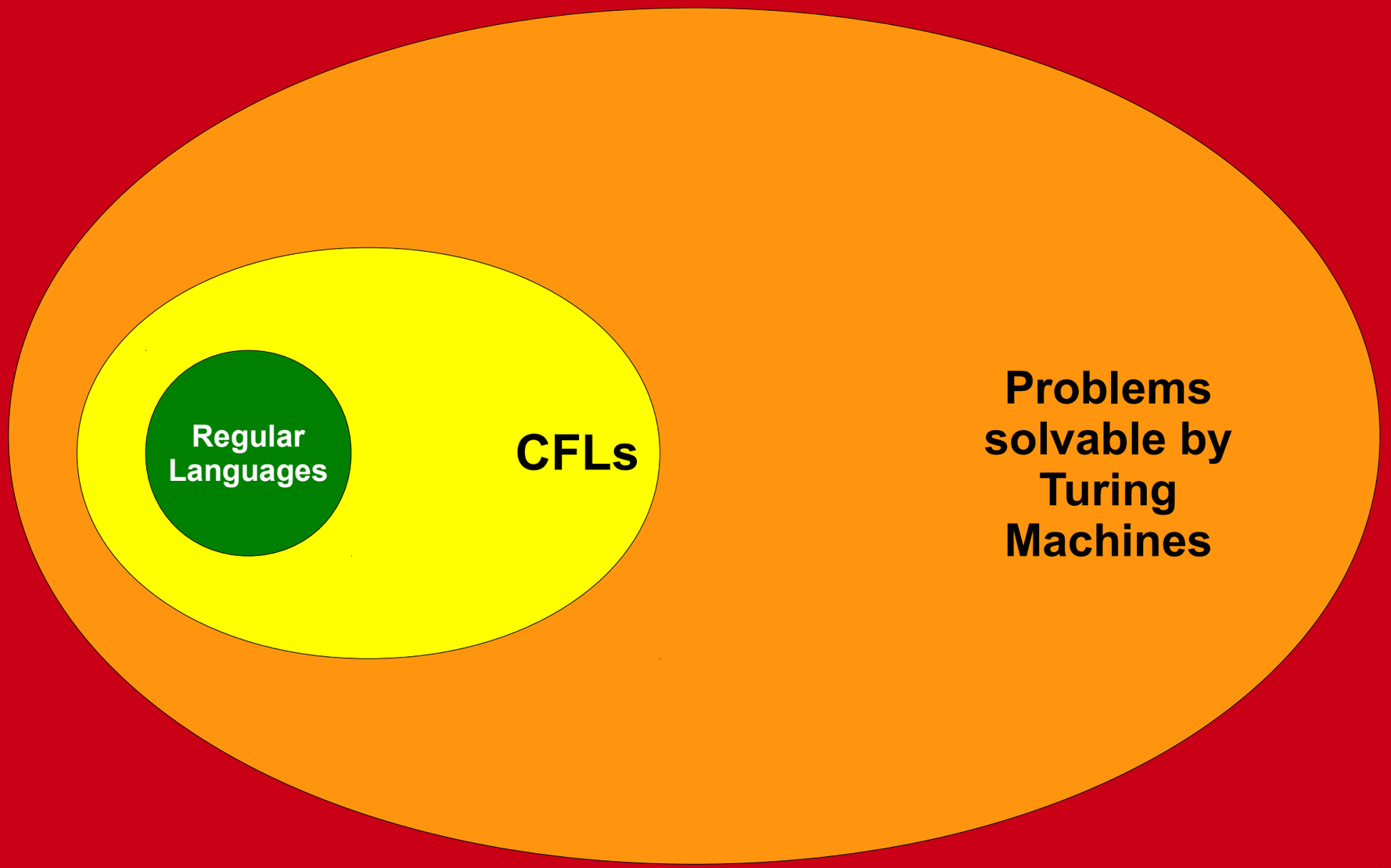
Turing Machines

Part Three

Outline for Today

- ***Why Languages and Strings?***
  - We've been using languages to model problems. Why?
- ***Universal Machines***
  - A single computer that can compute anything computable anywhere.
- ***Self-Referential Software***
  - Programs that compute on themselves.

Recap from Last Time


Regular Languages
CFLs
Problems solvable by Turing Machines
All Languages

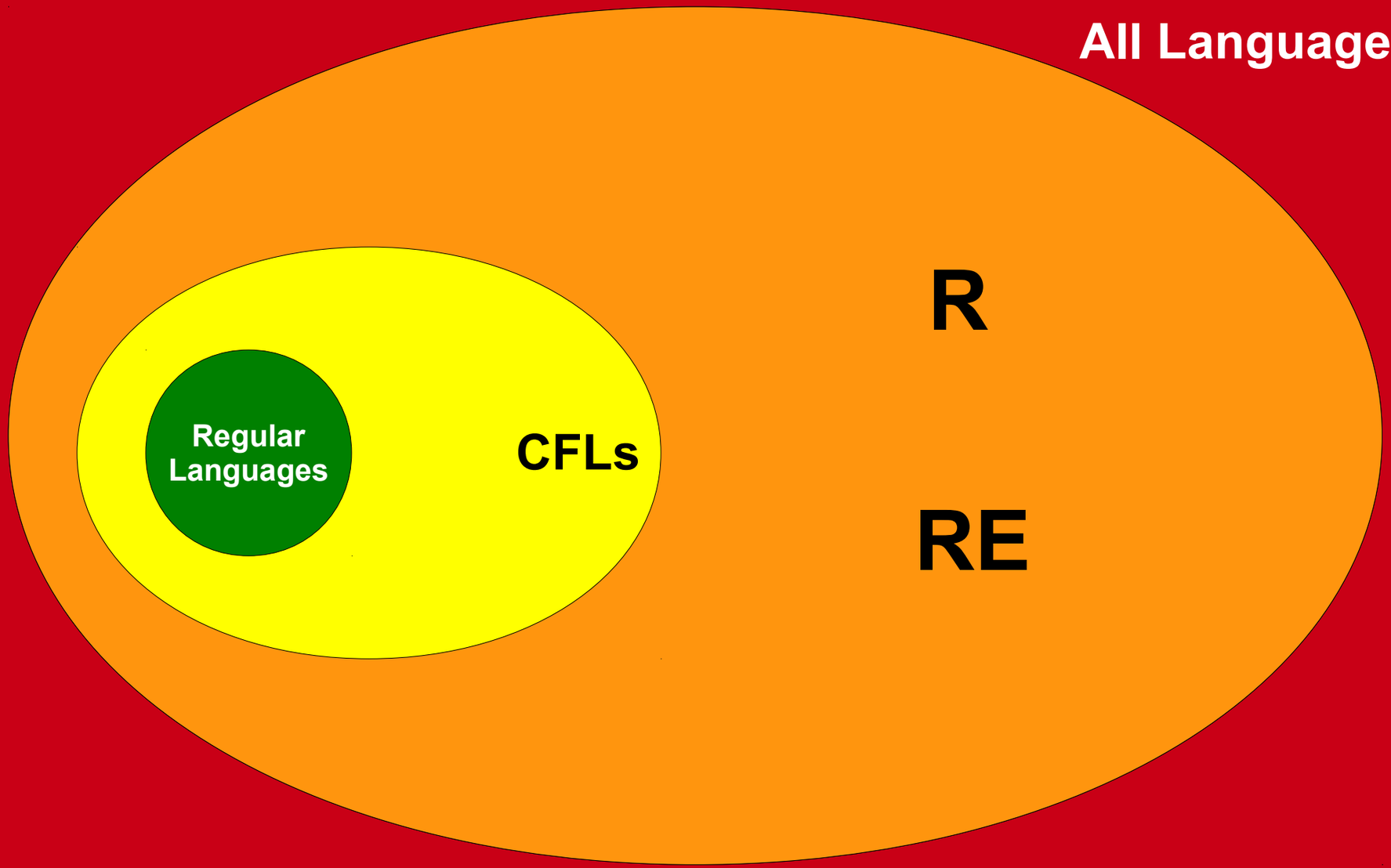
Recognizable Languages (**RE**)

- A language is called ***recognizable*** if it is the language of some TM.
  - For any $w \in \mathscr{L}(M)$, M accepts w.
  - For any $w \notin \mathscr{L}(M)$, M does not accept w.
    - M **might reject**, or it **might loop forever**.

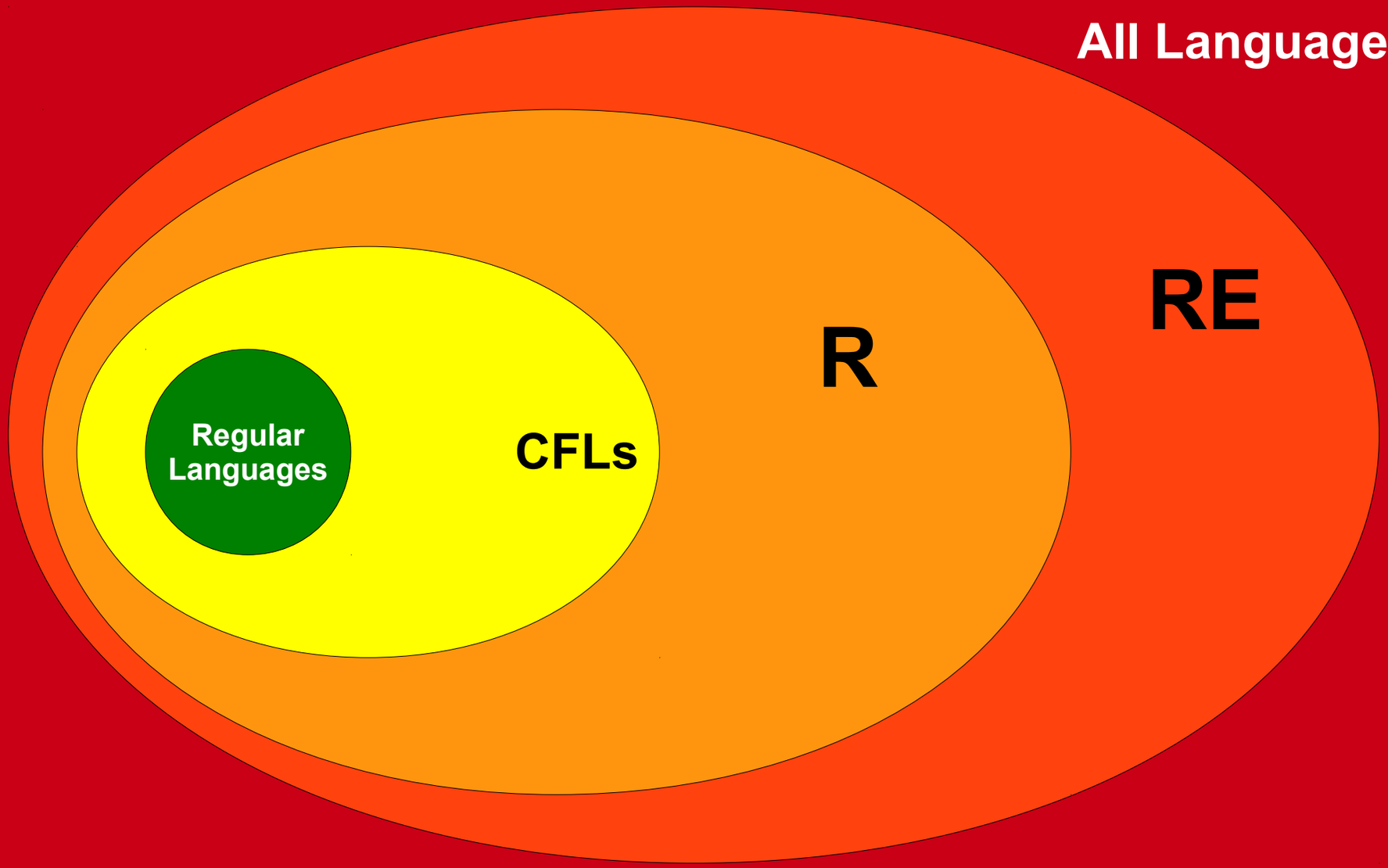
Decidable Languages (**R**)

- A language L is called ***decidable*** if there exists a decider M such that $\mathscr{L}(M) = L$.
    - Decider machines are implemented in a way that they have no danger/possibility of looping forever.

Which Picture is Correct?



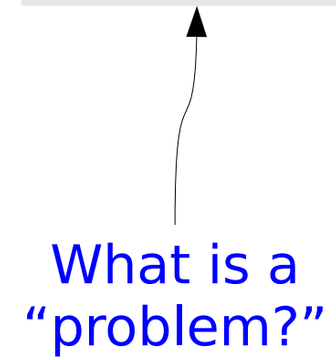
Which Picture is Correct?



New Stuff!

Strings, Languages, and Encodings

What problems can we solve with a computer?



What is a
“problem?”

Decision Problems

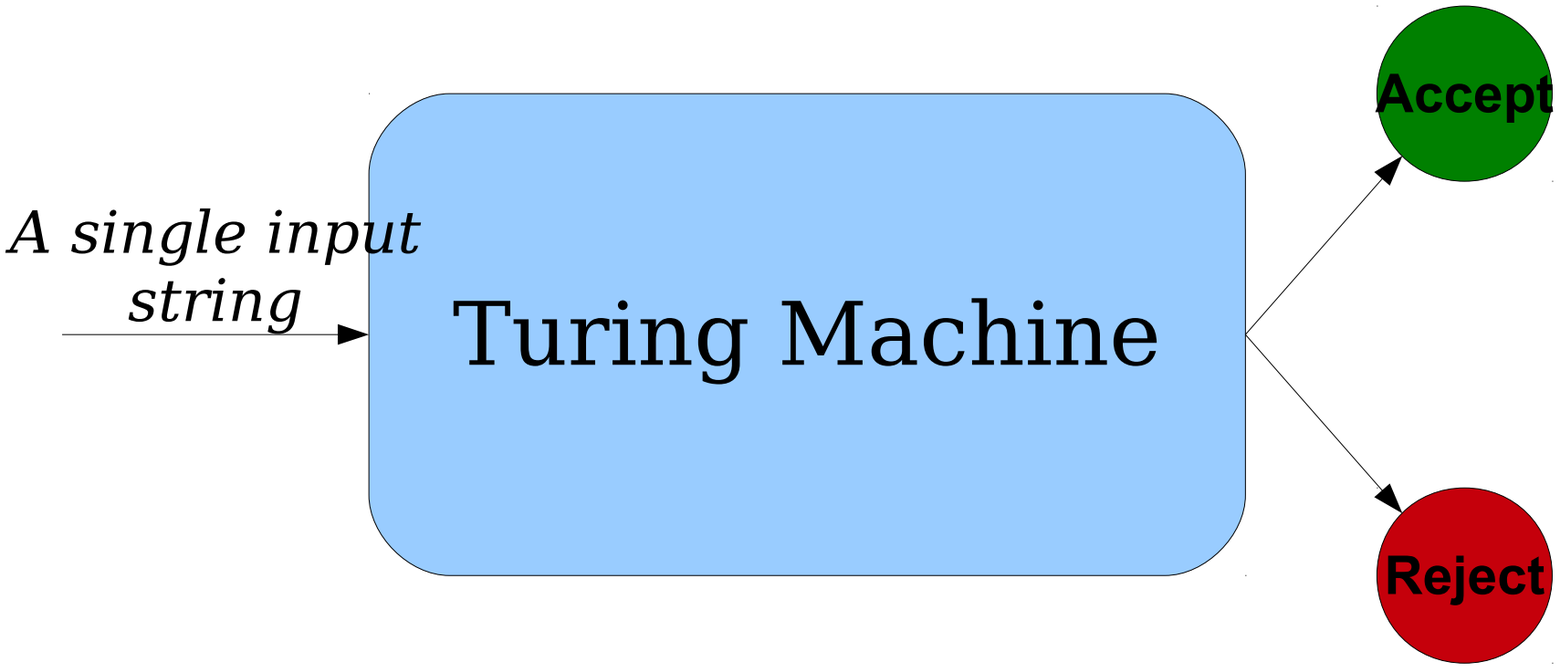
- A ***decision problem*** is a type of problem where the goal is to provide a yes or no answer.
- Example: Bin Packing

  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?

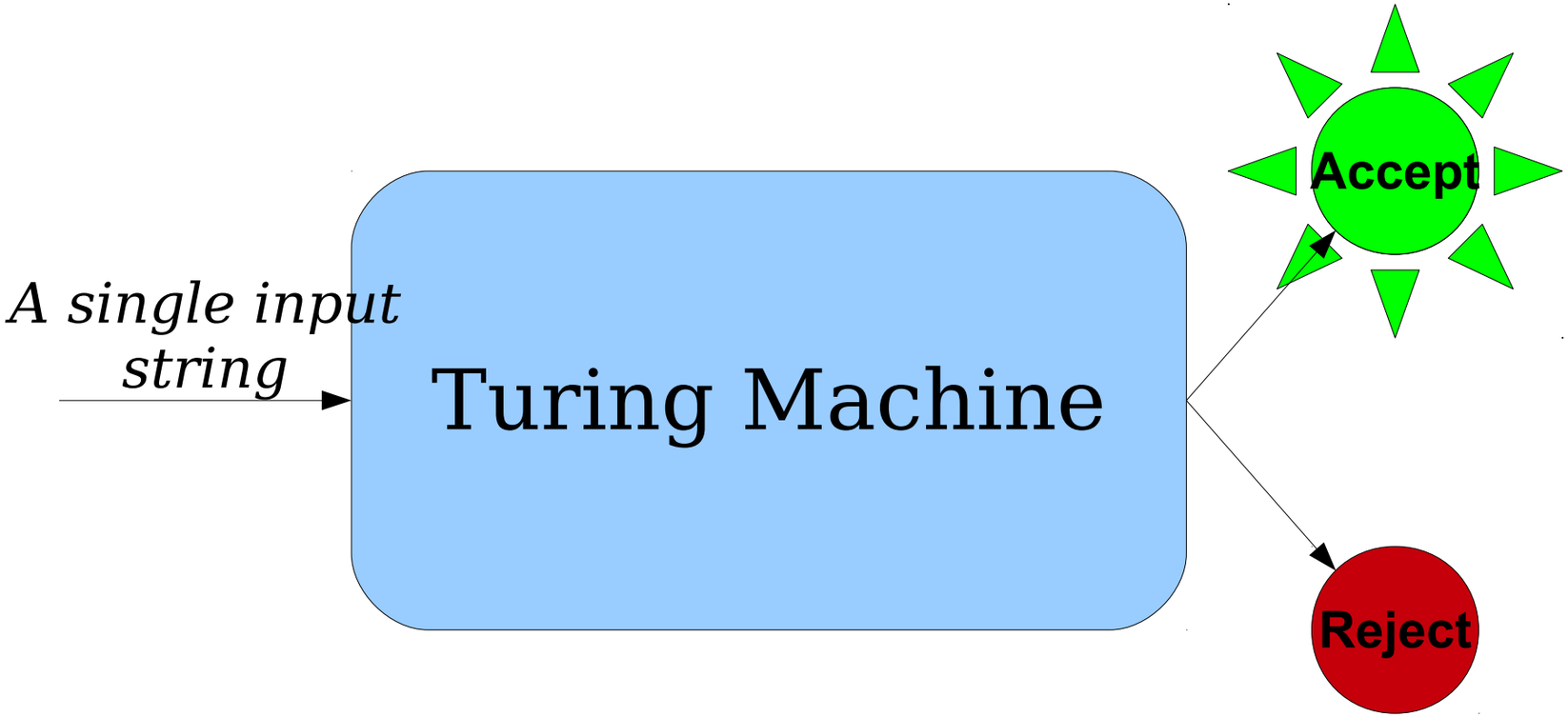
- Example: Dominating Set Problem

  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?

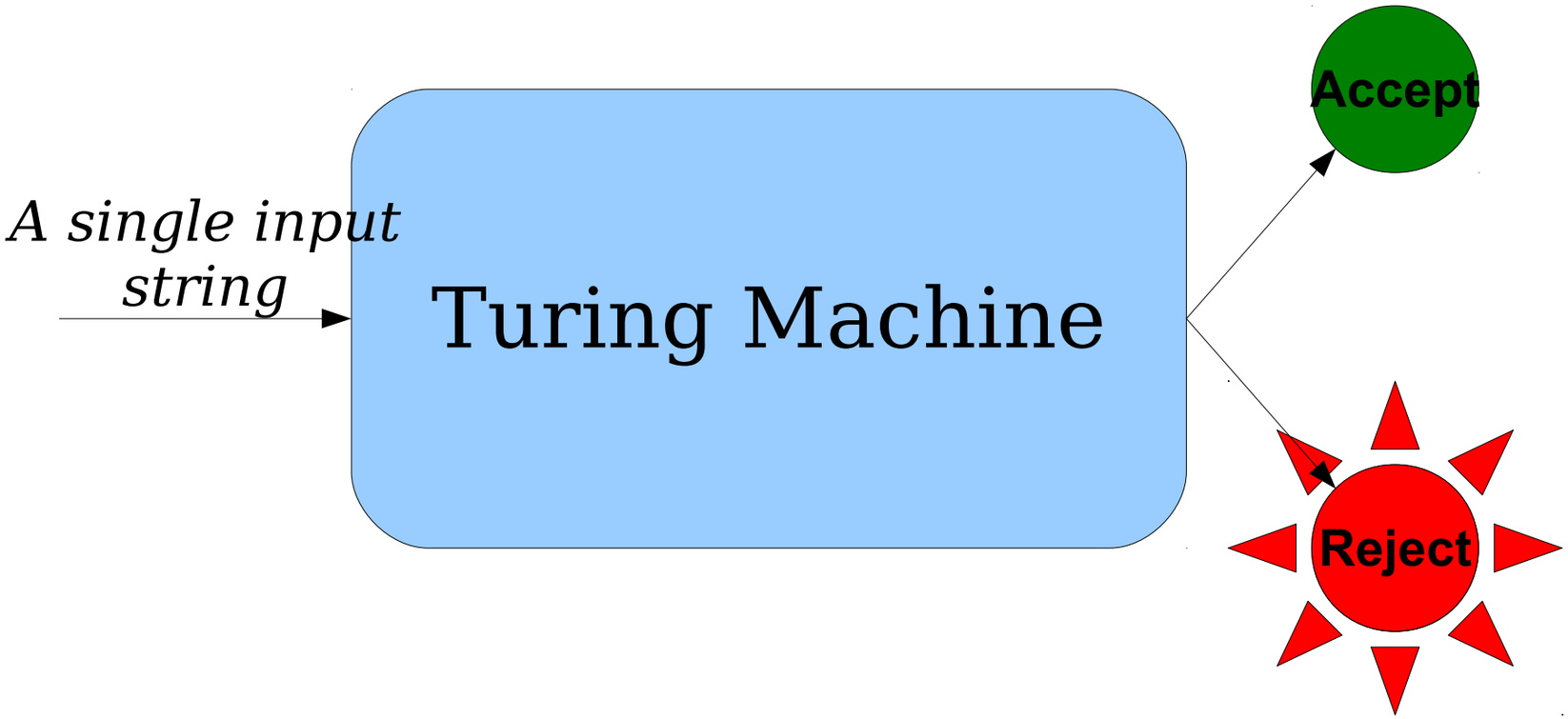
A Model for Solving Problems



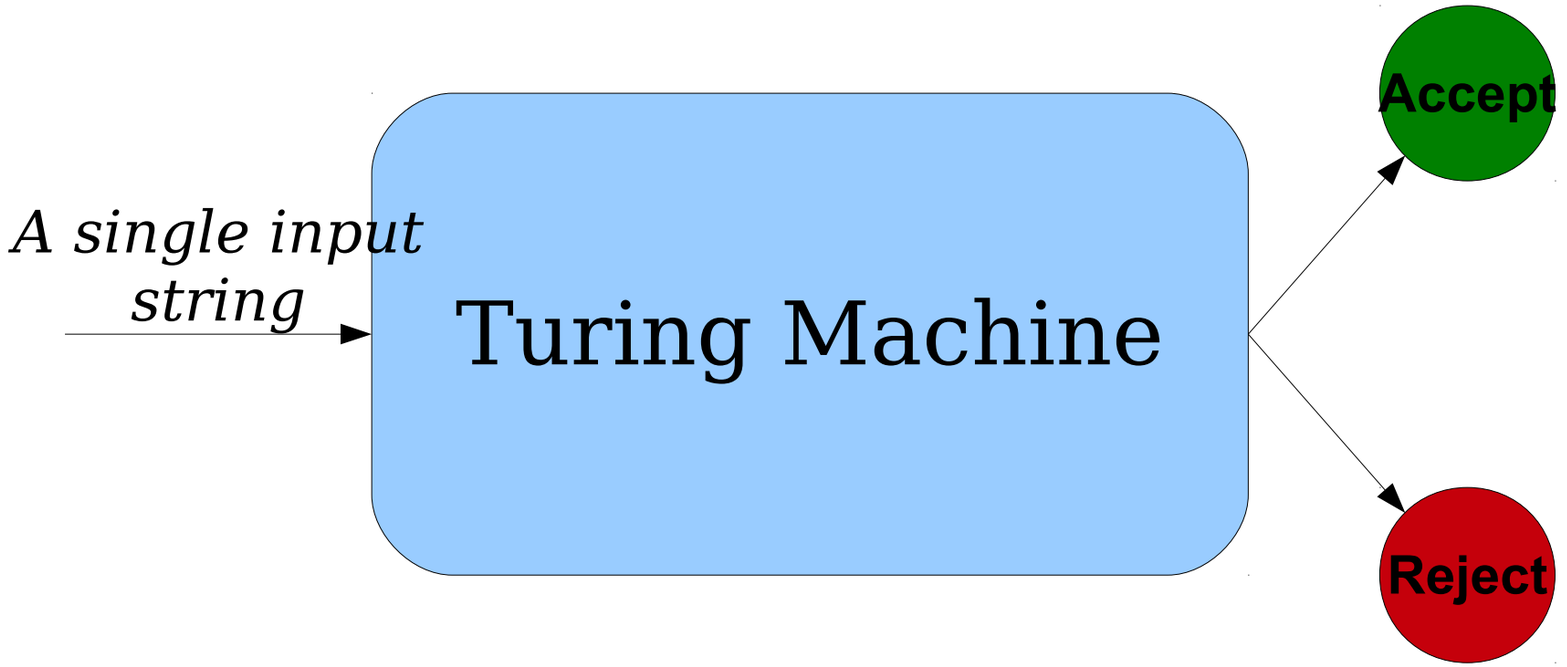
A Model for Solving Problems



A Model for Solving Problems



A Model for Solving Problems

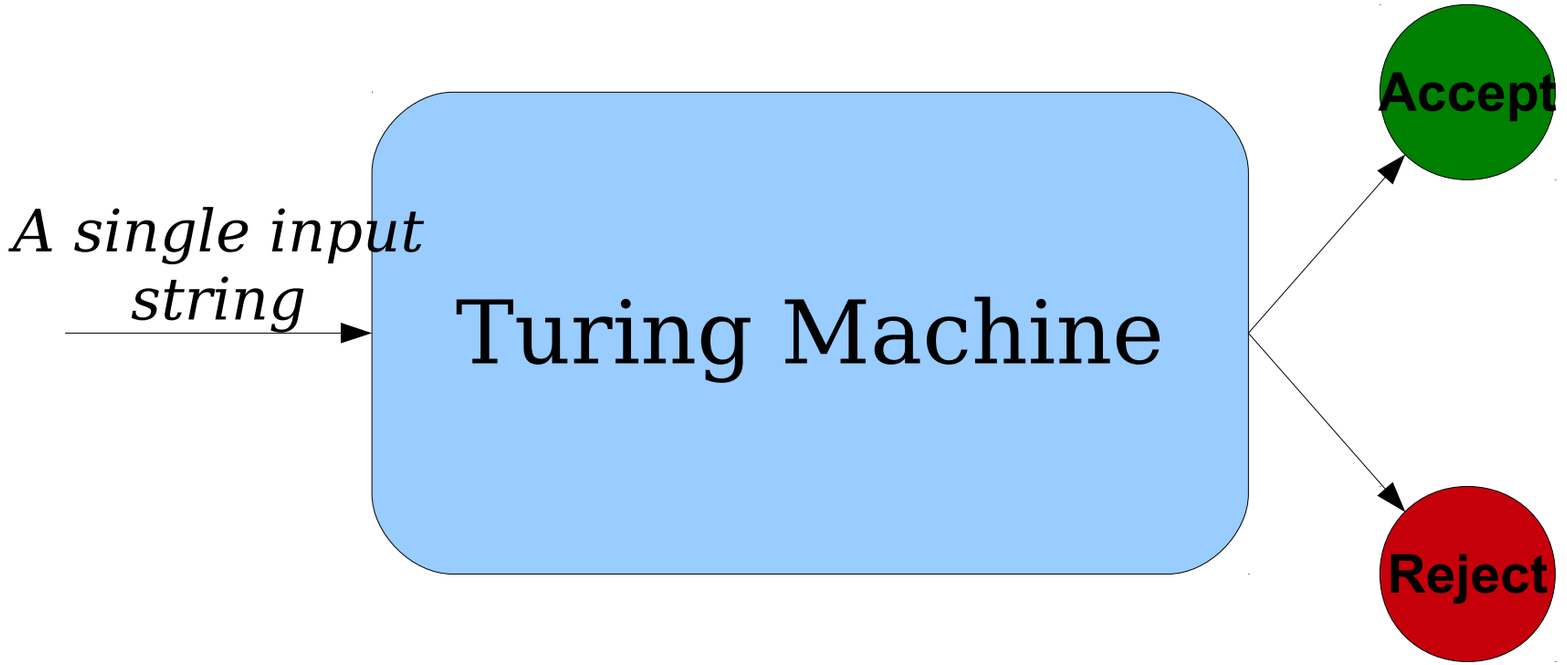



```
bool someFunctionName(string input) {

    // … do something …

}
```

A Model for Solving Problems

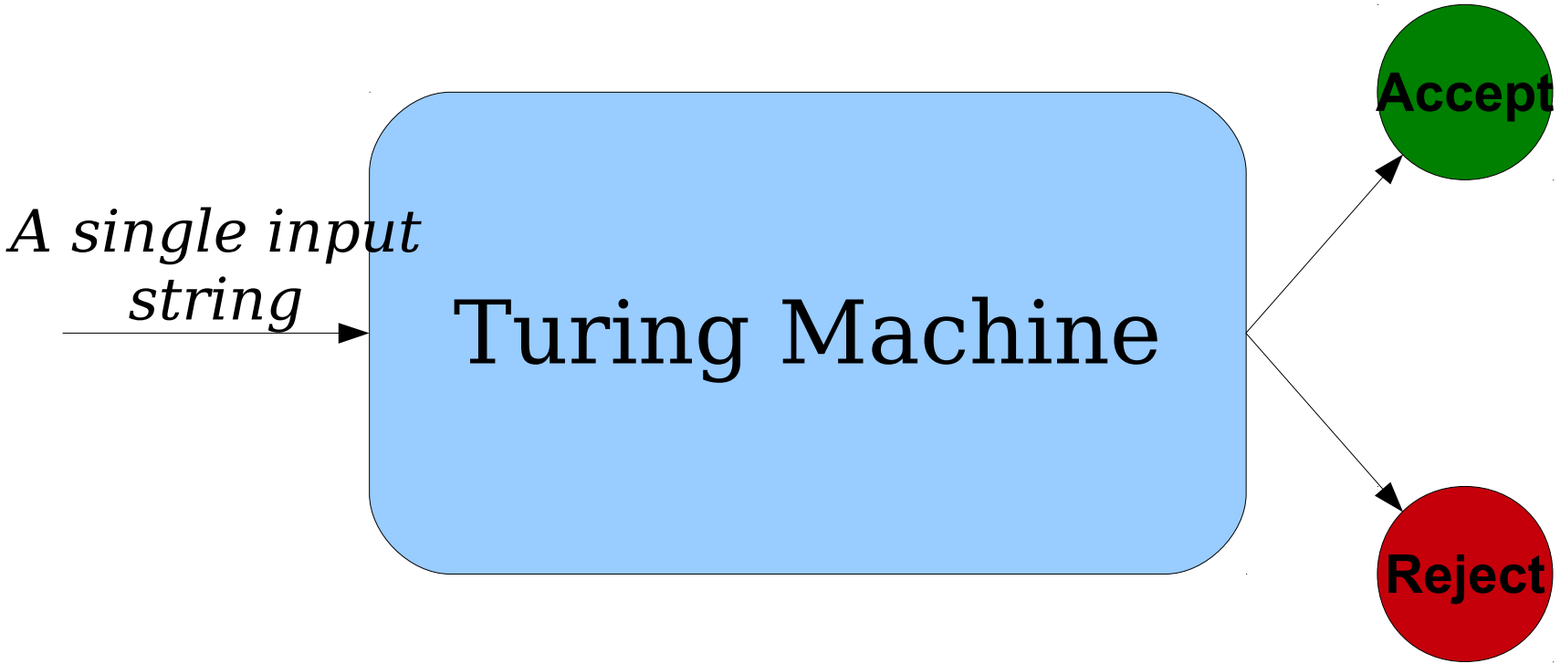



```
bool isAnBn(string input) {

    // … do something …

}
```

A Model for Solving Problems

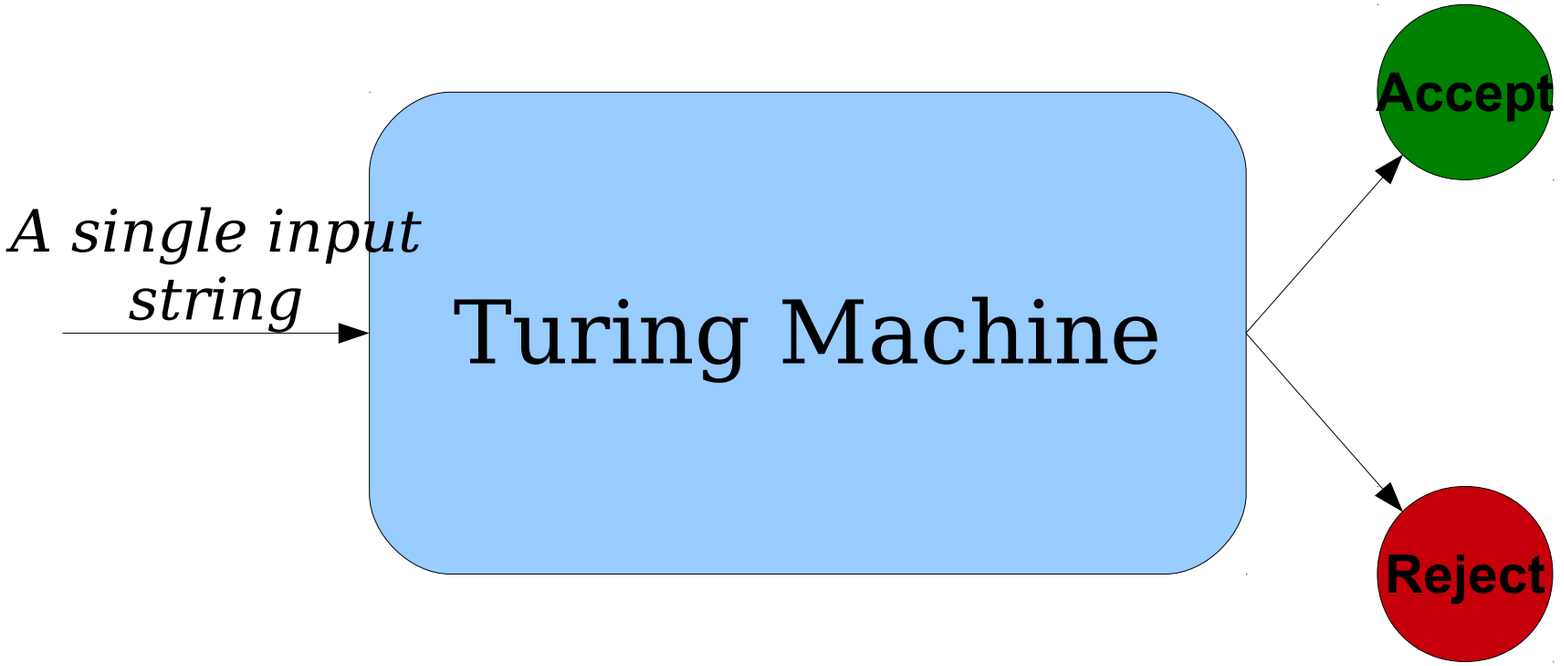



```
bool isPalindrome(string input) {

    // … do something …

}
```

A Model for Solving Problems



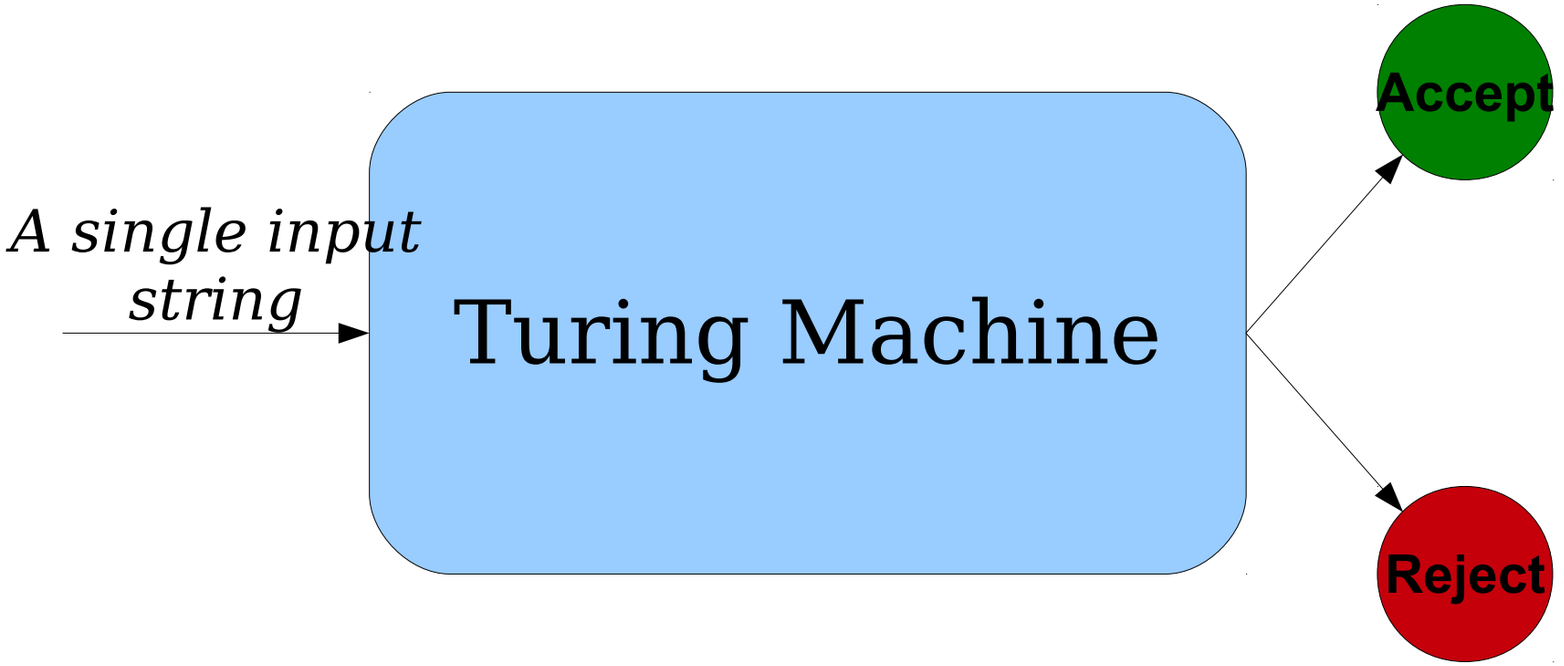

```
bool isLinkageGraph(Graph G) {

    // … 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.

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 ⟨G⟩ to mean "some encoding of a graph G" 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 person has figured out an encoding system for us and that we're using that encoding system.

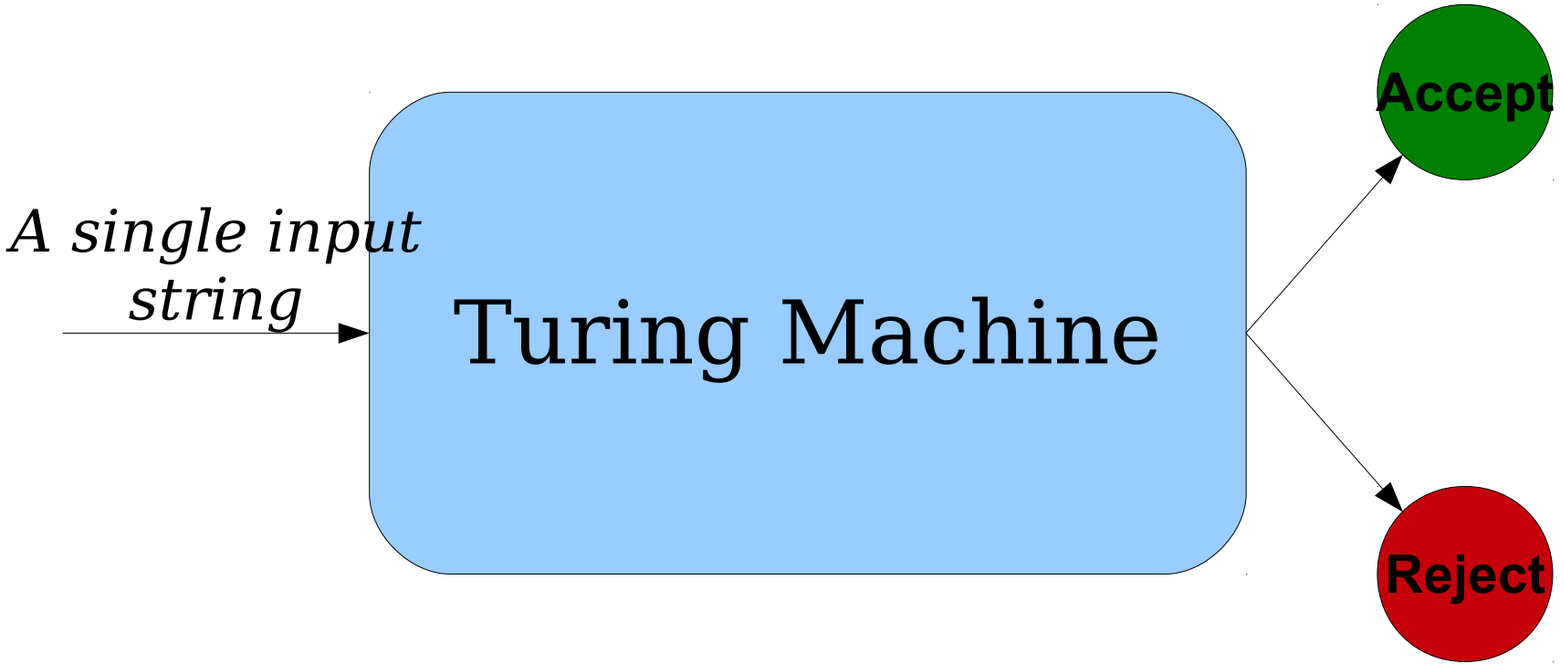
Object Encodings Caution

- ***The ⟨X⟩ notation isn't all-powerful magic.*** It must obey the rules of strings, which you recall are required to have finite length. Wrapping something that requires infinite length to describe in brackets is not allowed.
- ***Great intuition:*** If you can store an object as a file on disk, then you can encode it as a string.

Question: how many of these objects can *always* be encoded as a string?

- A DFA over the alphabet {**a**, **b**}.
- A regular expression.
- A subset of {**a**, **b**}*.
- A function f from $\{ k \in \mathbb{N} \mid k < n \}$ to itself, for some $n \in \mathbb{N}$.
- A graph whose nodes are the set $\{ k \in \mathbb{N} \mid k < n \}$, for some $n \in \mathbb{N}$.

A Model for Solving Problems




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

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

In summary: we define a “problem” as a language

- In other words, a set of strings.
- This may seem like an even bigger leap of faith than saying that our simple TM language is equivalent in expressive power to any possible programming language.
- But we can be creative about shoehorning problems into this form. Here’s how we could think of addition of natural numbers as a set of strings:
  - ADD = { s+t=u | s, t, and u are strings of digits and the sum is correct}

What problems can we solve with a computer?

Key Properties

- There are two key 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?

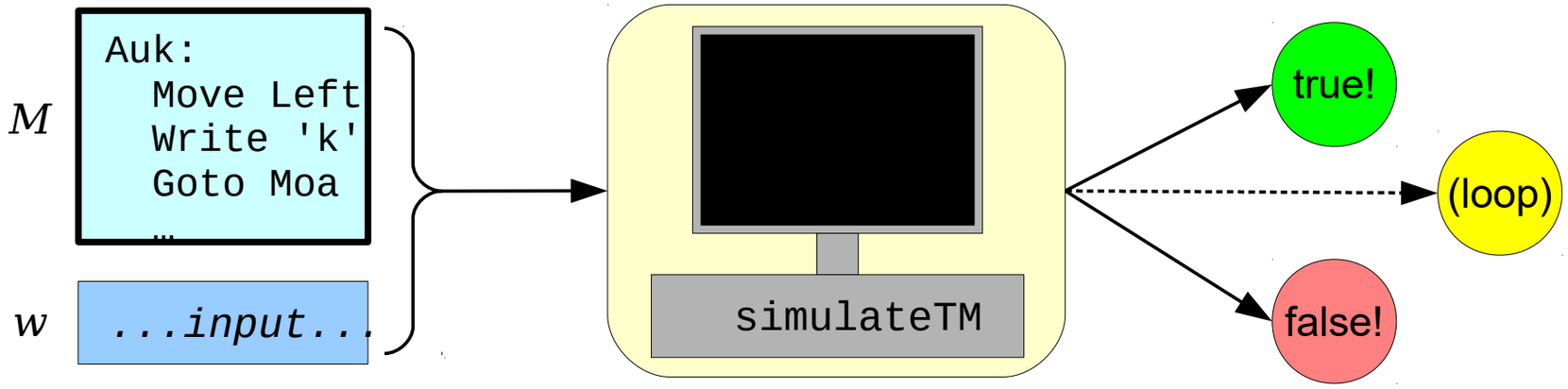
Can we make a “reprogrammable Turing machine?”

A TM Simulator

- It is possible to program a TM simulator on an unbounded-memory computer.
  - You've seen this in class, and you'll use one on PS8.
- We could imagine it as a method

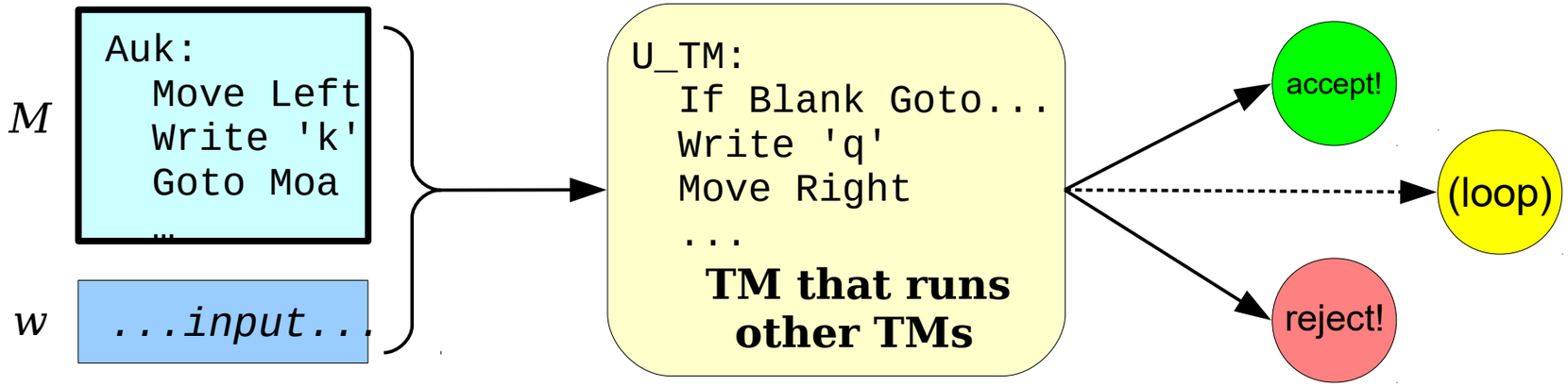
```
bool simulateTM(TM M, string w)
```

  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.
- This means that there must be some TM that has the behavior of this `simulateTM` method.
- What would that look like?



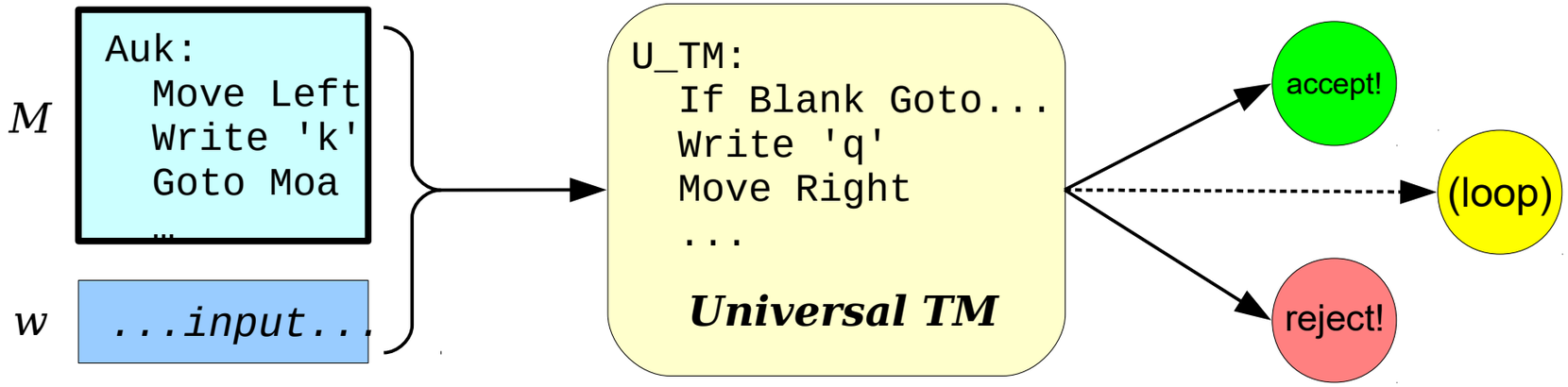
The Universal Turing Machine

- ***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 on $\langle M, w \rangle$

what

M does on 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

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

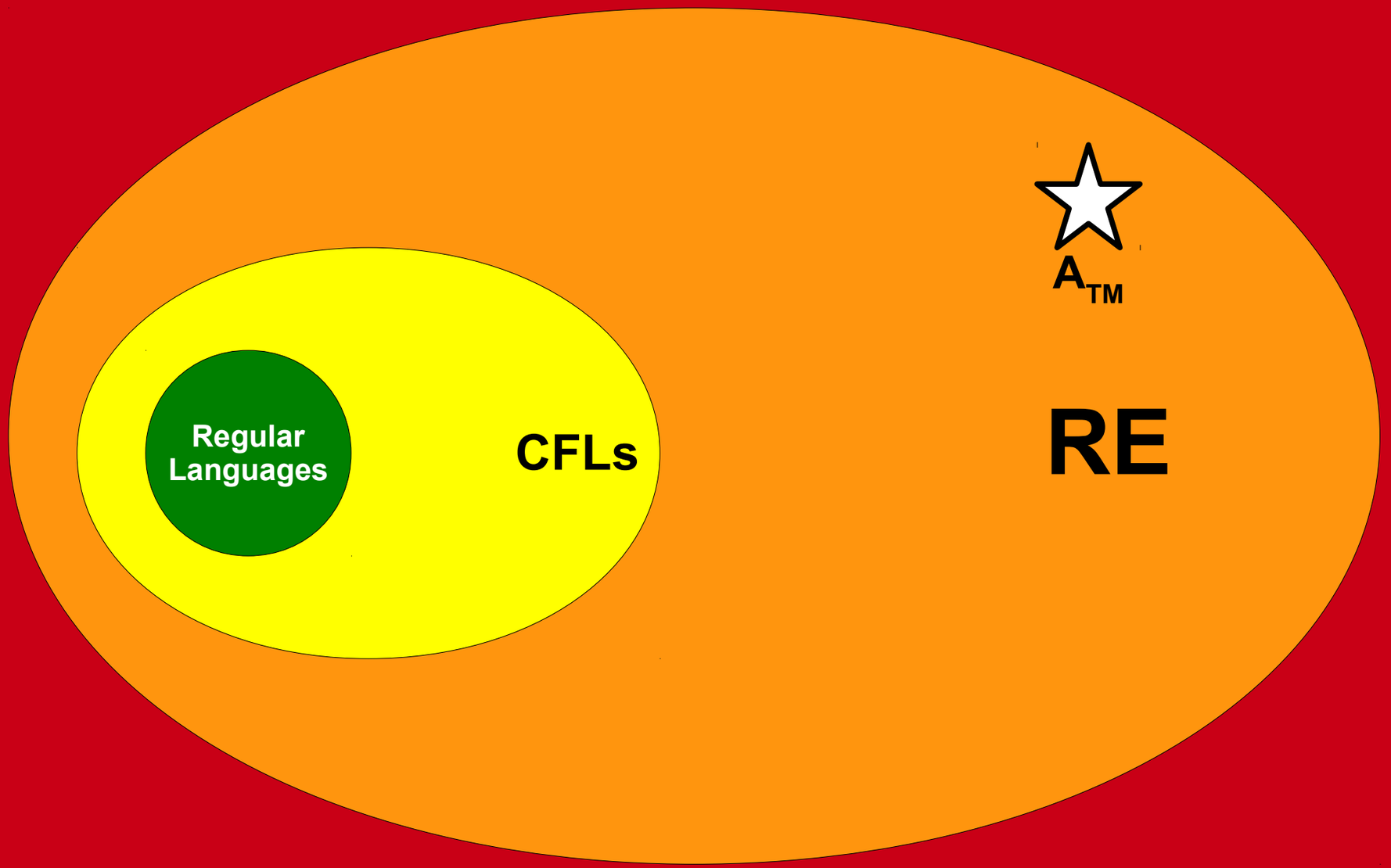
U_{TM} as a Recognizer

- 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 }

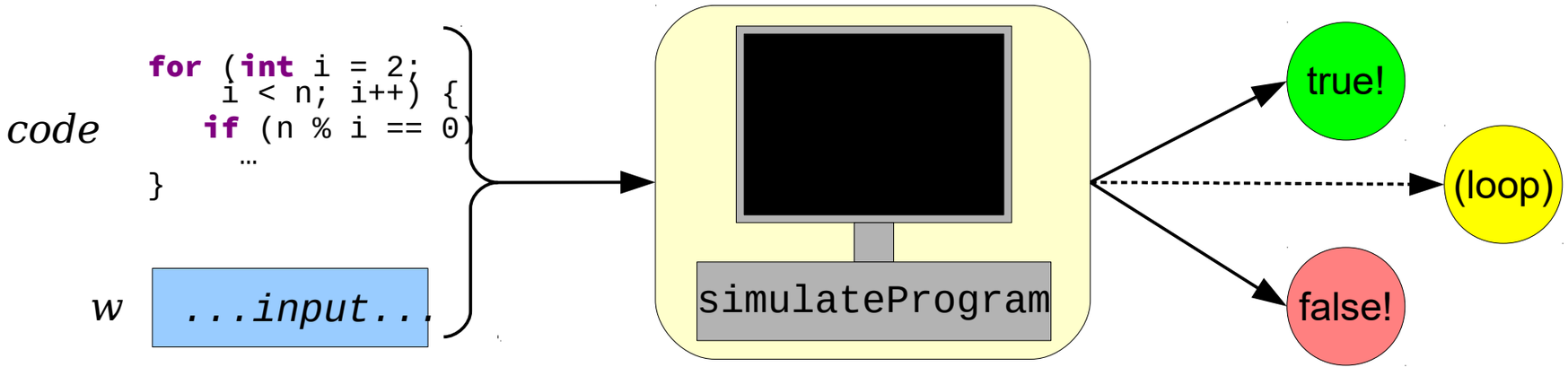

Regular Languages
CFLs
A_{TM}
RE
All Languages

Uh… so what?

Why Does This Matter?

The existence of a universal Turing machine has both theoretical and practical significance.

- 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.



Self-Reference

Self-Referential Programs

- If TMs can take other TMs as input, can 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 to a Turing machine in power possesses some mechanism for self-reference!
- Want to see how deep the rabbit hole goes? Take CS154!

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?