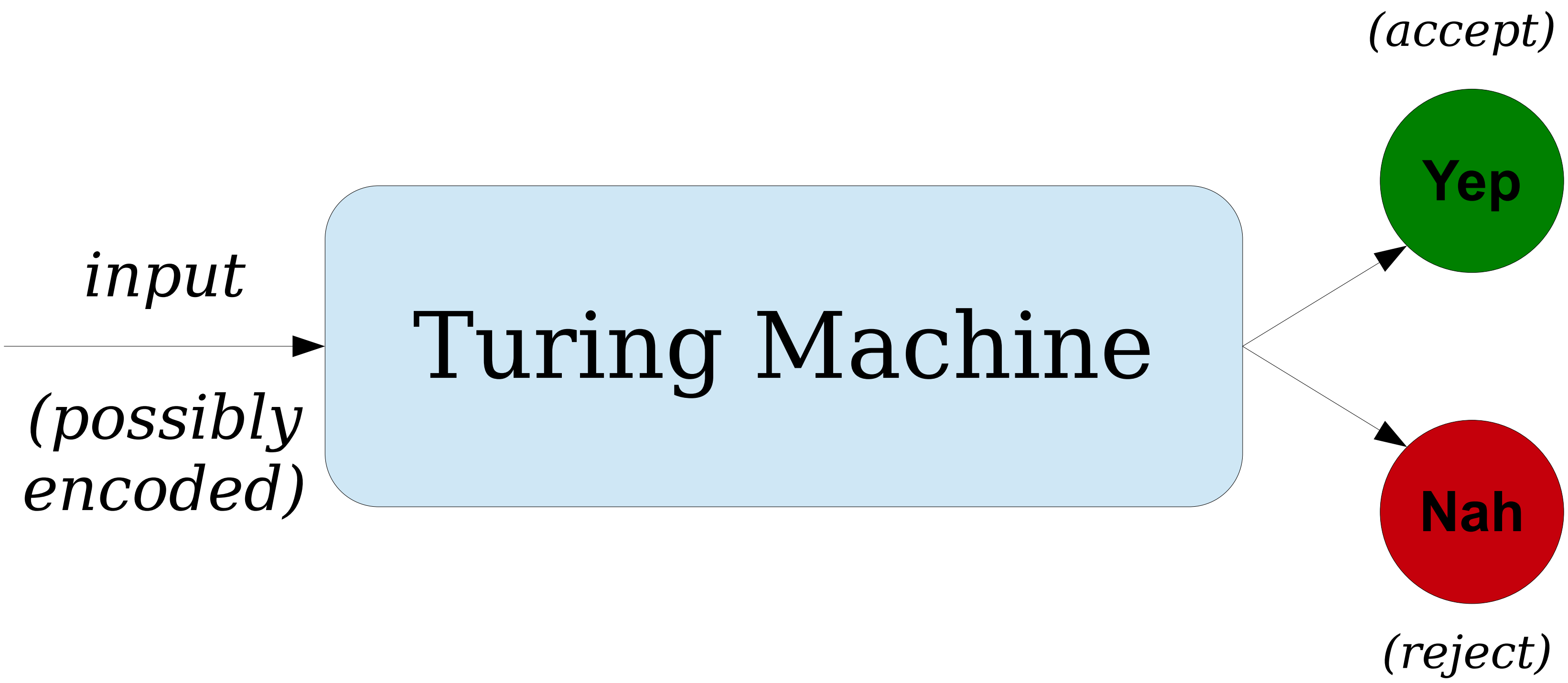
A Model for Solving Problems




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

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

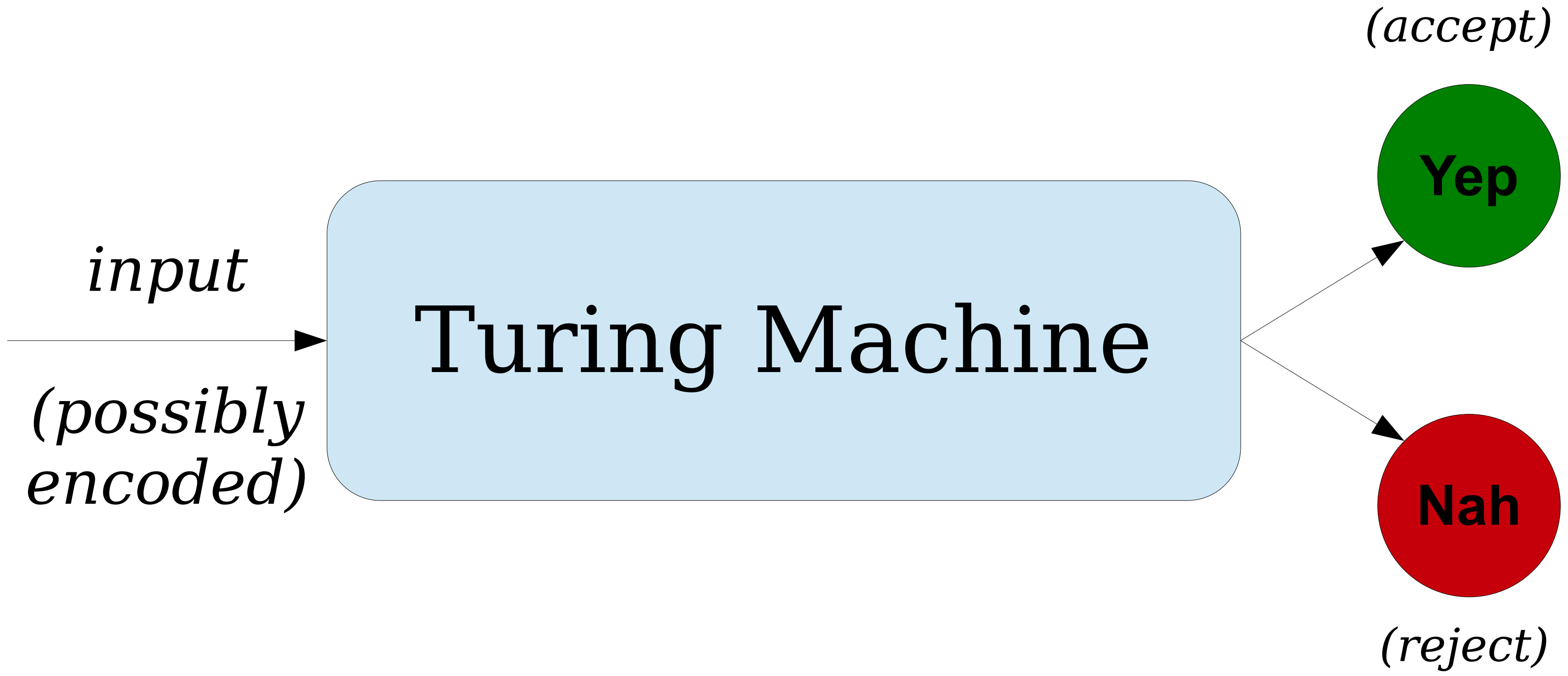
A Model for Solving Problems




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

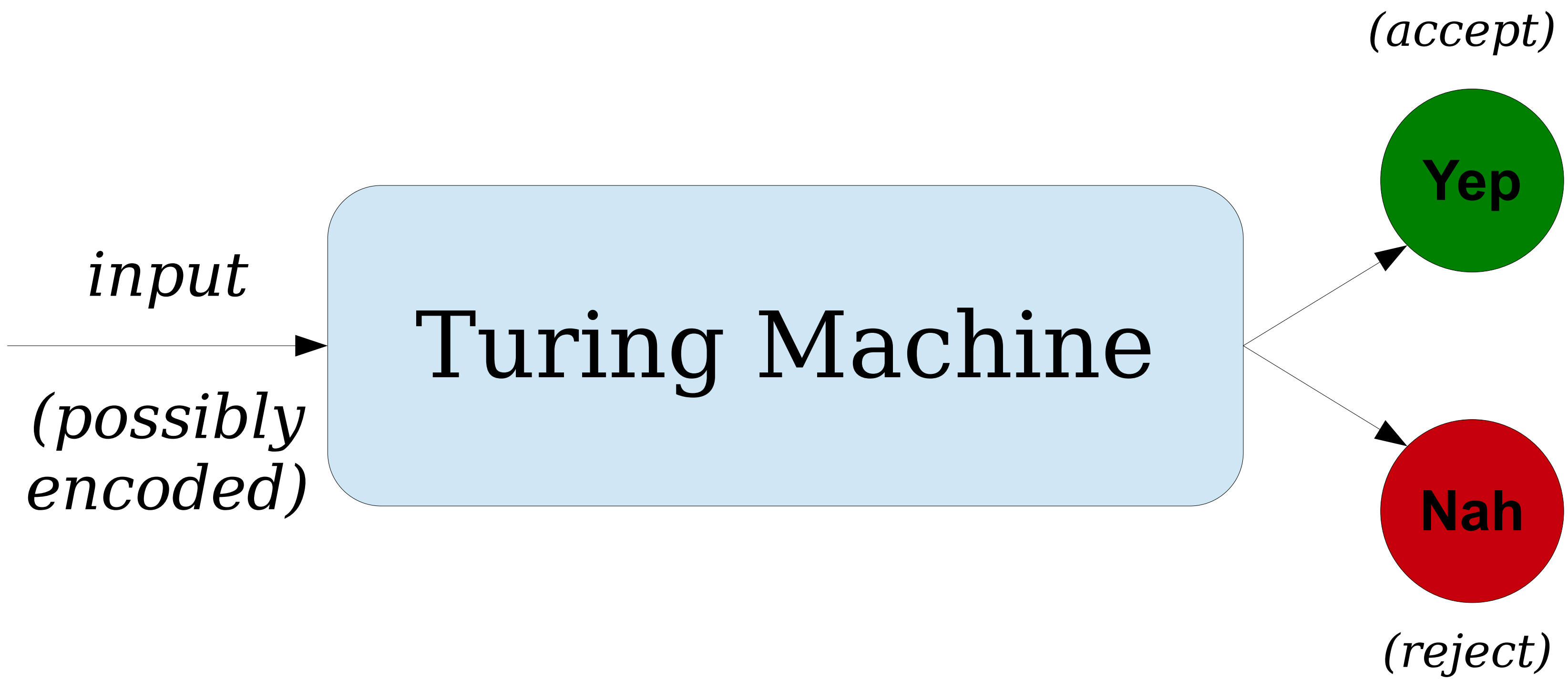
Internally, this is a sequence of **0**s and **1**s.

A Model for Solving Problems



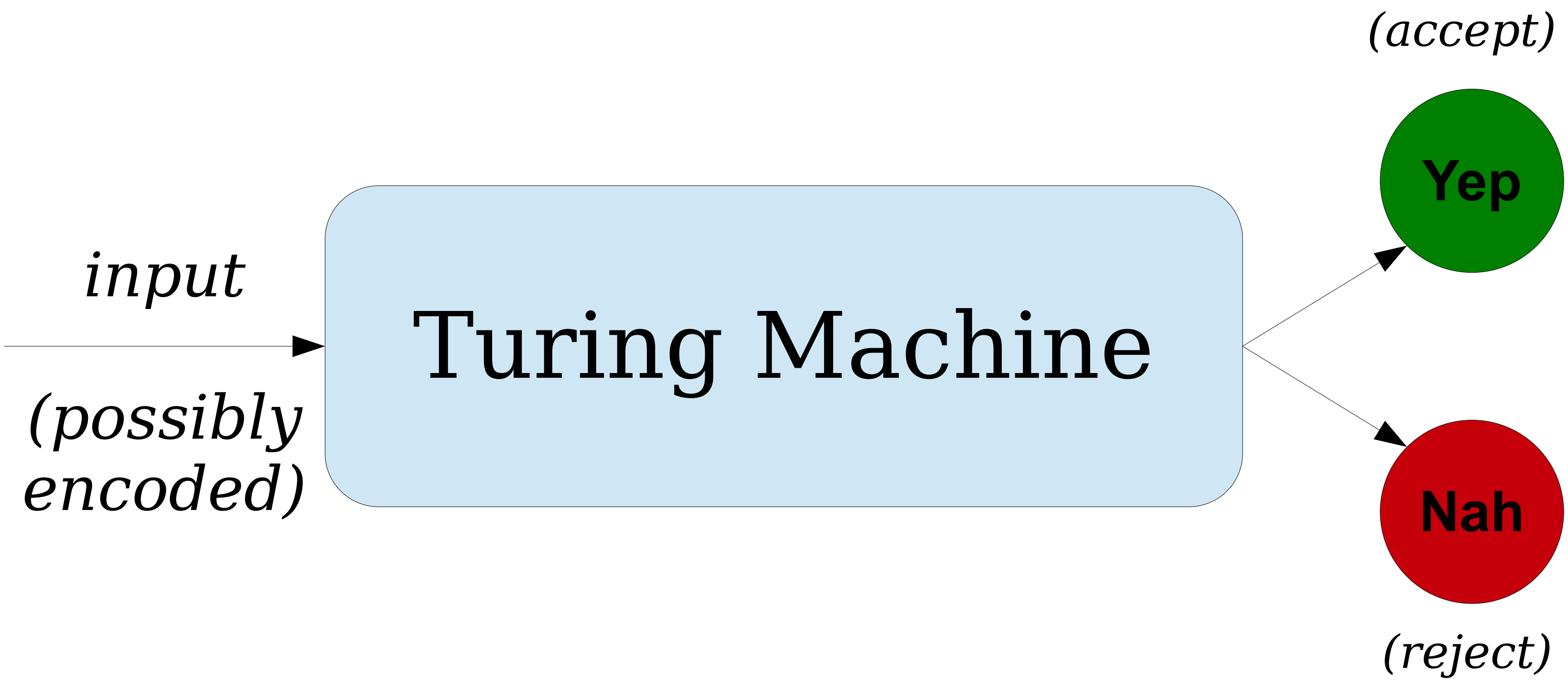

```
bool containsCat(Picture P) {
    // … do something …
}
```

A Model for Solving Problems




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

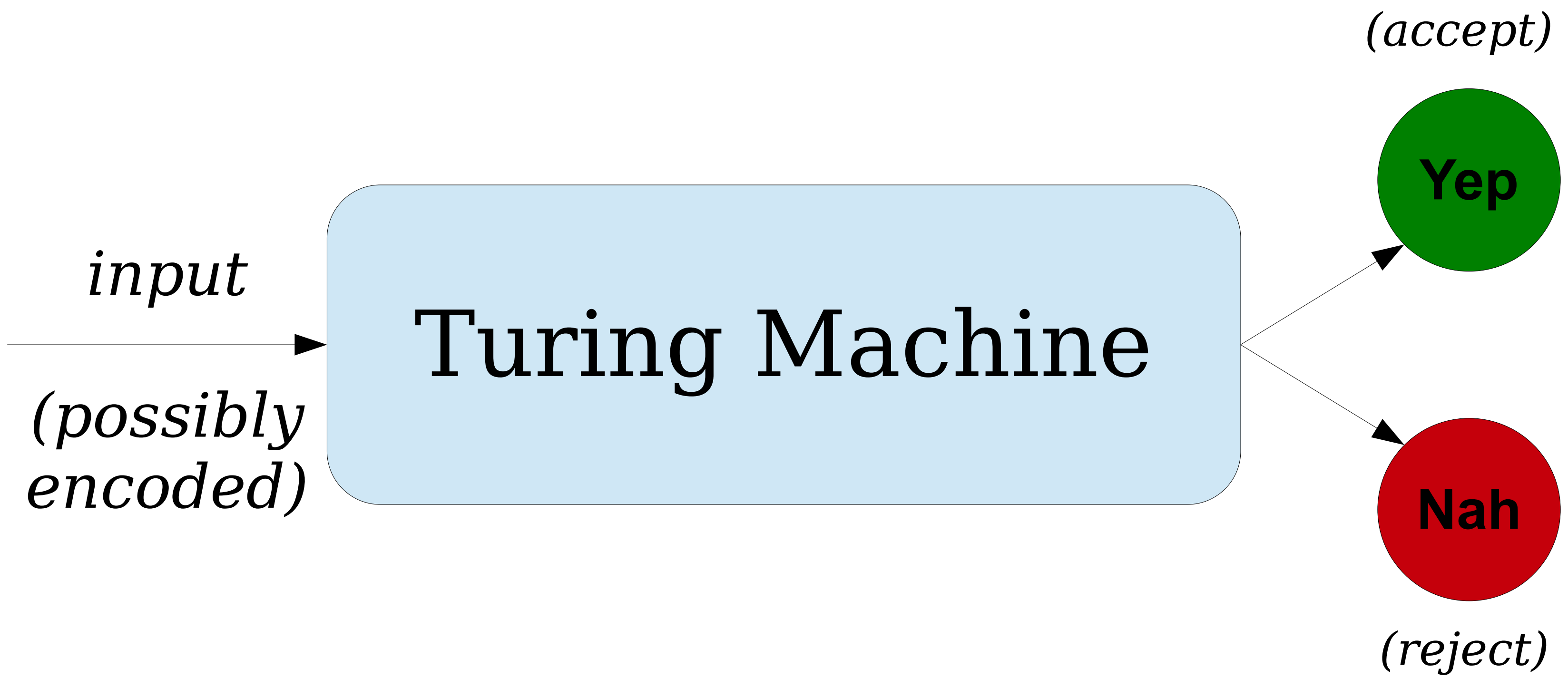
A Model for Solving Problems




```
bool isDominatingSet(Graph G, Set D) {
    // … do something …
}
```

How does this match our model?

A Model for Solving Problems



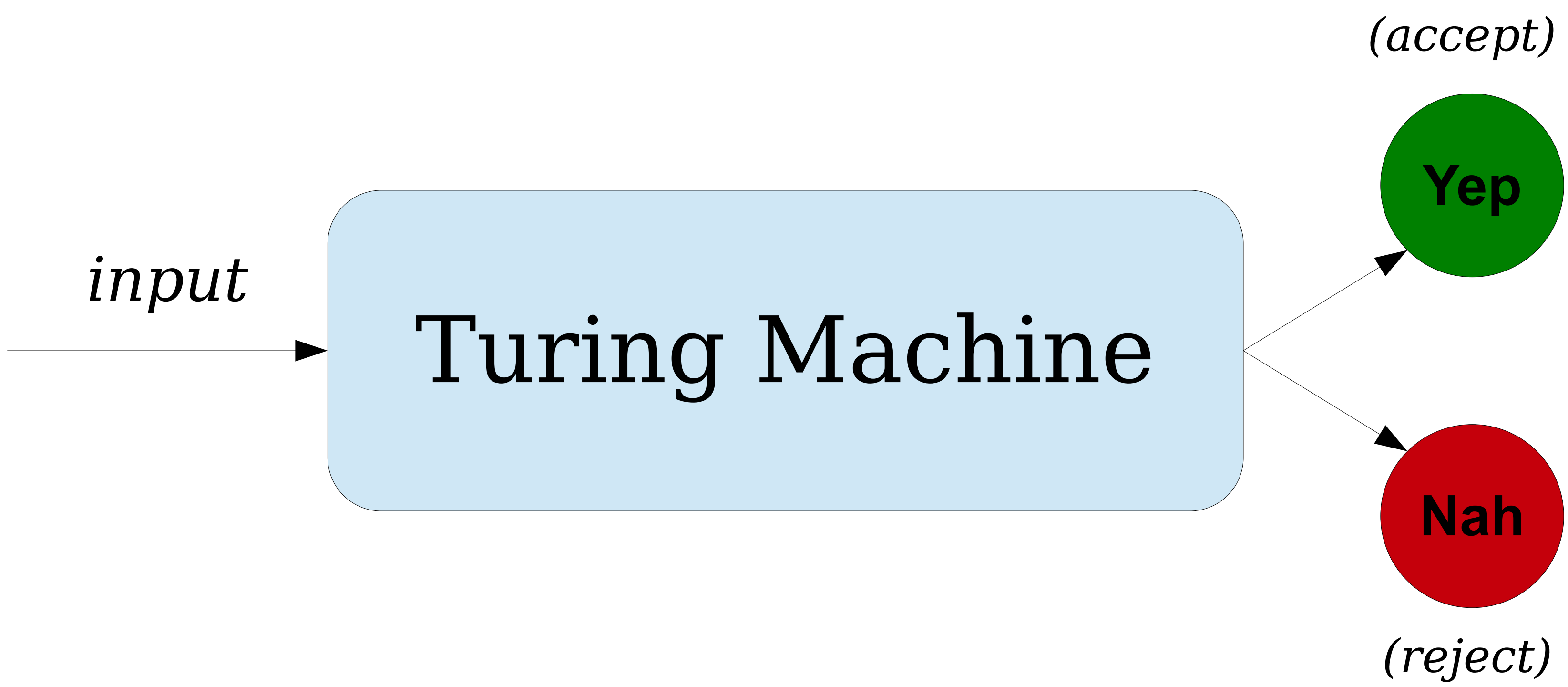

```
bool matchesRegex(string w, Regex R) {
    // … do something …
}
```

How does this match our model?

Encoding Groups of Objects

- Given a group of objects $Obj_1, Obj_2, \ldots, Obj_n$, 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 $\boldsymbol{\langle Obj_1, \ldots, Obj_n \rangle}$.

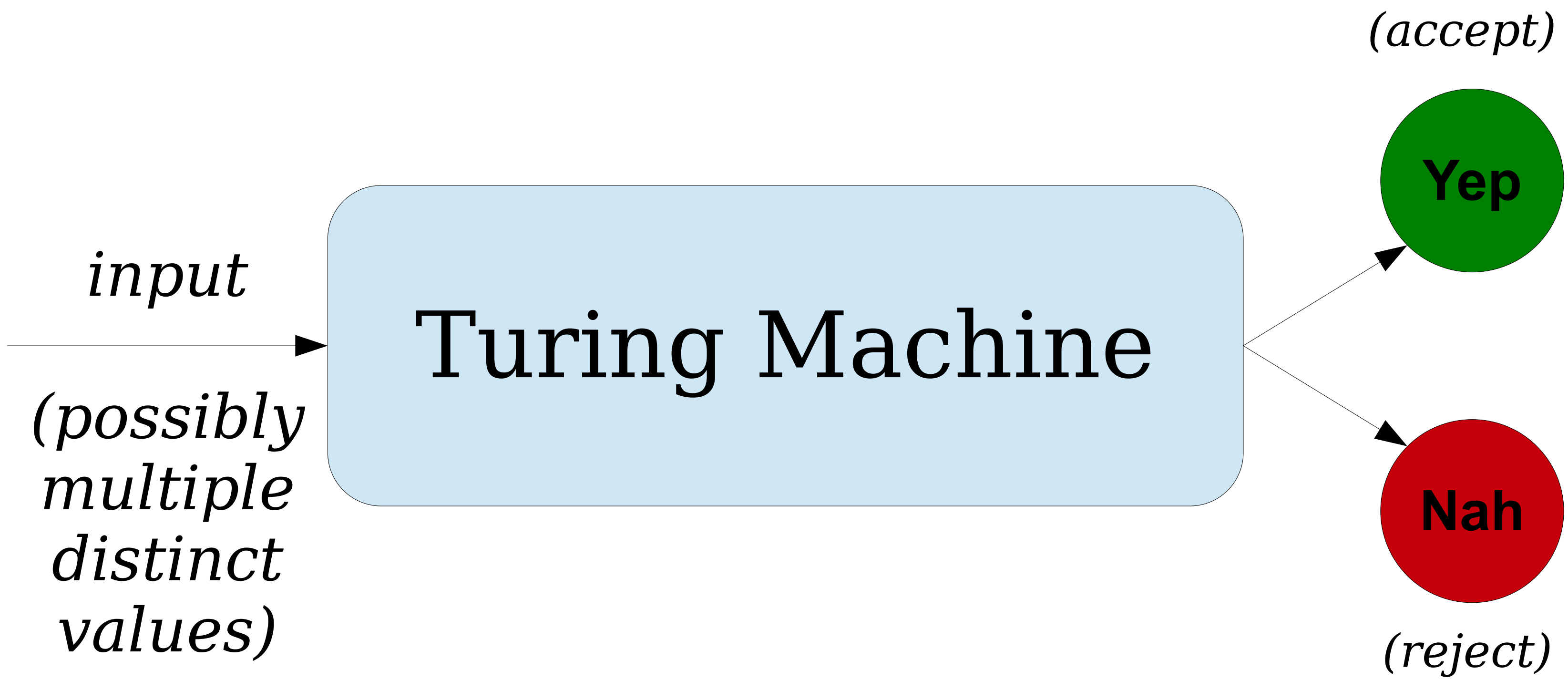
A Model for Solving Problems




```
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?

Time-Out for Announcements!

Problem Set Five Graded

- Your diligent and hardworking TAs have finished grading PS5. Grades and feedback will be available on Gradescope later today.
- As always, ***please review your feedback!*** Knowing where to improve is more important than just seeing a raw score.
- Did we make a mistake? Regrades on Gradescope will open tomorrow and are due in one week.

Remaining Problem Sets

- Problem Set 6 will be due on Friday at 5:30 PM.
- Problem Set 7 (our final problem set!) will be out on Friday at 6:00 PM and will be due ***Wednesday, August 14th*** at 5:30 PM.
  - This assignment will be shorter since you don’t have a full week to work on it.

Final Exam

- If you have exam accommodations, you will receive an email from Anthony later today with your arrangement. Please reach out if you don’t receive it.
- Next Wednesday’s lecture will be a review session

Back to CS103!

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?

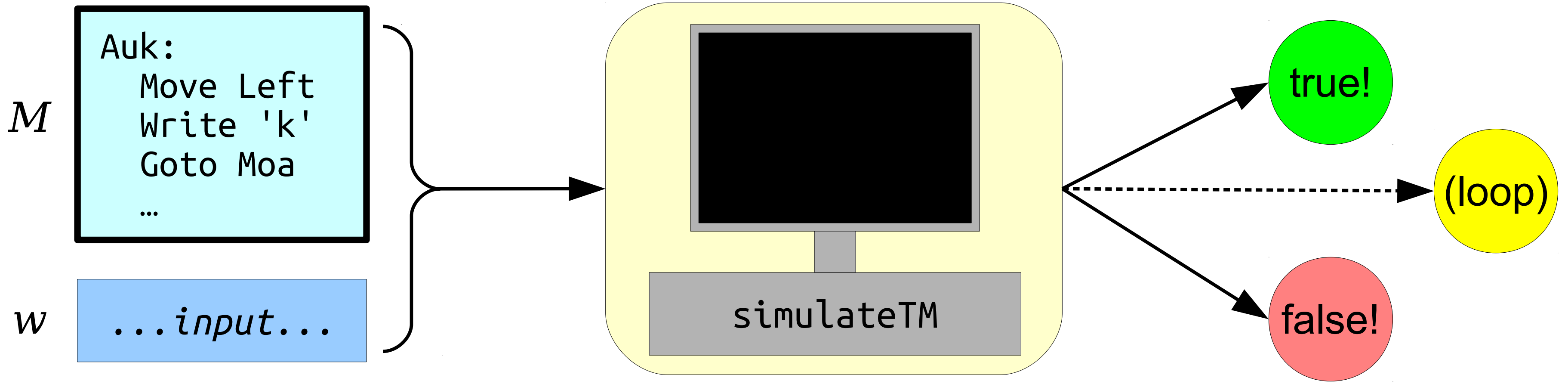
Can we make a “reprogrammable Turing machine?”

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

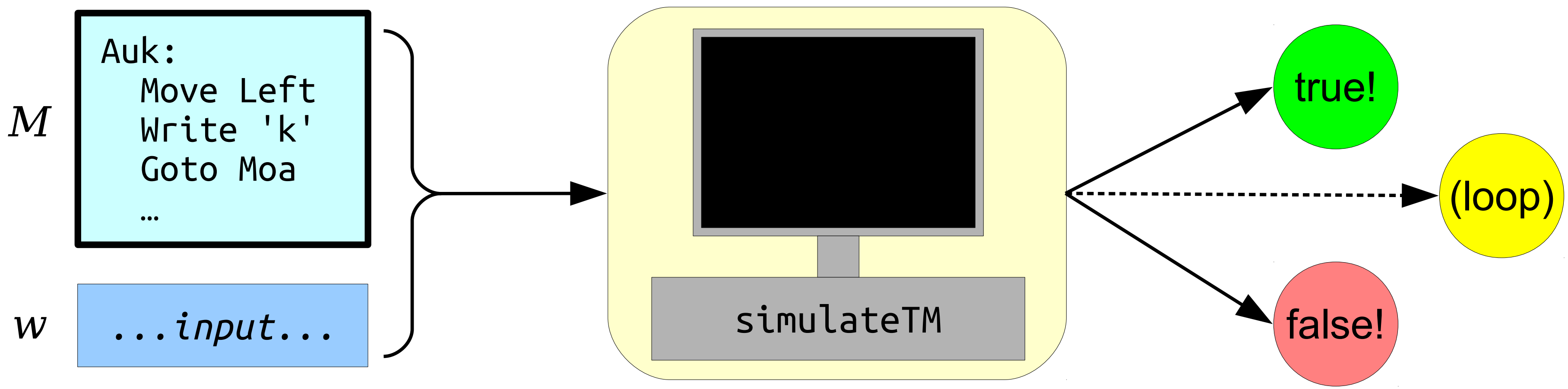
```
bool simulateTM(TM M, string w)
```

  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.



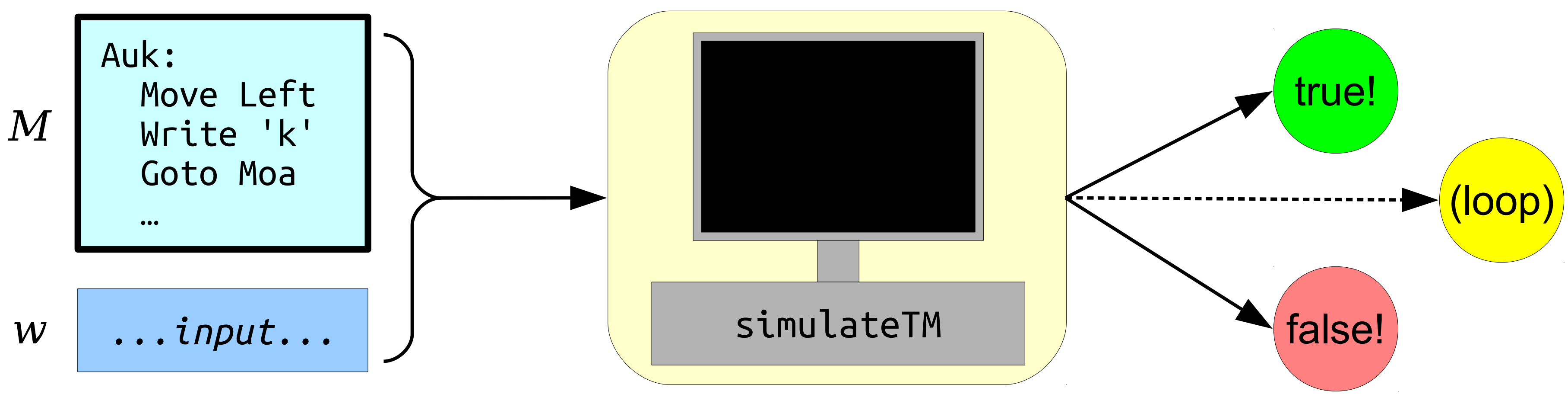
A TM Simulator

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



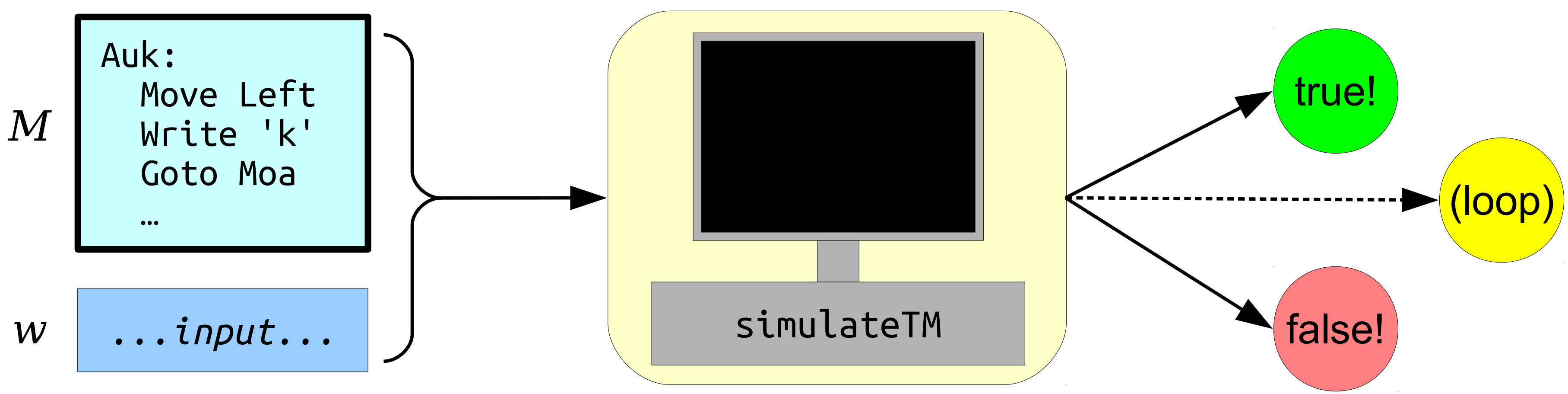
A TM Simulator

- 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.



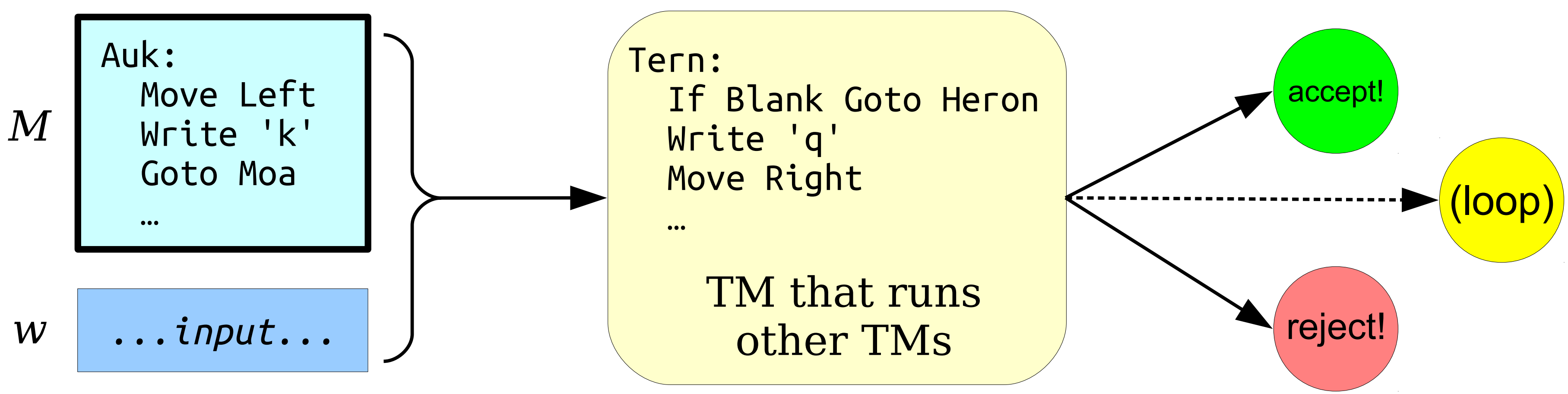
A TM Simulator

- 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?



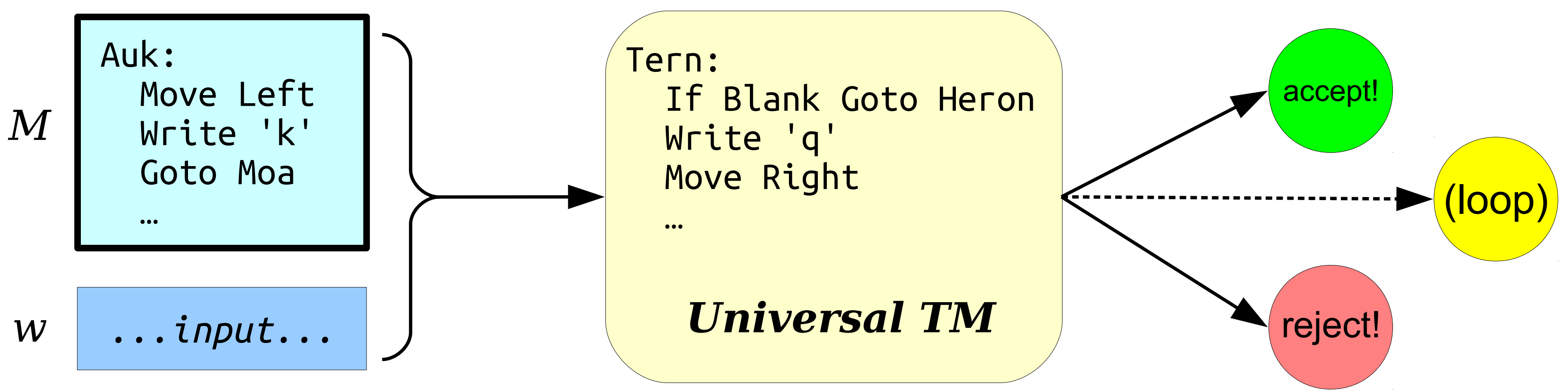
A TM Simulator

- 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?



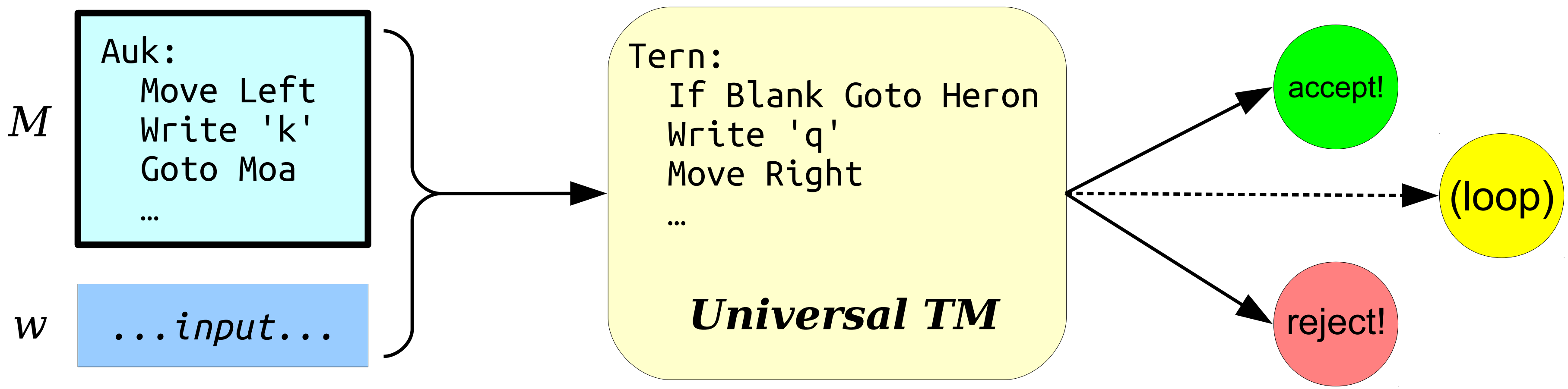
The Universal Turing Machine

- ***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 ⟨*M*, *w*⟩, where *M* is a TM and *w* is a string, will

  … accept ⟨*M*, *w*⟩ if *M* accepts *w*,

  … reject ⟨*M*, *w*⟩ if *M* rejects *w*, and

  … loop on ⟨*M*, *w*⟩ 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 ⟨*M*, *w*⟩, where *M* is a TM and *w* is a string, will
  - … accept ⟨*M*, *w*⟩ if *M* accepts *w*,
  - … reject ⟨*M*, *w*⟩ if *M* rejects *w*, and
  - … loop on ⟨*M*, *w*⟩ 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 ⟨*M*, *w*⟩, where *M* is a TM and *w* is a string, will

  … accept ⟨*M*, *w*⟩ if *M* accepts *w*,

  … reject ⟨*M*, *w*⟩ if *M* rejects *w*, and

  … loop on ⟨*M*, *w*⟩ 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

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

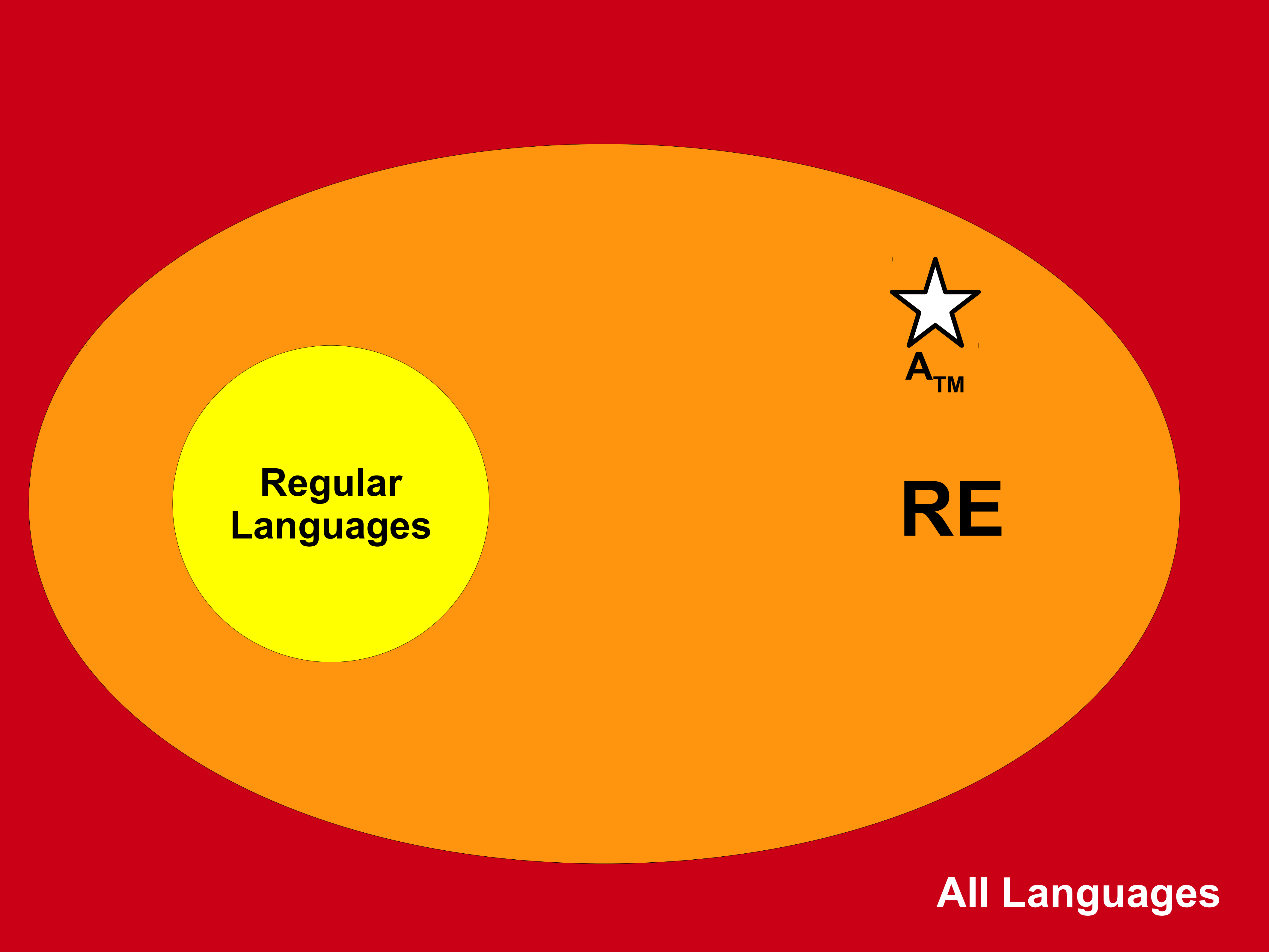
The Language A_{TM}

A_{TM} = { ⟨*M*, *w*⟩ | *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 ⟨*M*, *w*⟩.
  - $\langle M, w\rangle \in A_{TM}$.



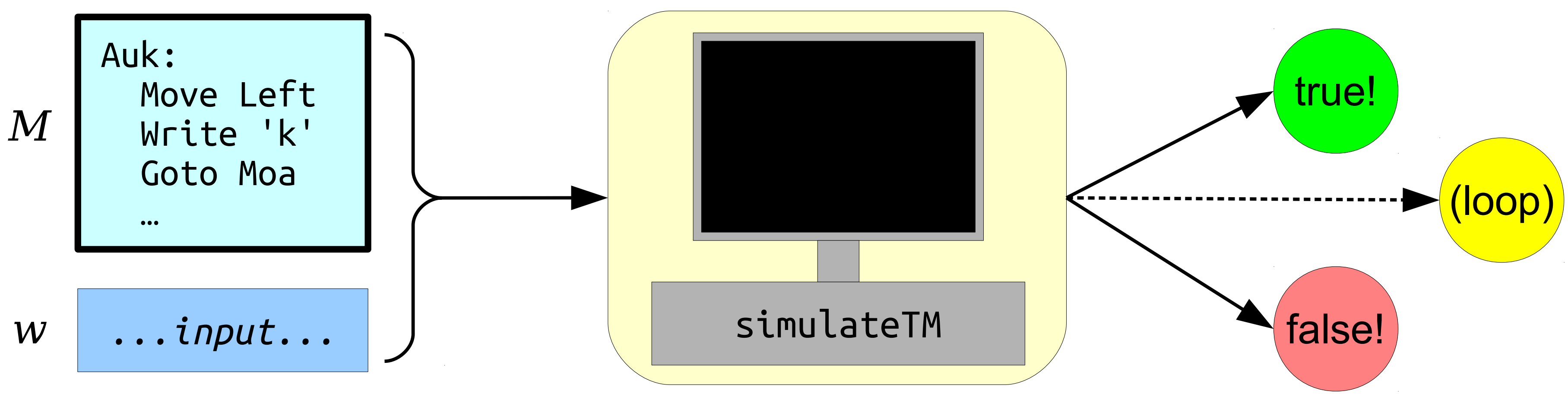
Regular Languages
A_{TM}
RE
All Languages

Uh… so what?

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

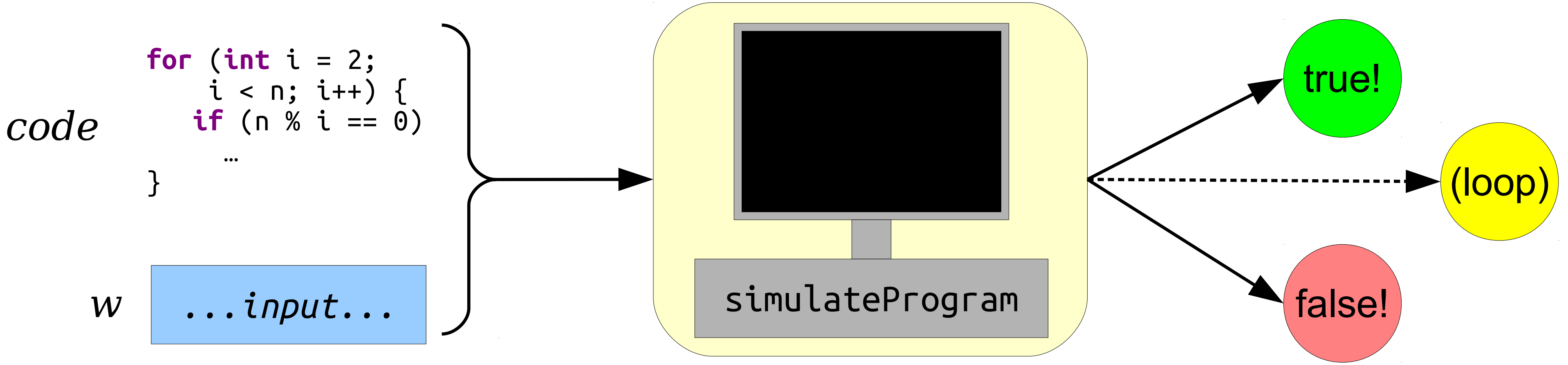
Why Does This Matter?

- 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?



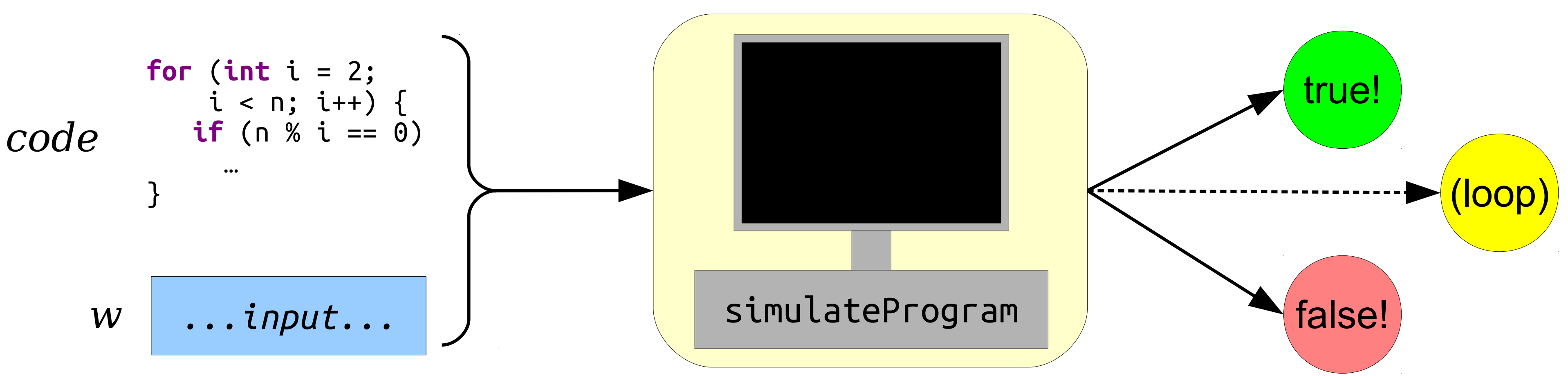
Why Does This Matter?

- 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?