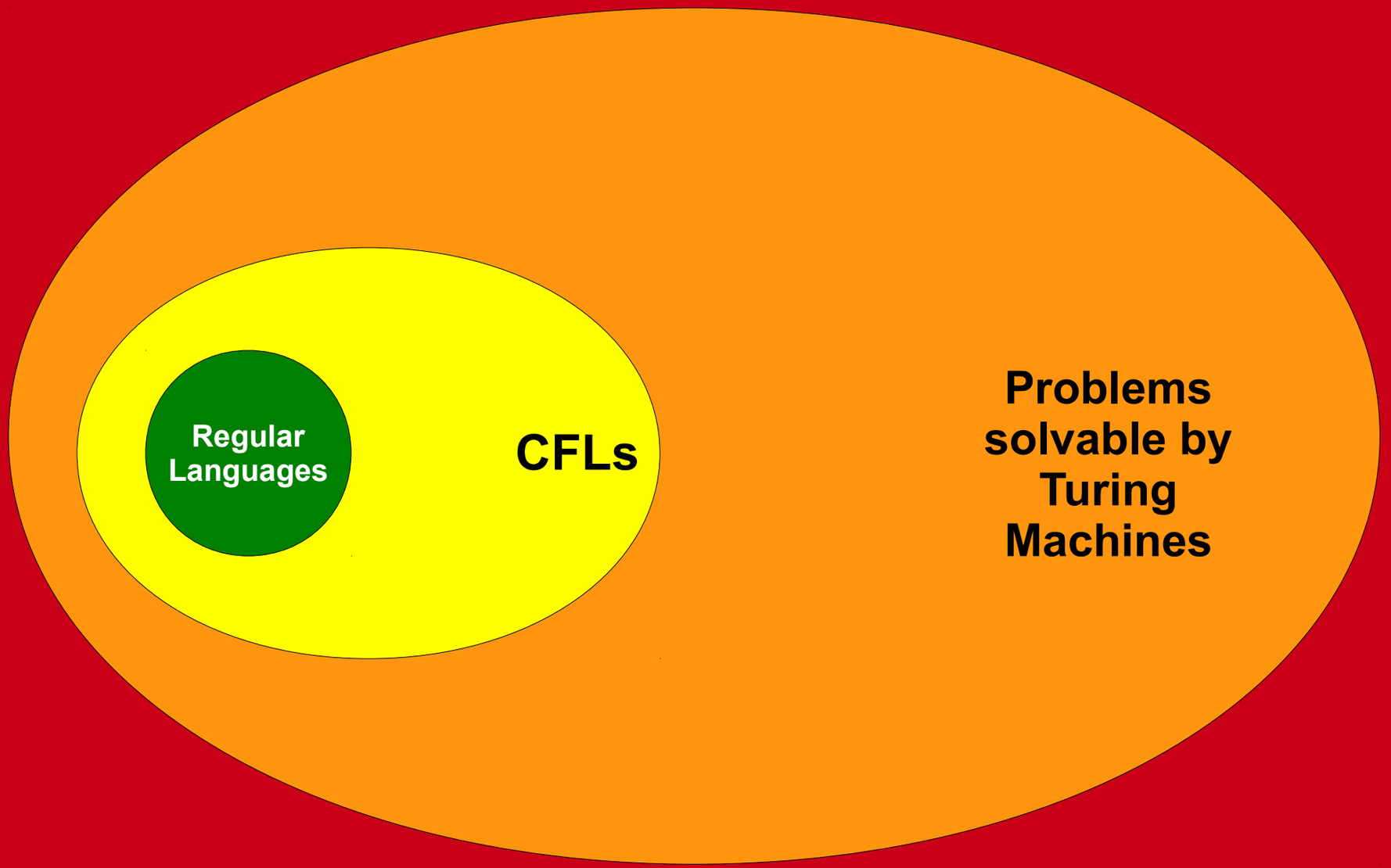
Turing Machines

Part Three

Recap from Last Time


Regular Languages
CFLs
Problems solvable by Turing Machines
All Languages

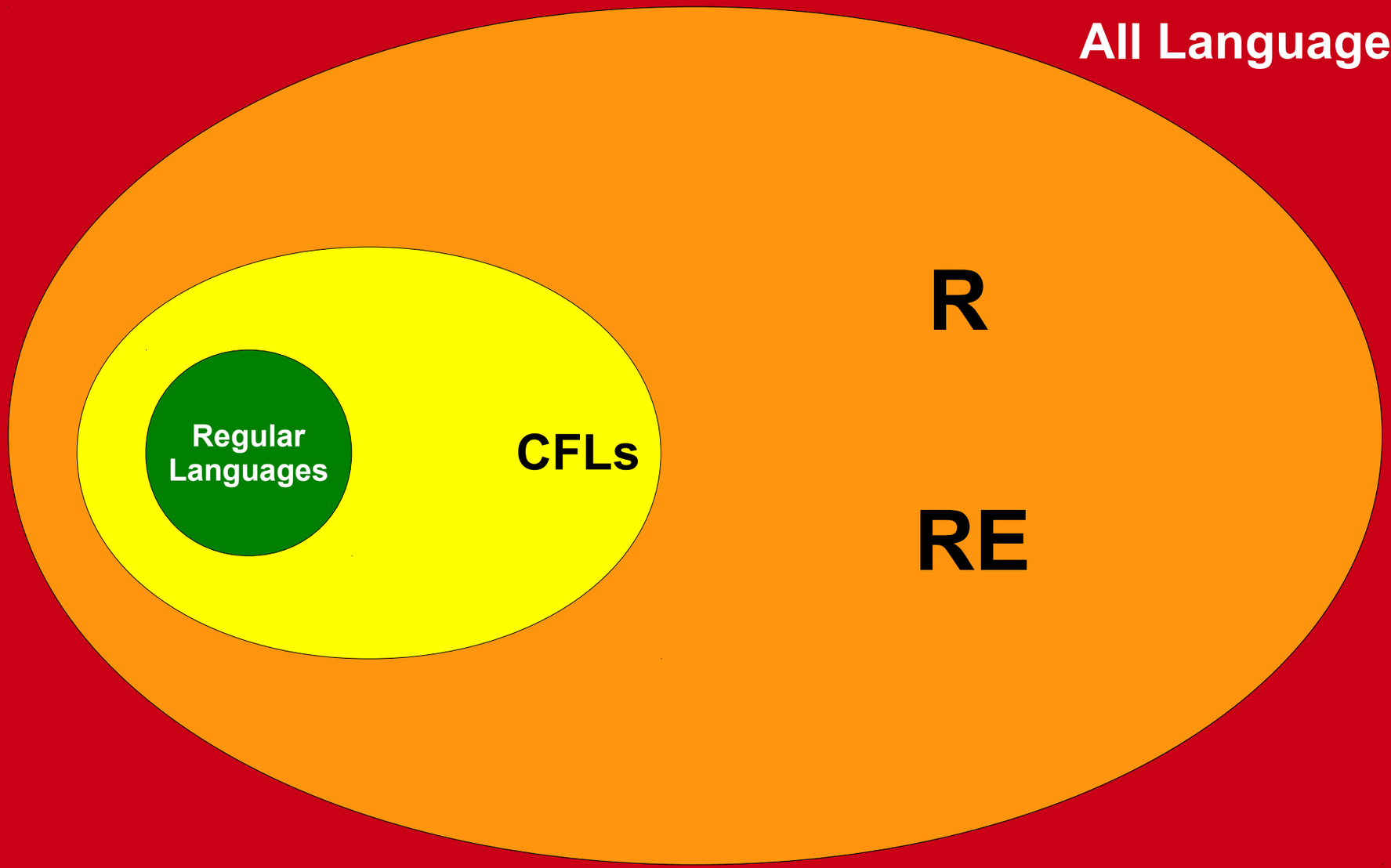
Recognizable Languages (**RE**)

- 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**.

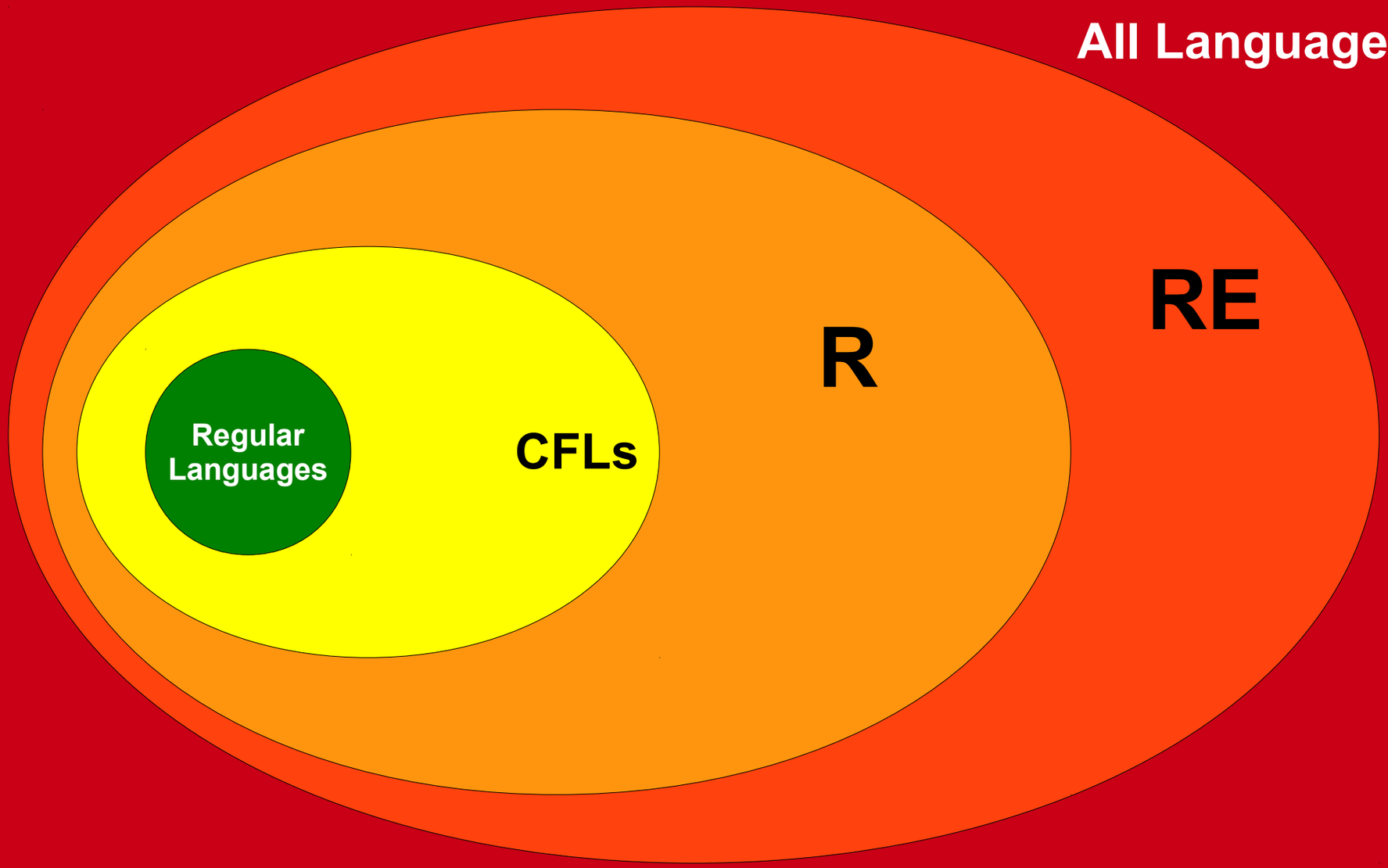
Decidable Languages (**R**)

- 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?



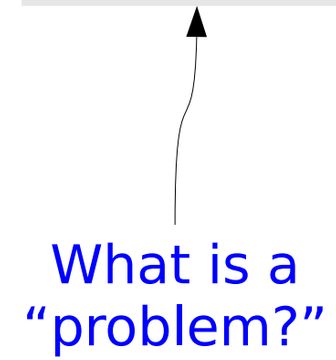
Which Picture is Correct?



New Stuff!

Strings, Languages, and Encodings

What problems can we solve with a computer?



What is a
“problem?”

Decision Problems

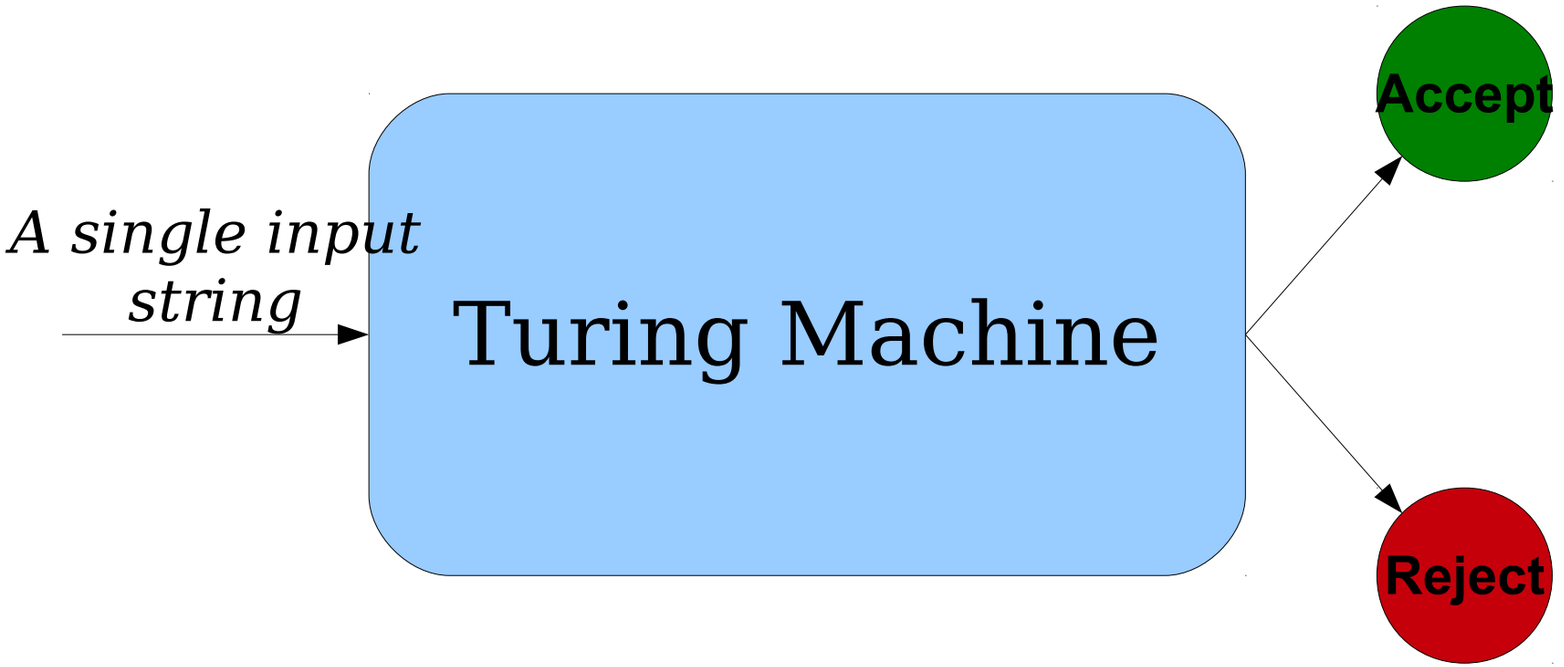
- A ***decision problem*** is a type of problem where the goal is to provide a yes or no answer.
- Example: Bin Packing

  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?

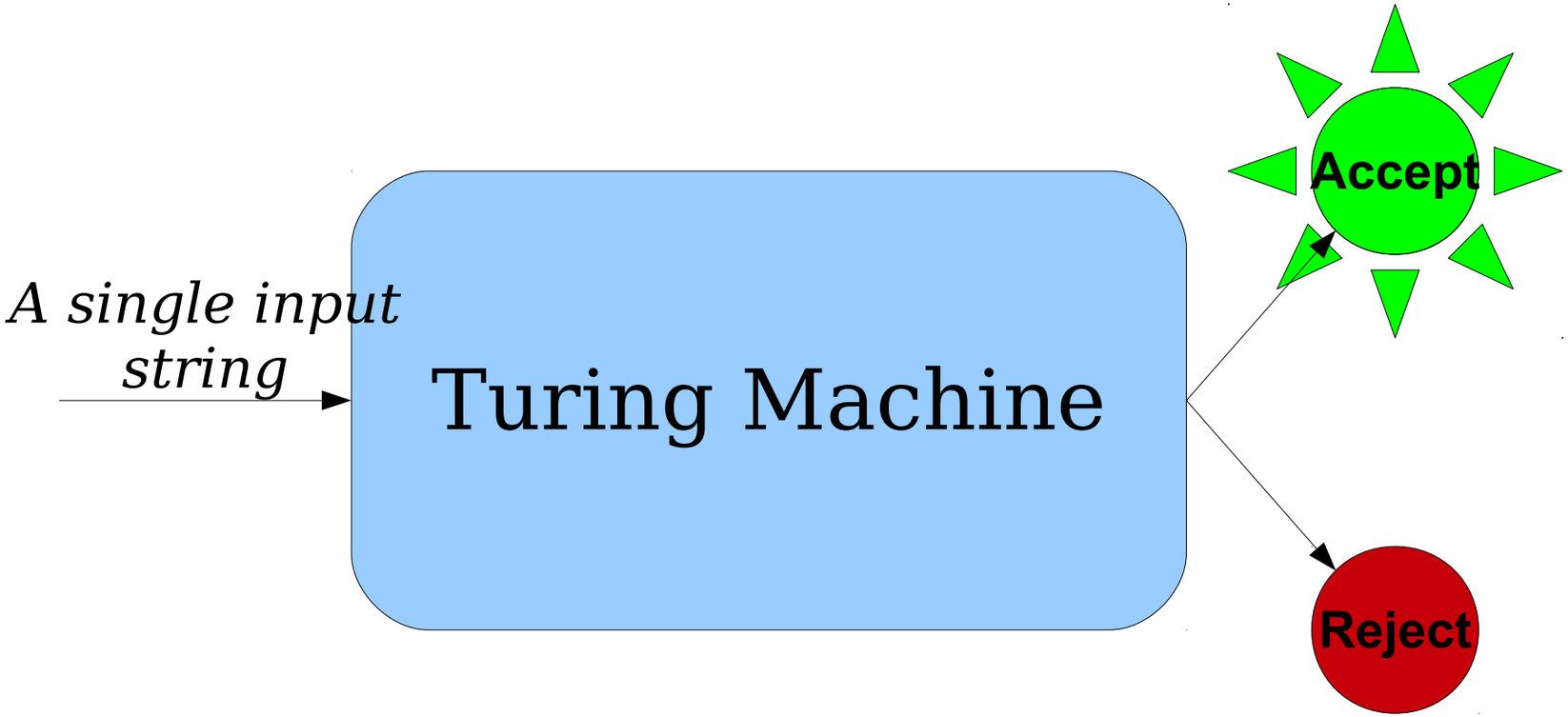
- Example: Dominating Set Problem

  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?

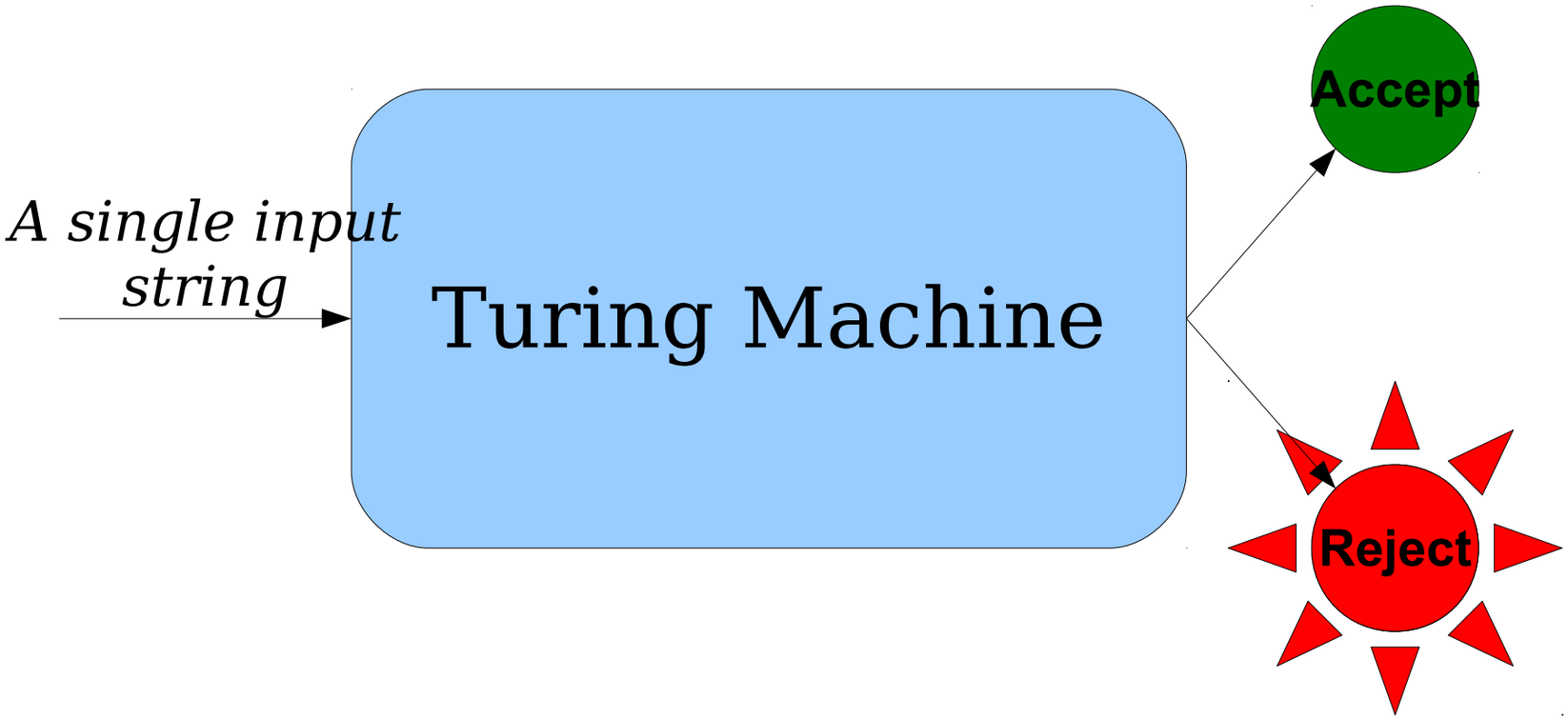
A Model for Solving Problems



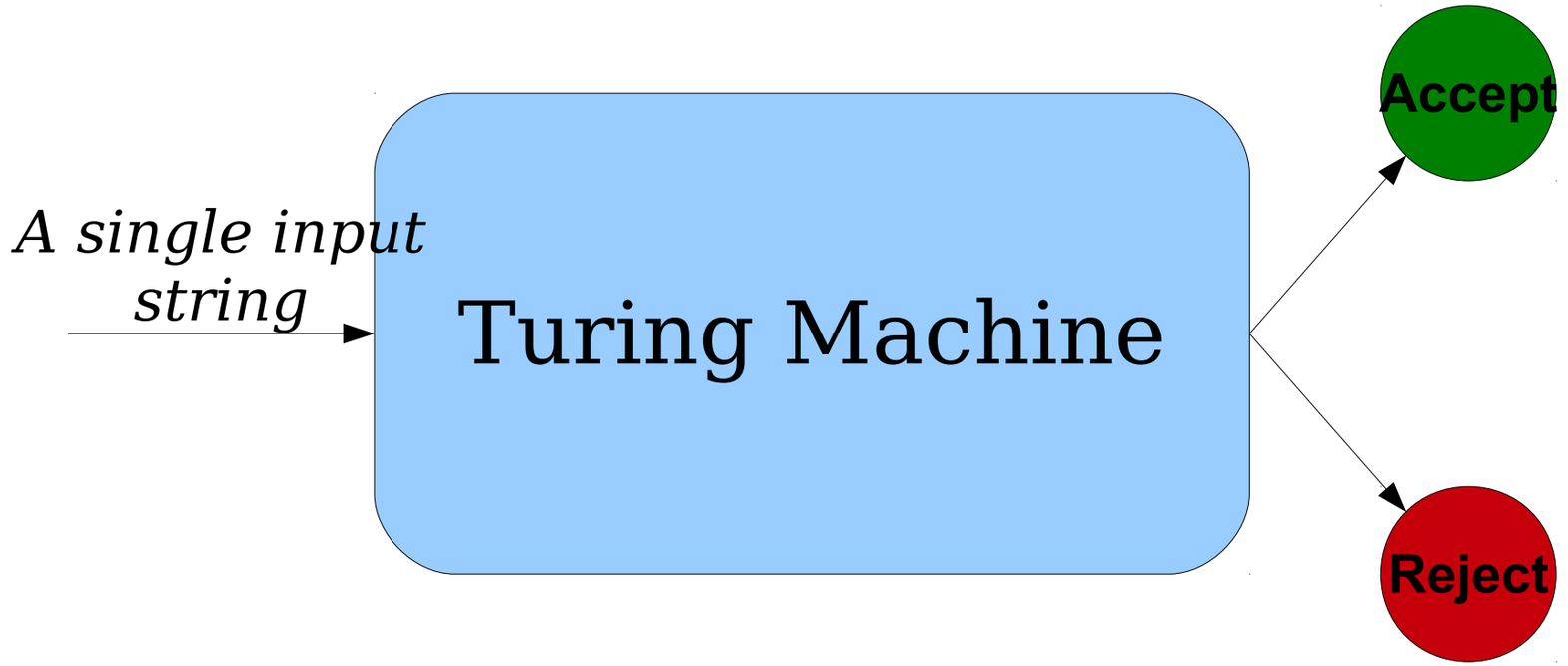
A Model for Solving Problems



A Model for Solving Problems



A Model for Solving Problems

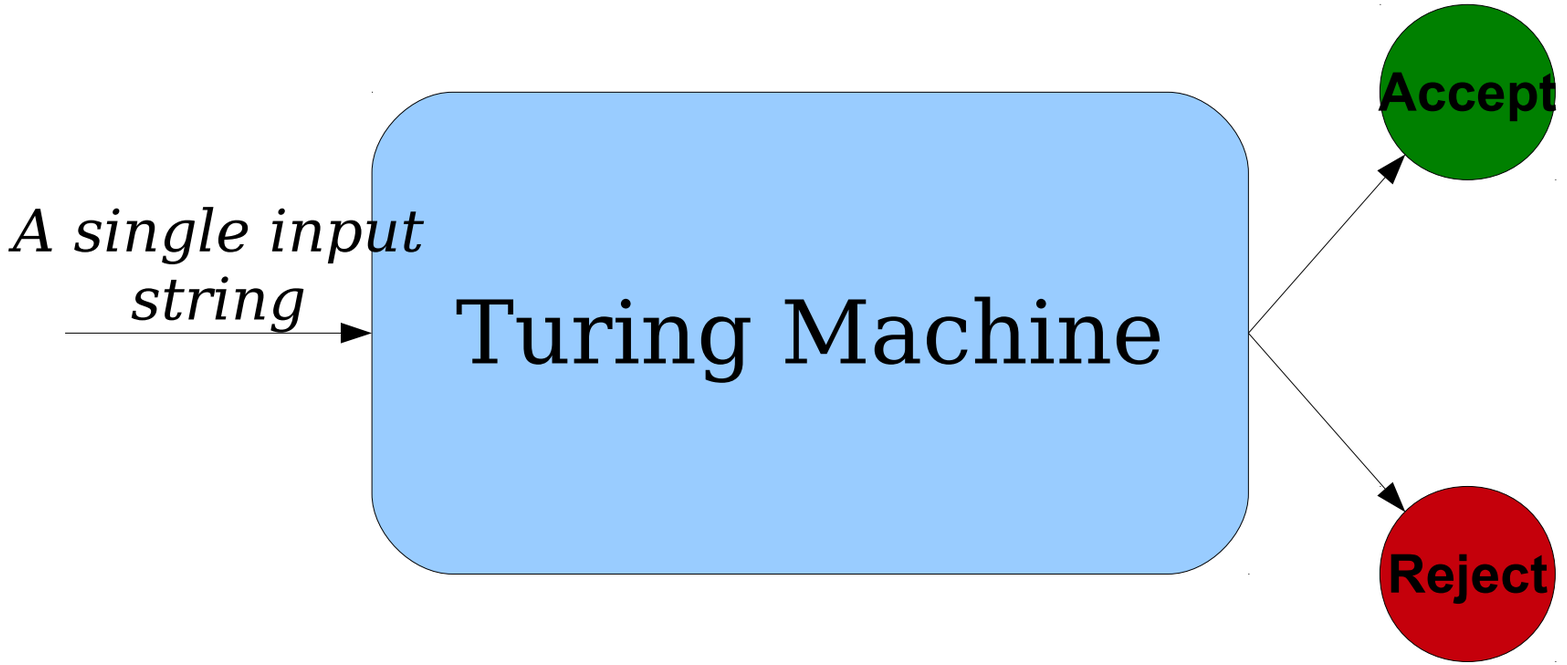



```
bool someFunctionName(string input) {

    // … do something …

}
```

A Model for Solving Problems

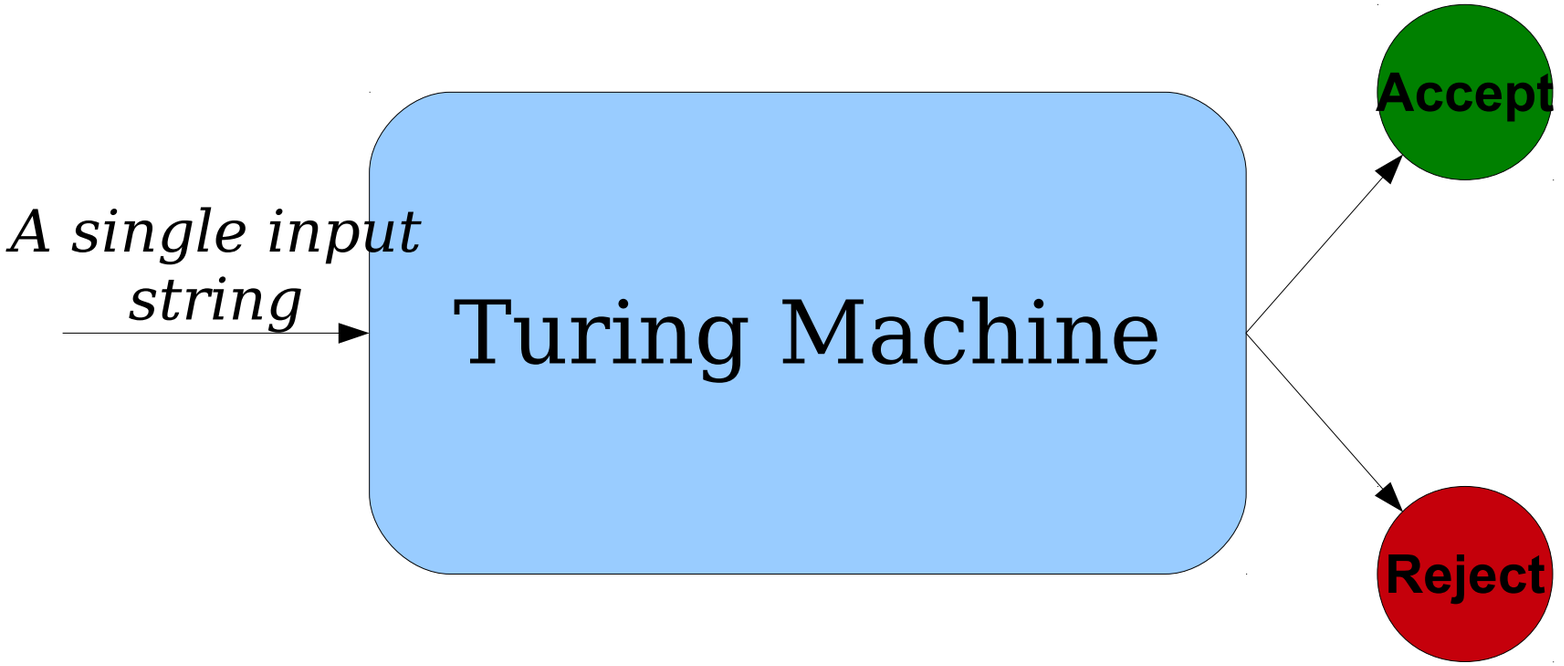



```
bool isAnBn(string input) {

    // … do something …

}
```

A Model for Solving Problems

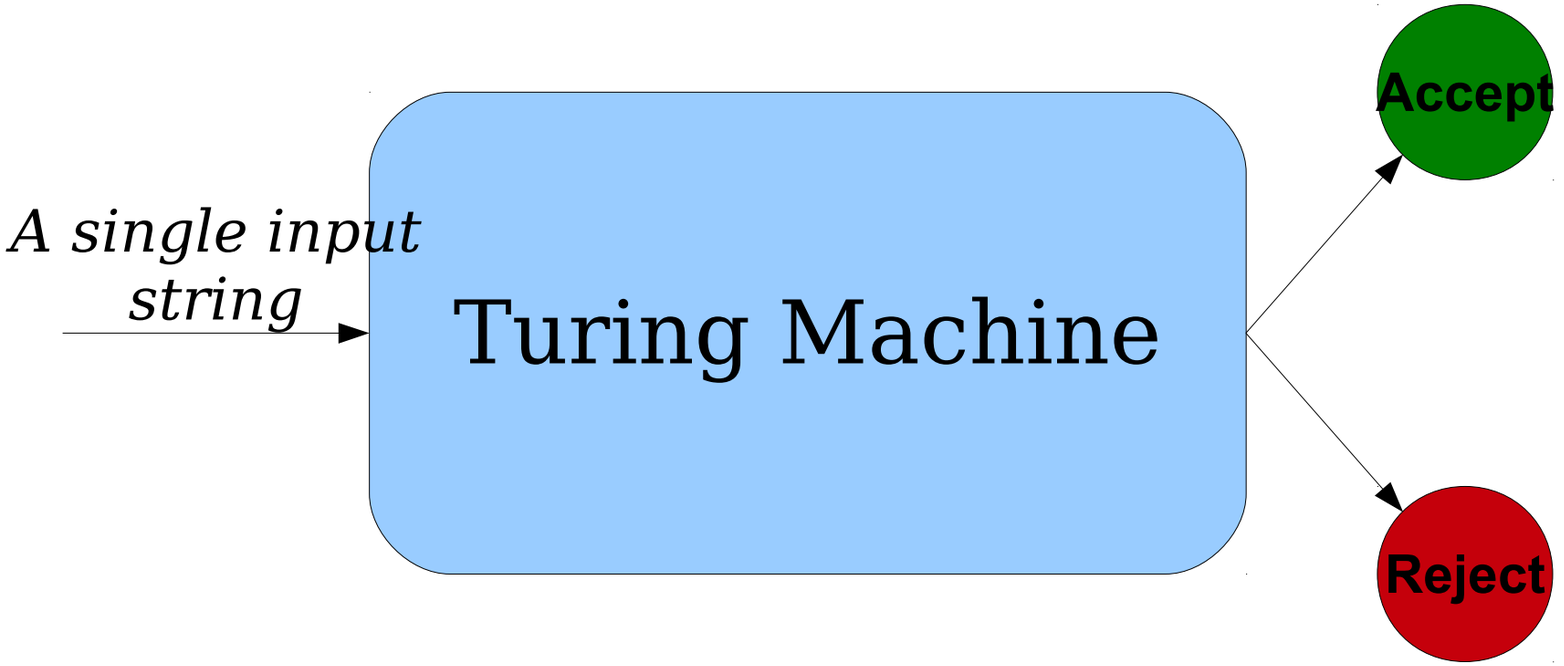



```
bool isPalindrome(string input) {

    // … do something …

}
```

A Model for Solving Problems



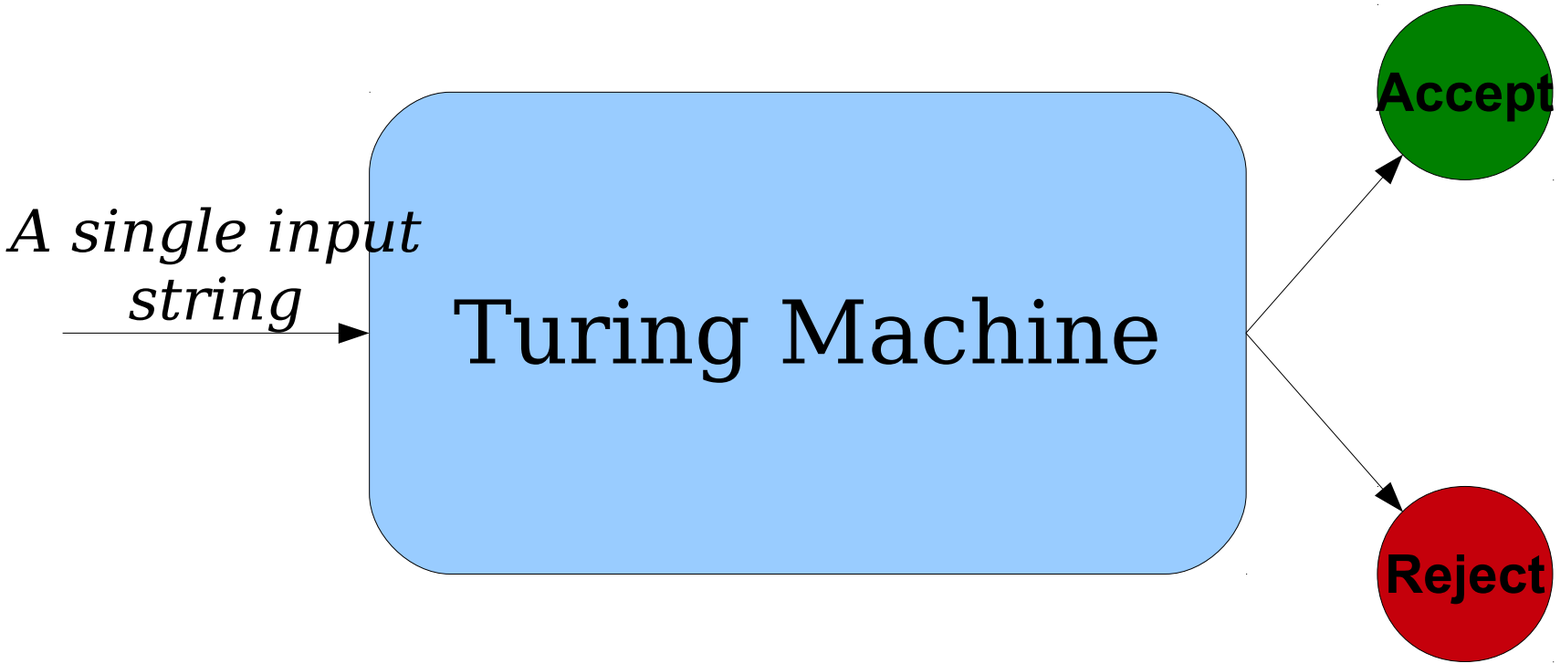

```
bool isLinkageGraph(Graph G) {

    // … 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.

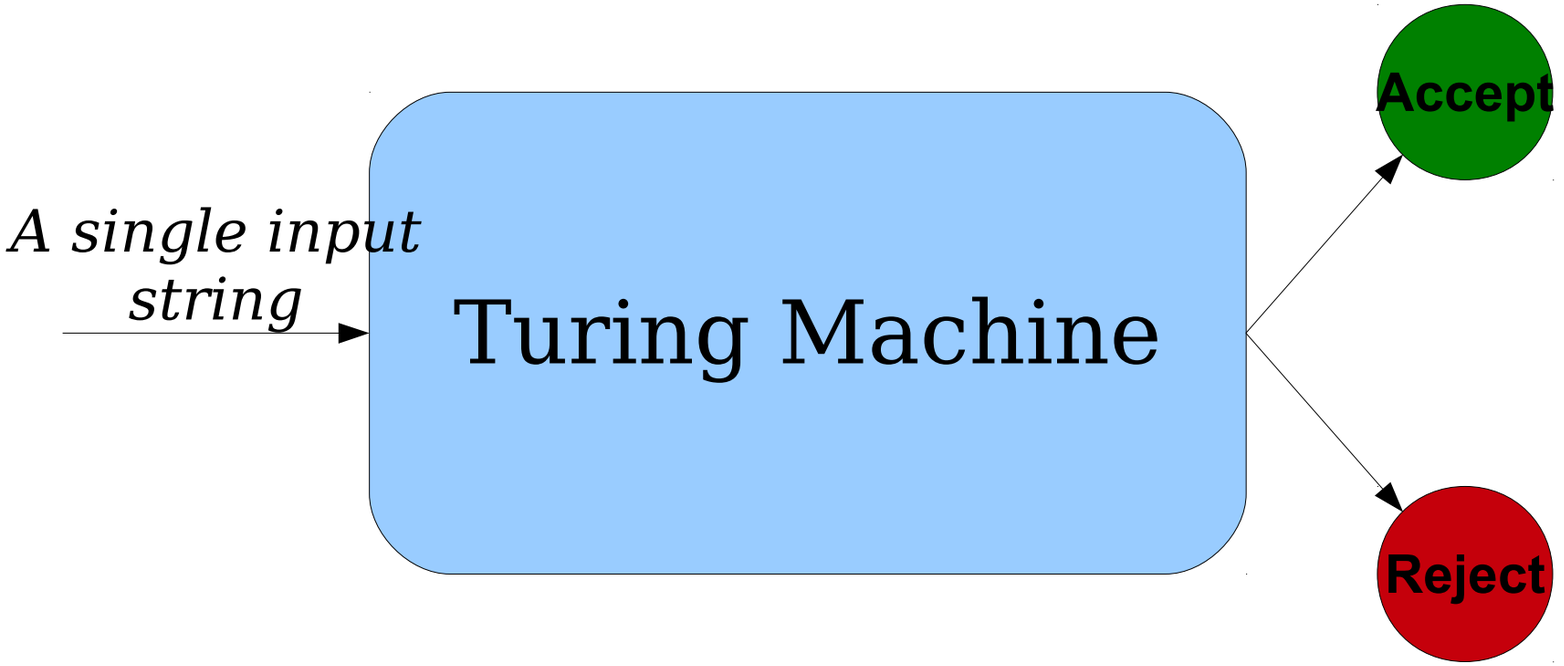
Object Encodings Caution

- ***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?

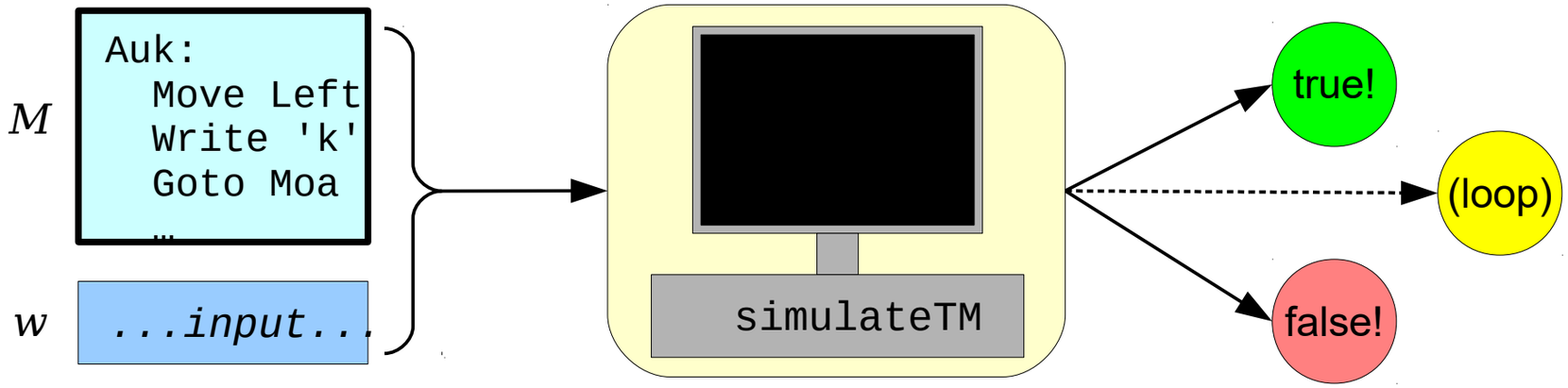
Can we make a “reprogrammable Turing machine?”

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

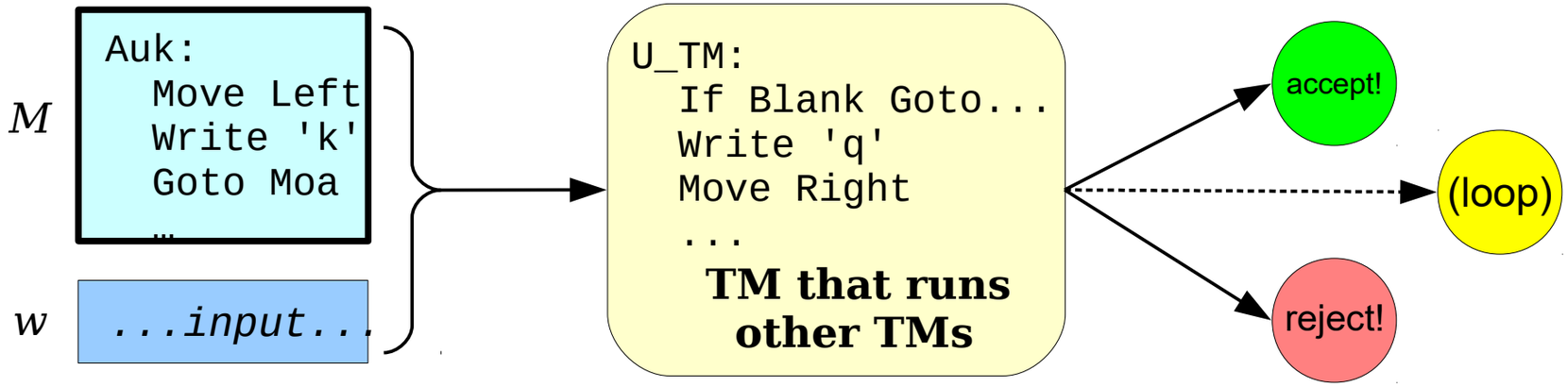
```
bool simulateTM(TM M, string w)
```

  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?



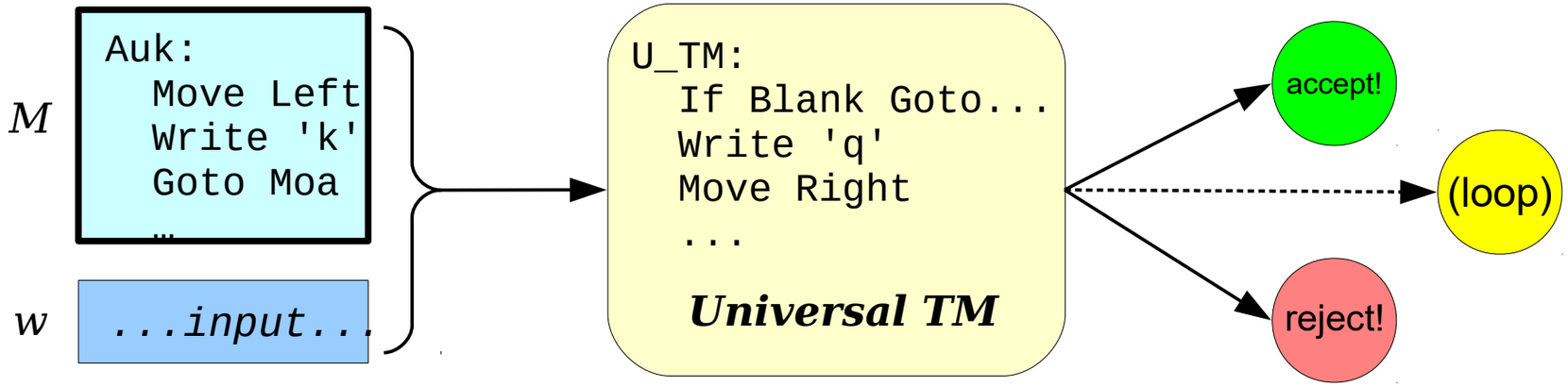
The Universal Turing Machine

- ***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

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

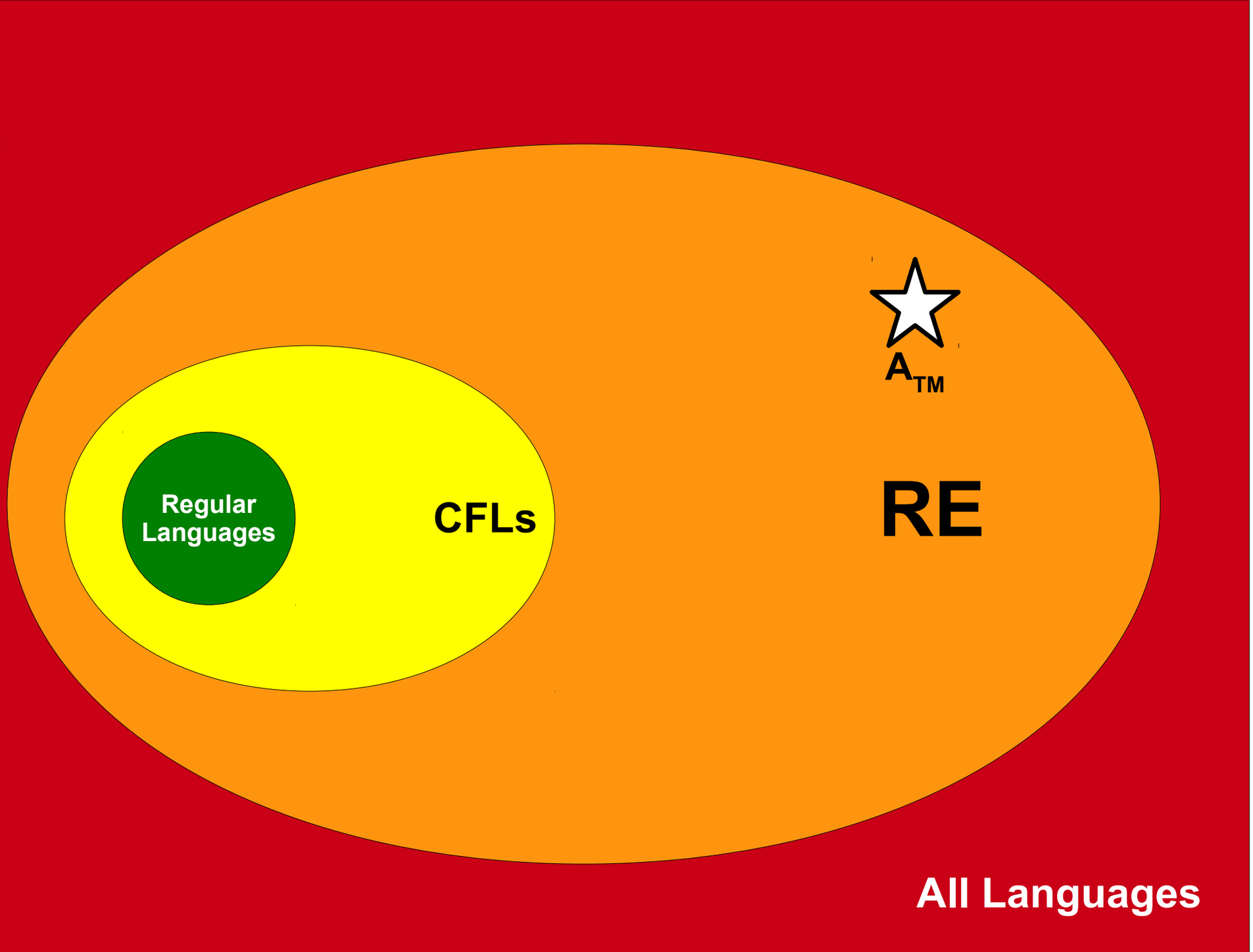
U_{TM} as a Recognizer

- 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 **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 }

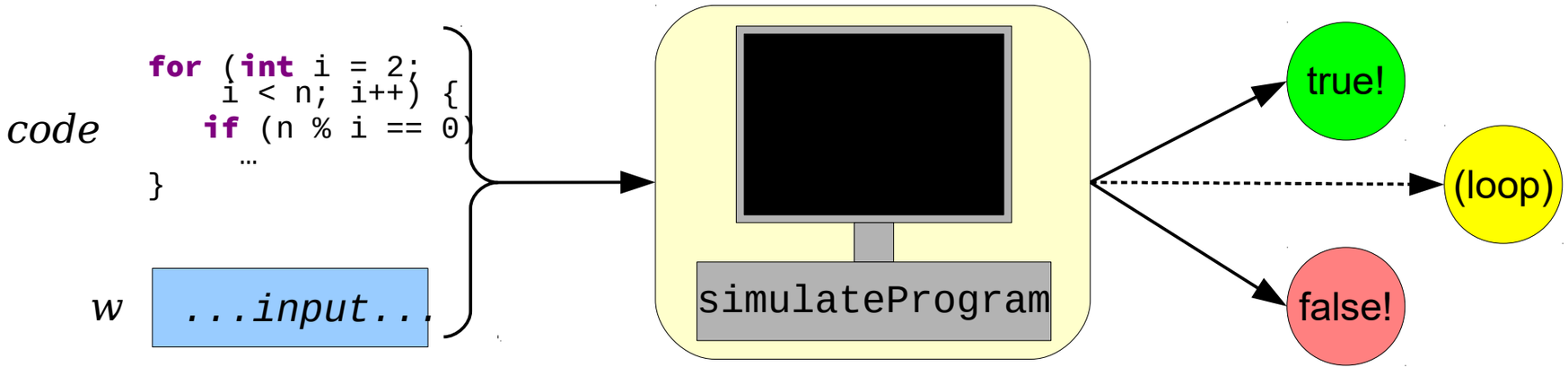

Regular Languages
CFLs
A_{TM}
RE
All Languages

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!

True or false:

```
"This string is 34 characters
          long."
```

True or false:

```
"This string is 34 characters long."
 12345678901234567890123456789012345
```

True or false:

"This sentence is written in blue."

True or false:

"This sentence is false."

Happy Story Time

In a certain isolated town, every house has a lawn and the city requires them all to be mowed. The town has only one gardener, who is also a resident of the town, and this gardener mows the lawns of residents iff they do not mow their own lawn.

Happy Story Time

In a certain isolated town, every house has a lawn and the city requires them all to be mowed. The town has only one gardener, who is also a resident of the town, and this gardener mows the lawns of residents iff they do not mow their own lawn.

True or false: The gardener mows their own lawn.


MY NOSE WILL
GROW NOW!

Self-Reference in Set Theory

- Now that we know that self-reference is dangerous (i.e., can lead to paradoxes) in propositions (e.g.,“This sentence is false”), we can look at it in other domains we’ve studied, like Set Theory.

Self-Reference in Set Theory

- Now that we know that self-reference is dangerous (i.e., can lead to paradoxes) in propositions (e.g.,“This sentence is false”), we can look at it in other domains we’ve studied, like Set Theory.
  - We know that sets can contain other sets.

Self-Reference in Set Theory

- Now that we know that self-reference is dangerous (i.e., can lead to paradoxes) in propositions (e.g.,“This sentence is false”), we can look at it in other domains we’ve studied, like Set Theory.
  - We know that sets can contain other sets.
  - We never said they can’t contain themselves (“The set of all sets” and “The set of all sets with infinite cardinality” would be examples of sets that would contain themselves.)

Self-Reference in Set Theory

- Now that we know that self-reference is dangerous (i.e., can lead to paradoxes) in propositions (e.g.,“This sentence is false”), we can look at it in other domains we’ve studied, like Set Theory.
  - We know that sets can contain other sets.
  - We never said they can’t contain themselves (“The set of all sets” and “The set of all sets with infinite cardinality” would be examples of sets that would contain themselves.)
  - Since we know that self-reference is dangerous, we might want to make a set called SAFE_LIST that is the set of all sets that do *not* contain themselves, since any set on that list is safe from paradoxes.

Self-Reference in Set Theory

- Now that we know that self-reference is dangerous (i.e., can lead to paradoxes) in propositions (e.g.,“This sentence is false”), we can look at it in other domains we’ve studied, like Set Theory.
  - We know that sets can contain other sets.
  - We never said they can’t contain themselves (“The set of all sets” and “The set of all sets with infinite cardinality” would be examples of sets that would contain themselves.)
  - Since we know that self-reference is dangerous, we might want to make a set called SAFE_LIST that is the set of all sets that do *not* contain themselves, since any set on that list is safe from paradoxes.

True or False: SAFE_LIST does *not* contain itself.

Self-Defeating Objects

Proofs by Contradiction in Number Theory

- One way to think about proofs by contradiction is that they lead to a kind of “impossible” situation that is similar to the paradoxes.
- One form of proof by contradiction makes use of what is called the “self-defeating object,” a thing that we assume for the sake of contradiction exists, but we will show can’t exist because its existence contradicts itself.
  - Similar to how SAFE_LIST, or the lawn mower, contradicted their own existences.

Proofs by Contradiction in Number Theory

- Here is a simple example:
  - **Thm.** There is no greatest integer.
  - **Proof, by contradiction.** Assume for the sake of contradiction that there is a greatest integer, call it g.
  - *[Now we will use g to write a mathematical expression that is a syntactically valid mathematical expression that should be fine to write,* ***if*** **g were real***.]*
  - Let $x = g + 1$.
  - We see that $x > g$.
  - But this is a contradiction, because g is the greatest integer.
  - So the assumption is false and the theorem is true. ■

Proofs by Contradiction in Number Theory

- Here is a si
  - **Thm.** The
  - **Proof, by contradiction.** Assume for the sake of contradiction that there is a greatest integer, call it g.
  - *[Now we will use g to write a mathematical expression that is a syntactically valid mathematical expression that should be fine to write,* ***if*** **g were real***.]*
  - **Let $x = g - 1$.**
  - We see that $x < g$.
  - There is no contradiction, so g actually is the greatest integer!
  - So the theorem is false, and there is a greatest integer. ■

What is the problem with this (modified) proof?

Proofs by Contradiction in Number Theory

- Here is a simple ex
  - **Thm.** There is no g
  - **Proof, by contradi**
    contradiction that t
  - *[Now we will use g*
    that is a syntactical
    that should be fine t
  - **Let $x = g + 1$.**
  - We see that $x > g$.
  - But this is a contradiction, because g is the greatest integer.
  - So the assumption is false and the theorem is true. ■

Observation: "Fixing" the proof by changing the math from addition to subtraction doesn't actually fix anything, because it wasn't the math that was the problem. It was g itself.

We chose addition on purpose because it's what we needed to do to **expose the existing problem with *g*.**

More Self-Reference!

(this time with Turing Machines)

A Decider for A_{TM}?

- ***Recall:*** A_{TM} is the language of the universal Turing machine.
- We know that $\langle M, w \rangle \in A_{TM}$ if and only if M accepts w.
- The universal Turing machine U_{TM} is a *recognizer* for A_{TM}. Could we build a *decider* for A_{TM}?

A Decider for A_{TM}?

- Suppose that $A_{TM} \in \mathbf{R}$.
- Formally, this means that there is a TM that decides A_{TM}.
- Intuitively, this means that there is a TM that takes as input ⟨M, w⟩, then
  - accepts if *M* accepts *w*, and
  - rejects if *M* does not accept *w*.
  - (i.e., infinite looping is not possible)

A Decider for A_{TM}: `willAccept`

- To make the previous discussion more concrete, let's talk about this hypothetical decider for A_{TM} as a computer program
- If A_{TM} is decidable, we could construct a function

```
bool willAccept(string program /*M*/,
                string input   /*w*/)
```

  that returns true if the program will accept the input and false otherwise (*never infinite looping*).
- **Hypothetically,** if `willAccept` existed, what could we do with it?

If A_{TM} is decidable, we could construct a function

```
bool willAccept(string program /*M*/,
                string input   /*w*/)
```

that returns true if the program will accept the input and false otherwise (*never infinite looping*).

How many of the following statements are true?

`willAccept("main() {   accept();   }",   "Emu")` returns true.
`willAccept("main() {   reject();   }",   "Yak")` returns false.
`willAccept("main() { while (true) {} }", "Cow")` loops forever.
`willAccept("main() { while (true) {} }", "Cow")` returns true.
`willAccept("main() { while (true) {} }", "Cow")` returns false.

What does this program do?

```
bool mystery(string input) {
   string me = mySource();
   if (willAccept(me, input)) {
      return false;
   } else {
      return true;
   }
}
```

If A_{TM} is decidable, we could construct a function

```
bool willAccept(string program /*M*/,
                string input   /*w*/)
```

that returns true if the program will accept the input and false otherwise (*never infinite looping*).

What does this program do?

```
bool mystery(string input) {
   string me = mySource();
   if (willAccept(me, input)) {
      return false;
   } else {
      return true;
   }
}
```

How many of

This prog
This prog
This prog

Try running this program on any input.
What happens if

... this program accepts its input?
It rejects the input!

... this program doesn't accept its input?
It accepts the input!

If A_{TM} is decidable, we coul

```
bool willAccept(st
                st
```

that returns true if the program will accept the input and false otherwise (*never infinite looping*).

What does this program do?

```
bool mystery(string input) {
   string me = mySource();
   if (willAccept(me, input)) {
      return false;
   } else {
      return true;
   }
}
```

Try running this program on any input.
What happens if

... this program accepts its input?
It rejects the input!

... this program doesn't accept its input?
It accepts the input!

If A_{TM} is decidable, we could construct a function

```
bool willAccept(string program /*M*/,
                string input   /*w*/)
```

that returns true if the program will accept the input and false otherwise (*never infinite looping*).

Knowing the Future

- This TM is analogous to a classical philosophical/logical paradox:

If you know what you are fated to do, can you avoid your fate?

- If A_{TM} is decidable, we can construct a TM that determines what it's going to do in the future (whether it will accept its input), then actively chooses to do the opposite.
- This leads to an impossible situation with only one resolution: **A_{TM} *must not be decidable!***

Next: writing this up as a proof

Theorem: $A_{TM} \notin \mathbf{R}$. *(Recall:* **R** *is the name of the set of decidable languages)*

Theorem: $A_{TM} \notin \mathbf{R}$. *(Recall:* **R** *is the name of the set of decidable languages)*

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 some decider D for A_{TM}, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

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

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. Then there is some decider D for A_{TM}, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program P:

```
bool mystery(string input) {
    string me = mySource();

    if (willAccept(me, input)) return false;
    else return true;
}
```

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

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. Then there is some decider D for A_{TM}, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program P:

```
bool mystery(string input) {
    string me = mySource();

    if (willAccept(me, input)) return false;
    else return true;
}
```

Pick an arbitrary string w and trace through the execution of program P on input w.

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

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. Then there is some decider D for A_{TM}, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program P:

```
bool mystery(string input) {
    string me = mySource();

    if (willAccept(me, input)) return false;
    else return true;
}
```

Pick an arbitrary string w and trace through the execution of program P on input w. If `willAccept(me, input)` returns true, then P must accept its input w.

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

Proof: By contradiction; assume that $A_{TM} \in \mathbf{R}$. Then there is some decider D for A_{TM}, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program P:

```
bool mystery(string input) {
    string me = mySource();

    if (willAccept(me, input)) return false;
    else return true;
}
```

Pick an arbitrary string w and trace through the execution of program P on input w. If `willAccept(me, input)` returns true, then P must accept its input w. However, in this case P proceeds to reject its input w. Otherwise, if `willAccept(me, input)` returns false, then P must not accept its input w. However, in this case P proceeds to accept its input w.

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 some decider D for A_{TM}, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

Given this, we could then construct this program P:

```
bool mystery(string input) {
    string me = mySource();

    if (willAccept(me, input)) return false;
    else return true;
}
```

Pick an arbitrary string w and trace through the execution of program P on input w. If `willAccept(me, input)` returns true, then P must accept its input w. However, in this case P proceeds to reject its input w. Otherwise, if `willAccept(me, input)` returns false, then P must not accept its input w. However, in this case P proceeds to accept its input w.

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 some decider D for A_{TM}, which we can represent in software as a method `willAccept` that takes as input the source code of a program and an input, then returns true if the program accepts the input and false otherwise.

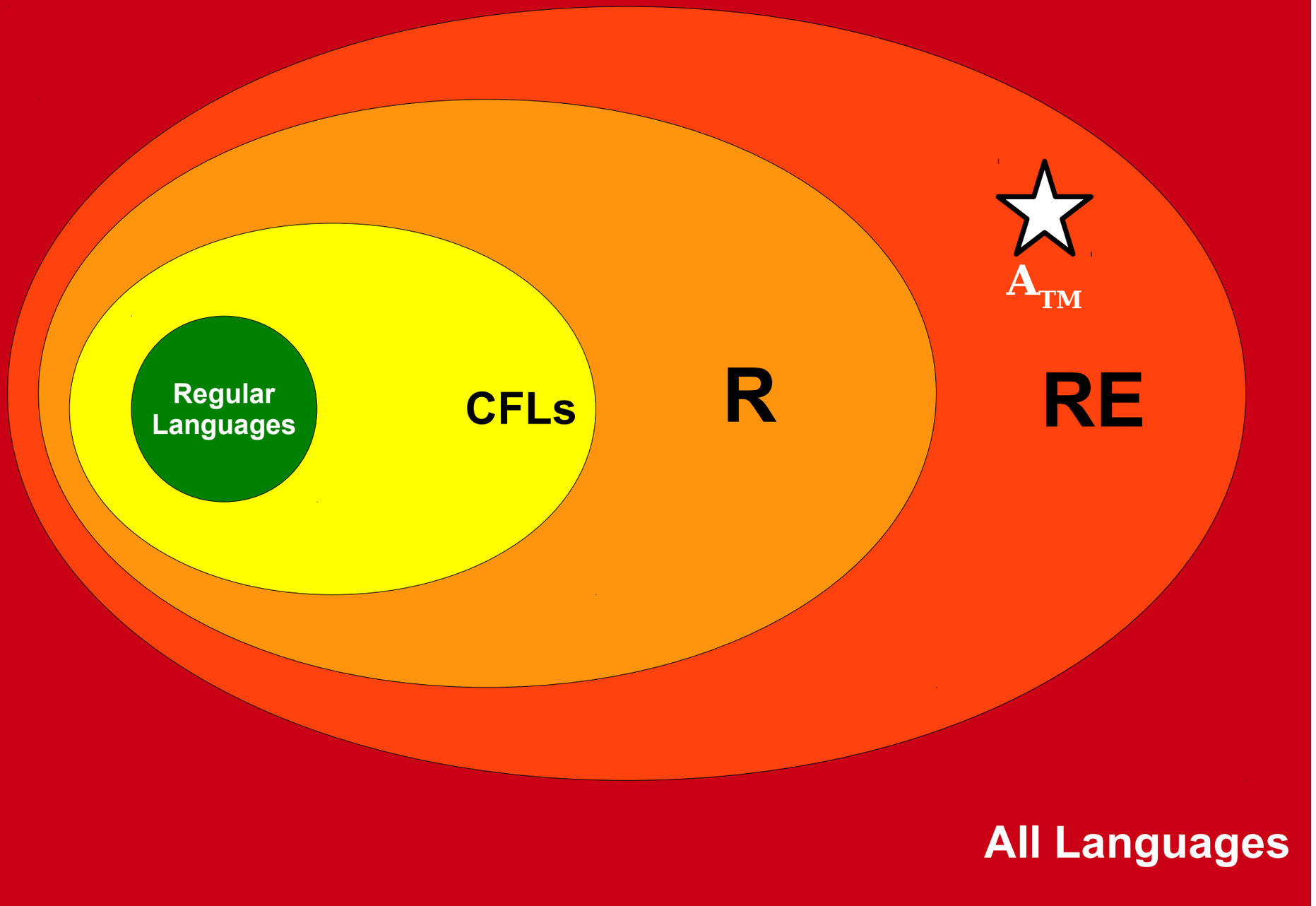
Given this, we could then construct this program P:

```
bool mystery(string input) {
    string me = mySource();

    if (willAccept(me, input)) return false;
    else return true;
}
```

Pick an arbitrary string w and trace through the execution of program P on input w. If `willAccept(me, input)` returns true, then P must accept its input w. However, in this case P proceeds to reject its input w. Otherwise, if `willAccept(me, input)` returns false, then P must not accept its input w. However, in this case P proceeds to accept its input w.

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 two statements really mean? As in, why should you care?

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

- The proof we've done says that

  There is no possible way to design an algorithm that will determine whether a program will accept an input.

- Notice that our proof just assumed there was some decider for A_{TM} and didn't assume anything about how that decider worked. In other words, no matter how you try to implement a decider for A_{TM}, you can never succeed!

R ≠ RE

- Because **R ≠ RE**, there are some problems where “yes” answers can be checked, but there is no algorithm for deciding what the answer is.
- ***In some sense, it is fundamentally harder to solve a problem than it is to check an answer.***

More Undecidability Results

The Halting Problem

- The most famous undecidable problem is the ***halting problem***, which asks:

  Given a TM *M* and a string *w*,
  will *M* halt* when run on *w*?

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

  ***HALT* = { ⟨*M*, *w*⟩ | *M* is a TM that halts on *w* }**

- How hard is this problem to solve?

* i.e., accept or reject, as opposed to infinite loop

$HALT \in \mathbf{RE}$

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

```
bool checkHalt(TM M, string w) {
   feed w into M;
   while (true) {
      if (M is in an accepting state) accept();
      else if (M is in a rejecting state) accept();
      else simulate one more step of M running on w;
   }
}
```

HALT ∉ **R**

- ***Claim:*** *HALT* ∉ **R**.
- If *HALT* is decidable, we could write some function

```
bool willHalt(string program,
              string input)
```

  that accepts as input a program and a string input, then reports whether the program will halt when run on the given input.
- Then, we could do this...

What does this program do?

```
bool mystery(string input) {
   string me = mySource();
   if (willHalt(me, input)) { //decider
      while (true) {
         // loop infinitely
      }
   } else {
      accept();
   }
}
```

What does this program do?

```
bool mystery(string input) {
   string me = mySource();
   if (willHalt(me, input)) { //decider
      while (true) {
         // loop infinitely
      }
   } else {
      accept();
   }
}
```

Imagine running this program on some input. What happens if...

... this program halts on that input?
It loops on the input!

... this program loops on this input?
It halts on the input!

Theorem: *HALT* $\notin$ **R**.

Proof: By contradiction; assume that *HALT* $\in$ **R**. Then there's a decider *D* for *HALT*, which we can represent in software as a method `willHalt` that takes as input the source code of a program and an input, then returns true if the program halts on the input and false otherwise.

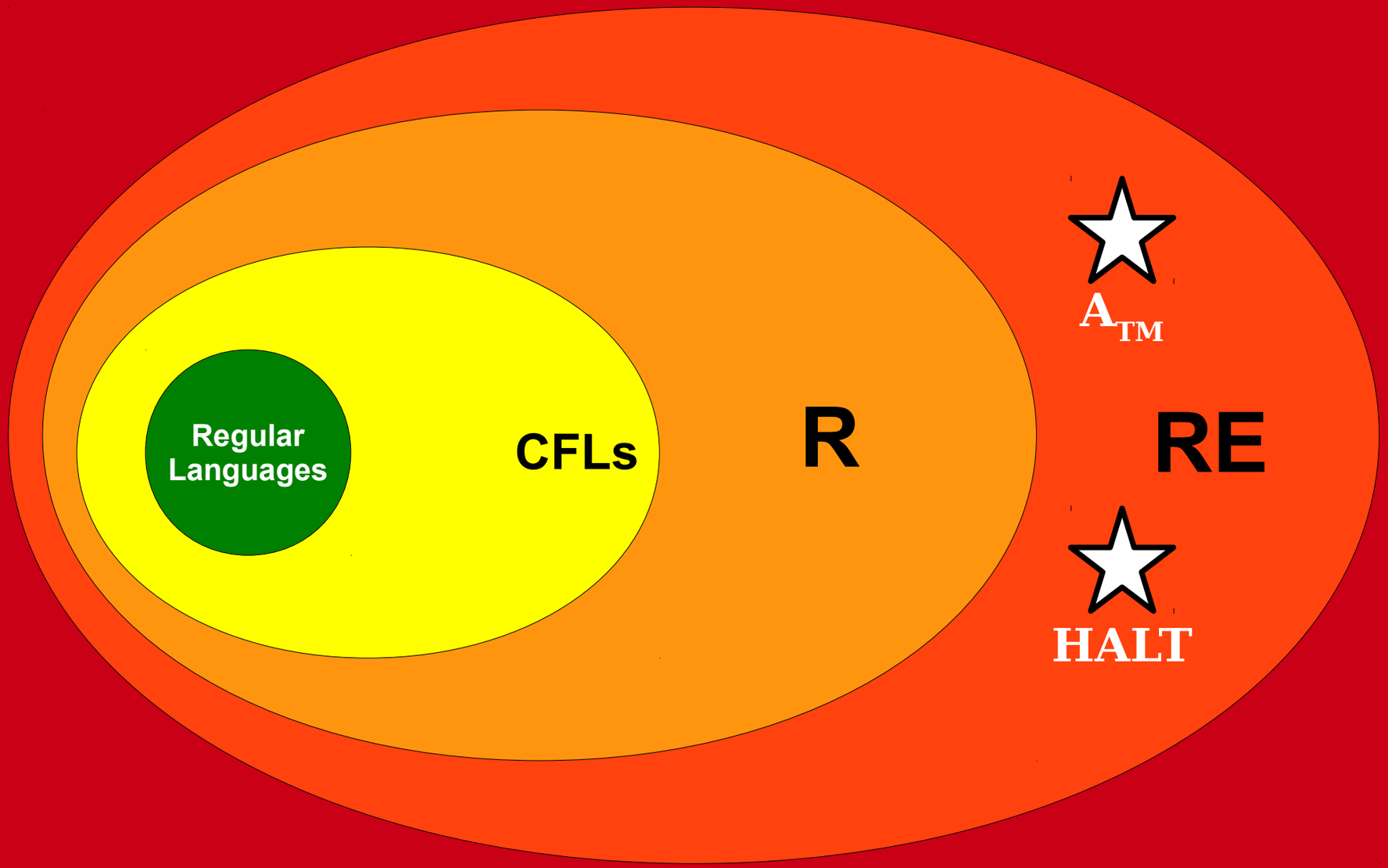
Given this, we could then construct this program *P*:

```
bool mystery(string input) {

   string me = mySource();
   if (willHalt(me, input)) { //decider
       while (true) {
           // loop infinitely
       }
   } else {
       accept();
   }
}
```

Choose any string *w* and trace through the execution of program *P* on input *w*, focusing on the answer given back by the `willHalt` method. If `willHalt(me, input)` returns true, then *P* must halt on its input *w*. However, in this case *P* proceeds to loop infinitely on *w*. Otherwise, if `willHalt(me, input)` returns false, then *P* must not halt its input *w*. However, in this case *P* proceeds to accept its input *w*.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, *HALT* $\notin$ **R**. ■


Regular Languages
CFLs
R
RE
A_{TM}
HALT
All Languages