Turing Machines

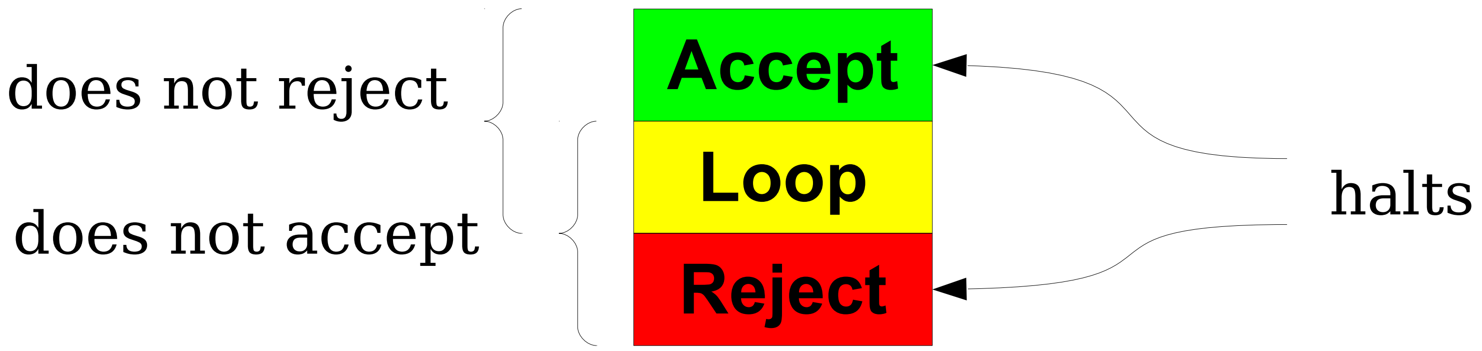
Part Three

What problems can we solve with a computer?

What kind of computer?

Very Important Terminology

- Let M be a Turing machine.
- M ***accepts*** a string w if it enters an accept state when run on w.
- M ***rejects*** a string w if it enters a reject state when run on w.
- M ***loops infinitely*** (or just ***loops***) on a string w if when run on w it enters neither an accept nor a reject state. (such a w is **not** in the language of this TM)



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**.
    -
    -
    -

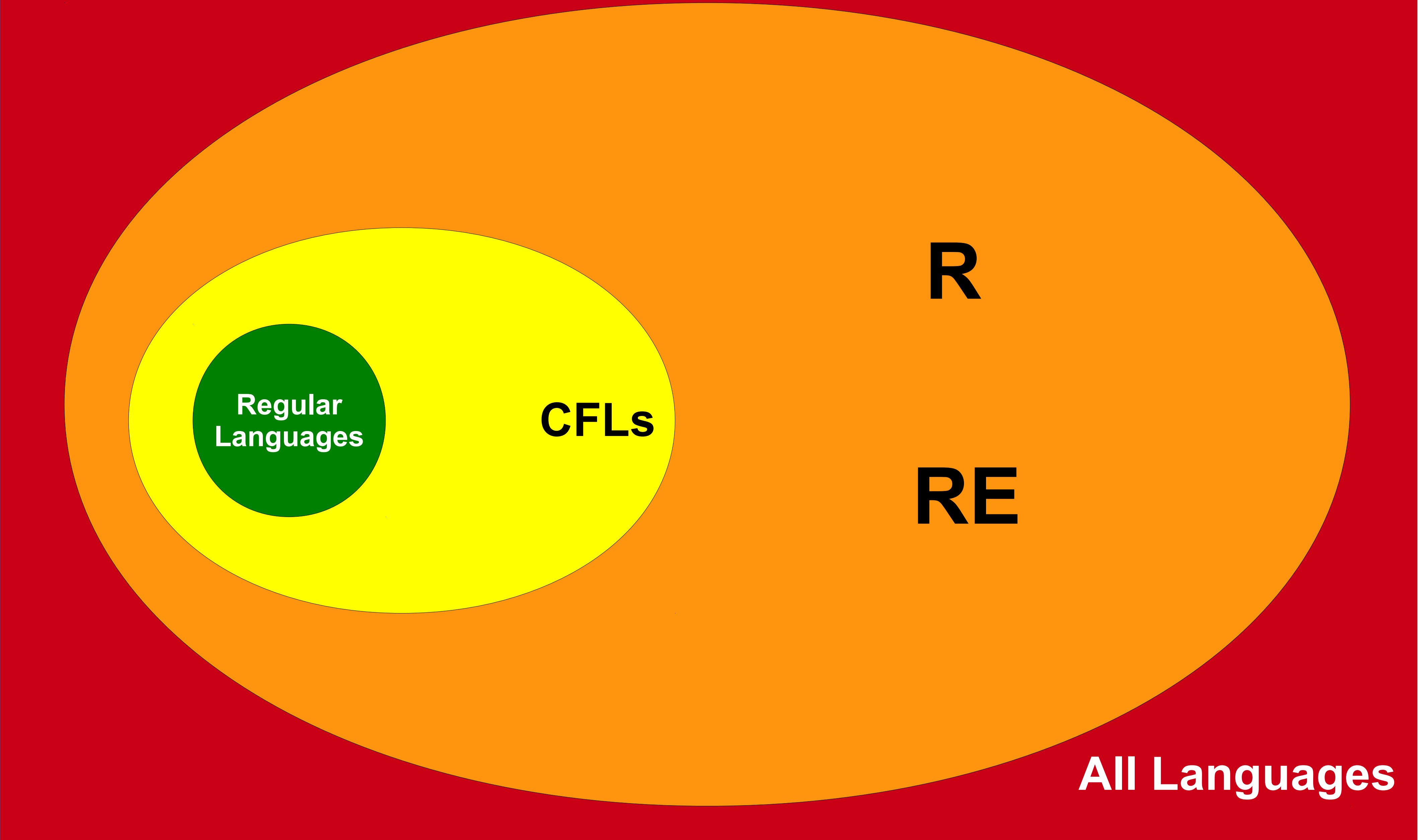
Decidable Languages (R)

- 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.

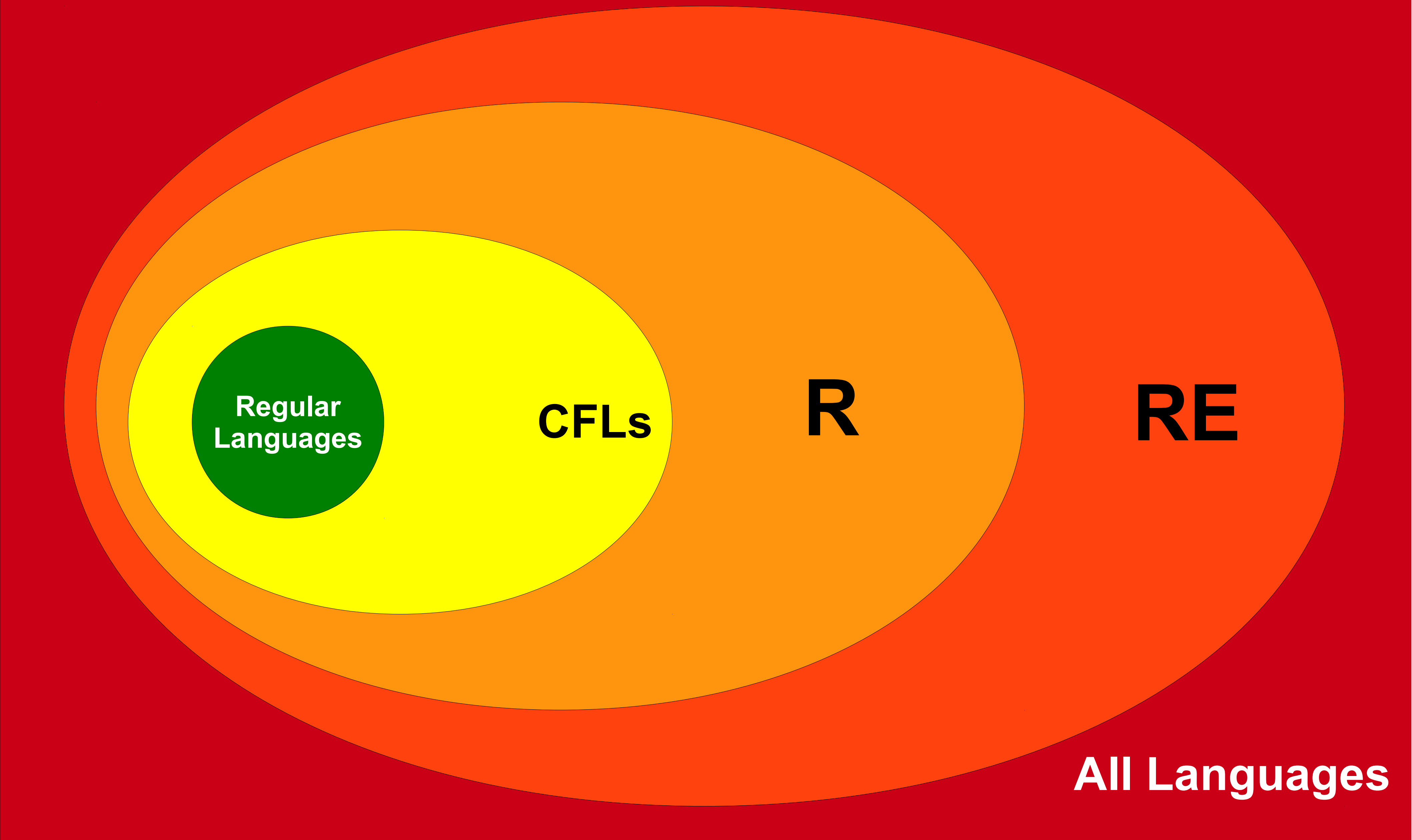
R and **RE** Languages

- Every decider is a Turing machine, but not every Turing machine is a decider.
- This means that **R** ⊆ **RE**.
- But is it a *strict* subset?
- 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?



To answer the question is R a *strict* subset of RE, I need to quickly convince you of three little things.

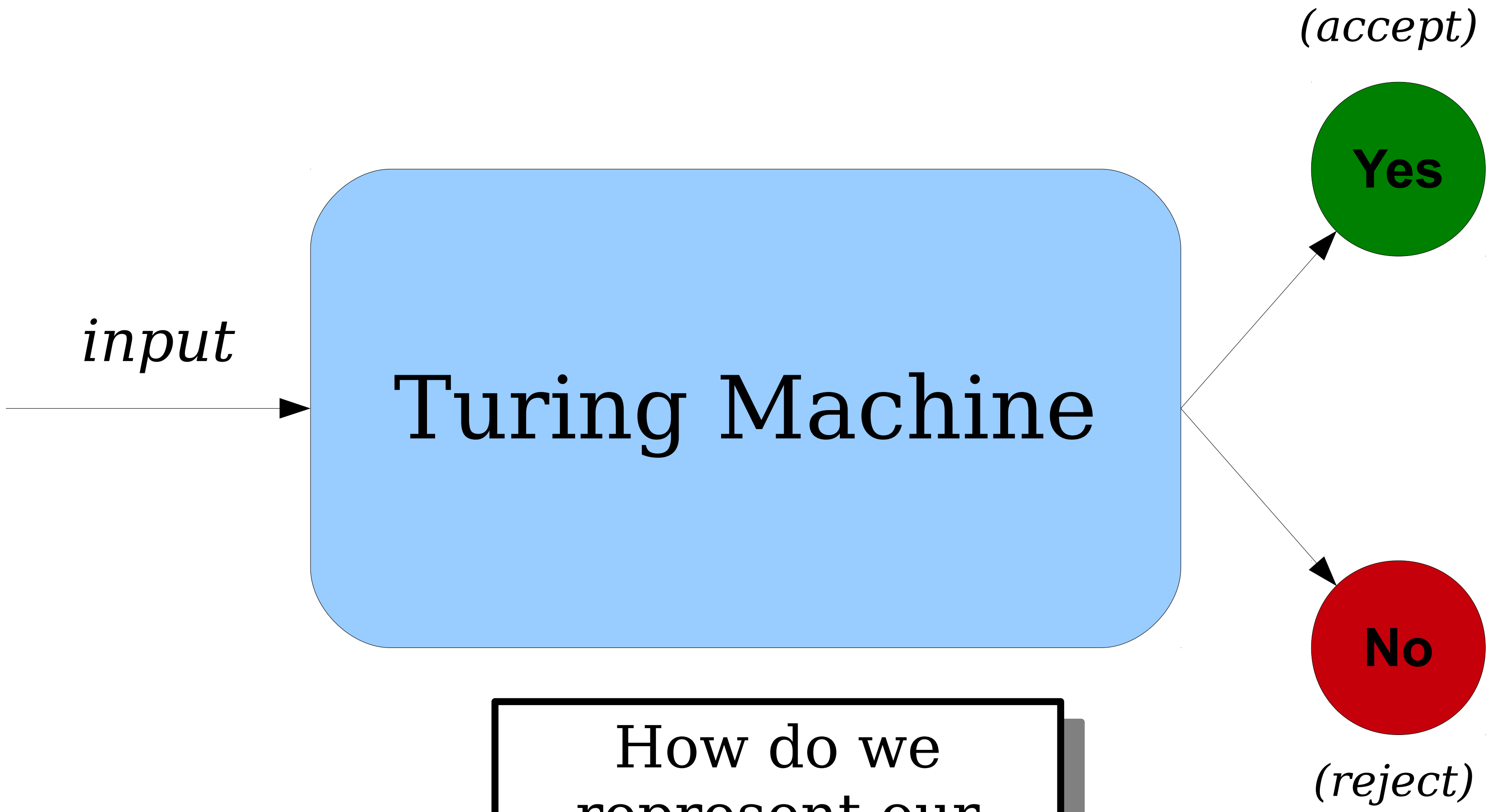
1. We can give a machine all kinds of inputs as a string, including groups of inputs and machines themselves!

2. There is a TM that can take TMs as input and run them to see what they do. (“Universality” property)

3. That TM, or any TM that takes other TMs as input, could take itself as input. (“Self-Reference” property)

1. We can give a machine all kinds of inputs as a string, including groups of inputs and machines themselves!

A Model for Solving Problems



How do we represent our inputs?

On your computer, *everything* is numbers!

• Images (gif, jpg, png):	binary numbers
• Integers (int):	binary numbers
• Non-integer real numbers (double):	binary numbers
• Letters and words (ASCII, Unicode):	binary numbers
• Music (mp3):	binary numbers
• Movies (streaming) STRANGER THINGS :	binary numbers
• Doge pictures :	binary numbers
• Email messages:	binary numbers

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.

⟨ ⟩ = 11011100101110111100010011…110

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 could say $\langle G_{0n1n} \rangle$ to mean “some encoding of a Context-Free Grammar for the language 0^n1^n” without worrying about exactly how a grammar is encoded.
  - *Intuition check*: could I type up any grammar and save it as a file on my computer? Yes? Ok then, we know we can put a grammar into a string.
- **As long as we’re convinced a thing could be saved in a file**, we don’t spend time specifying the exact file format in our proof (some proofs do in certain circumstances where it might be questioned where it’s possible, but in this class we won’t).

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.
  - Think of it like a .zip file!
- We'll denote the encoding of all of these objects as a single string by $\boldsymbol{\langle Obj_1, \ldots, Obj_n \rangle}$.
  - This lets us feed a group of inputs into our computational device as a single input.

2. There is a TM that can take TMs as input and run them to see what they do.

An Observation

- When we've been discussing Turing machines, we've talked about designing specific TMs to solve specific problems.
- Does this match your real-world experiences? Do you have one computing device for each task you need to perform?

Can there be a program that simulates what another program does?

- Sure.
- These programs go by many names:
  - An ***interpreter***, like the Java Virtual Machine or most implementations of Python.
  - A ***virtual machine***, like VMWare or VirtualBox, that simulates an entire computer.

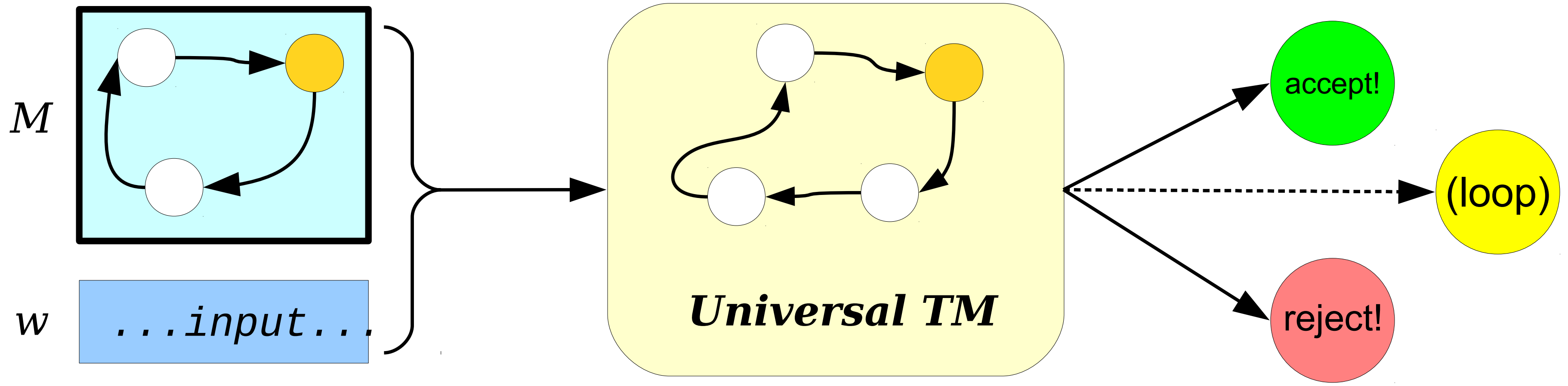
Can there be a TM that simulates what another TM does?

- Sure.
- You could imagine a TM taking a TM table like this as input, and then seeing how it would perform on various sample inputs by simulating its behavior by referring to the table at each step.

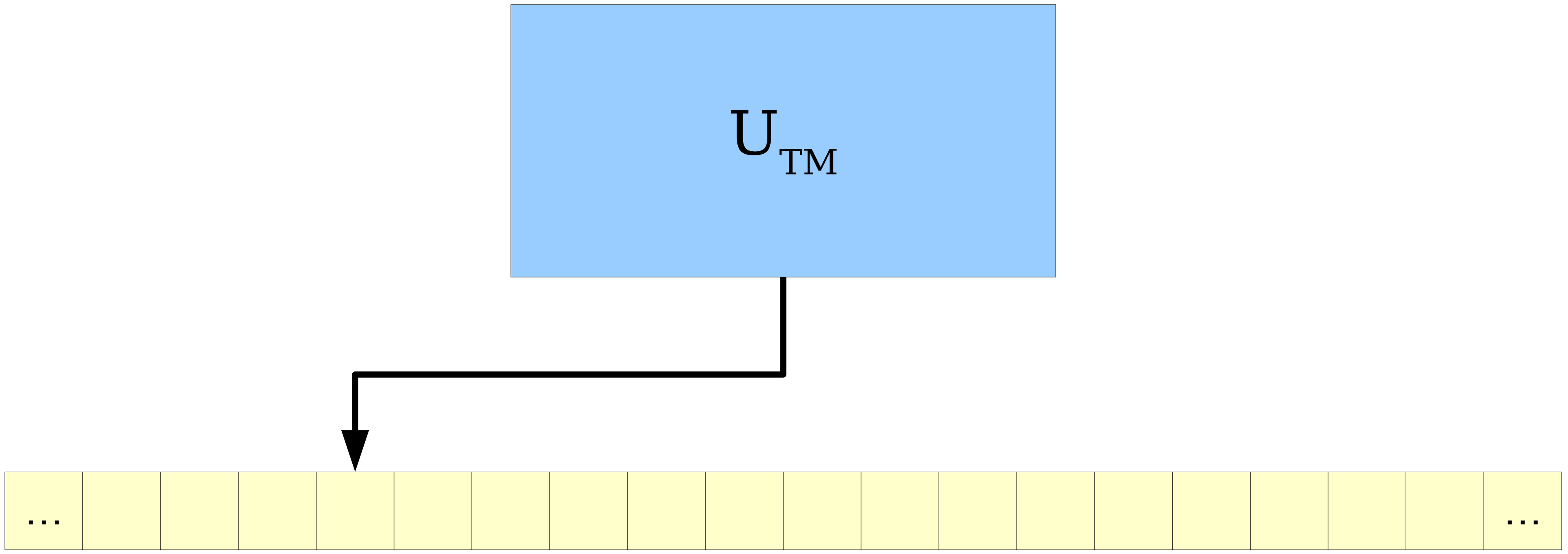
	0			**1**			□		
q_0	q_1	□	**R**	q_r	□	**R**	q_a	□	**R**
q_1	q_1	**0**	**R**	q_1	**1**	**R**	q_2	□	**L**
q_2	q_r	**0**	**R**	q_3	□	**L**	q_r	□	**R**
q_3	q_3	**0**	**L**	q_3	**1**	**L**	q_0	□	**R**

The Universal Turing Machine

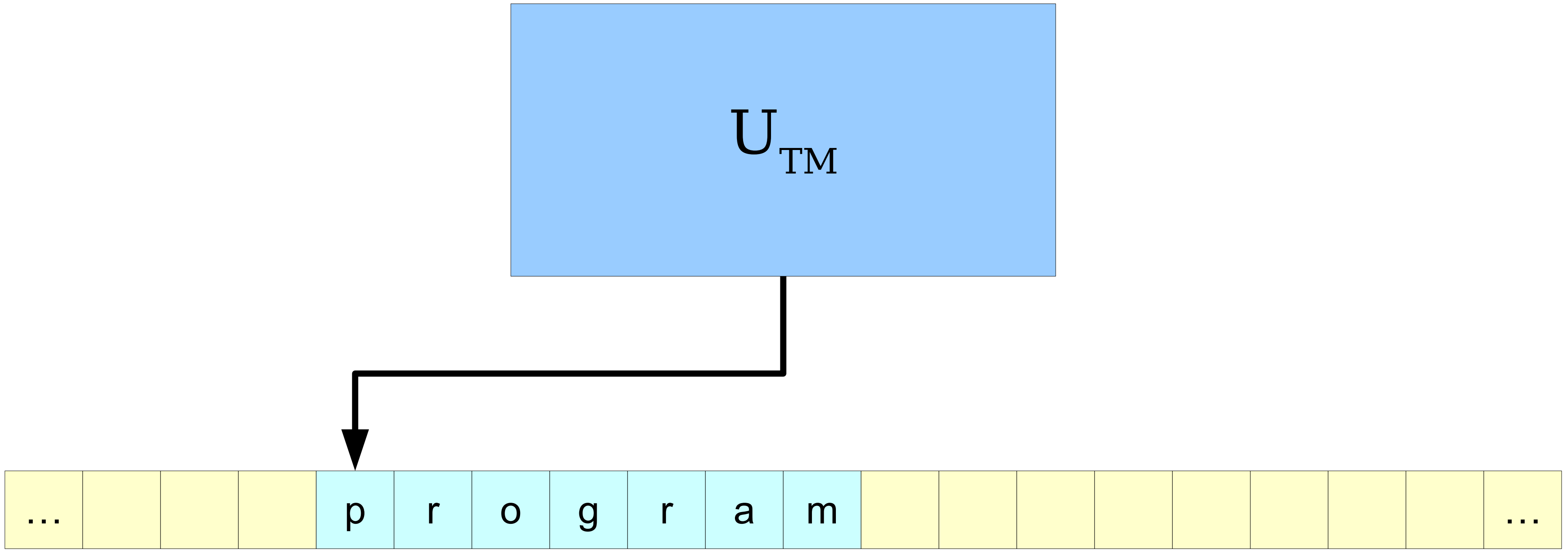
- ***Theorem (Turing, 1936)***: There is a Turing machine **U_{TM}** called the ***universal Turing machine*** that, when run on an input of the form $\langle M, w\rangle$, where M is a Turing machine and w is a string, simulates M running on w and does whatever M does on w (accepts, rejects, or loops).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



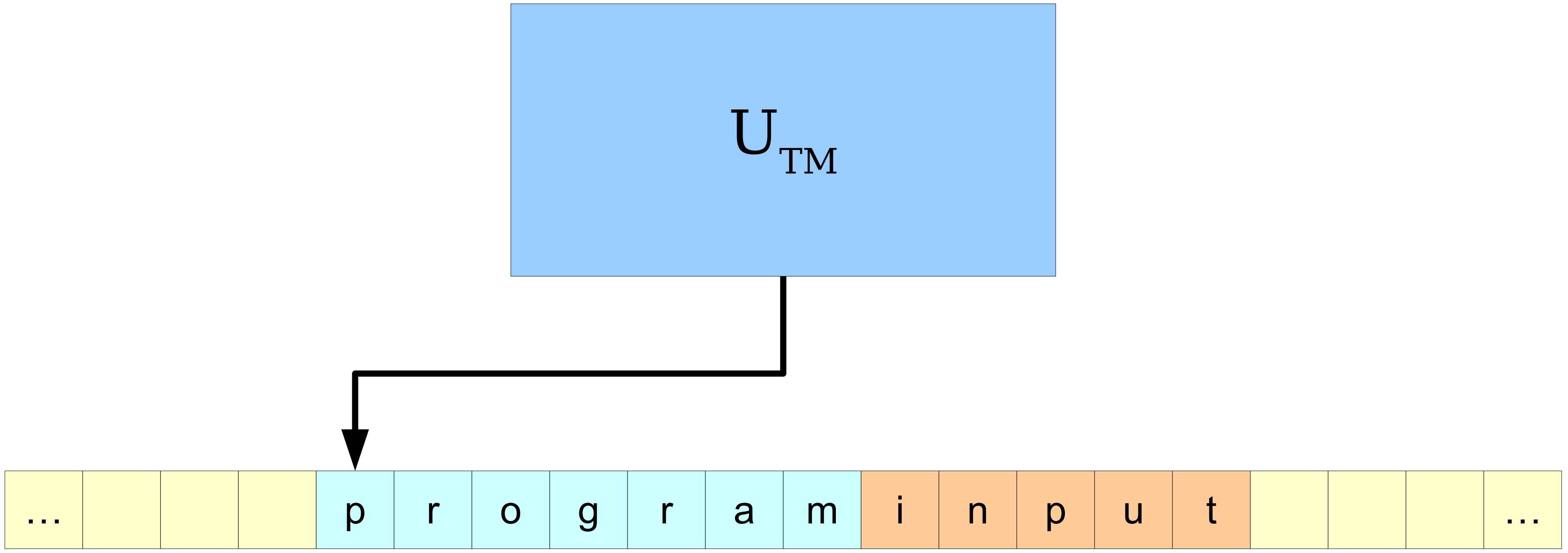
A Universal Machine



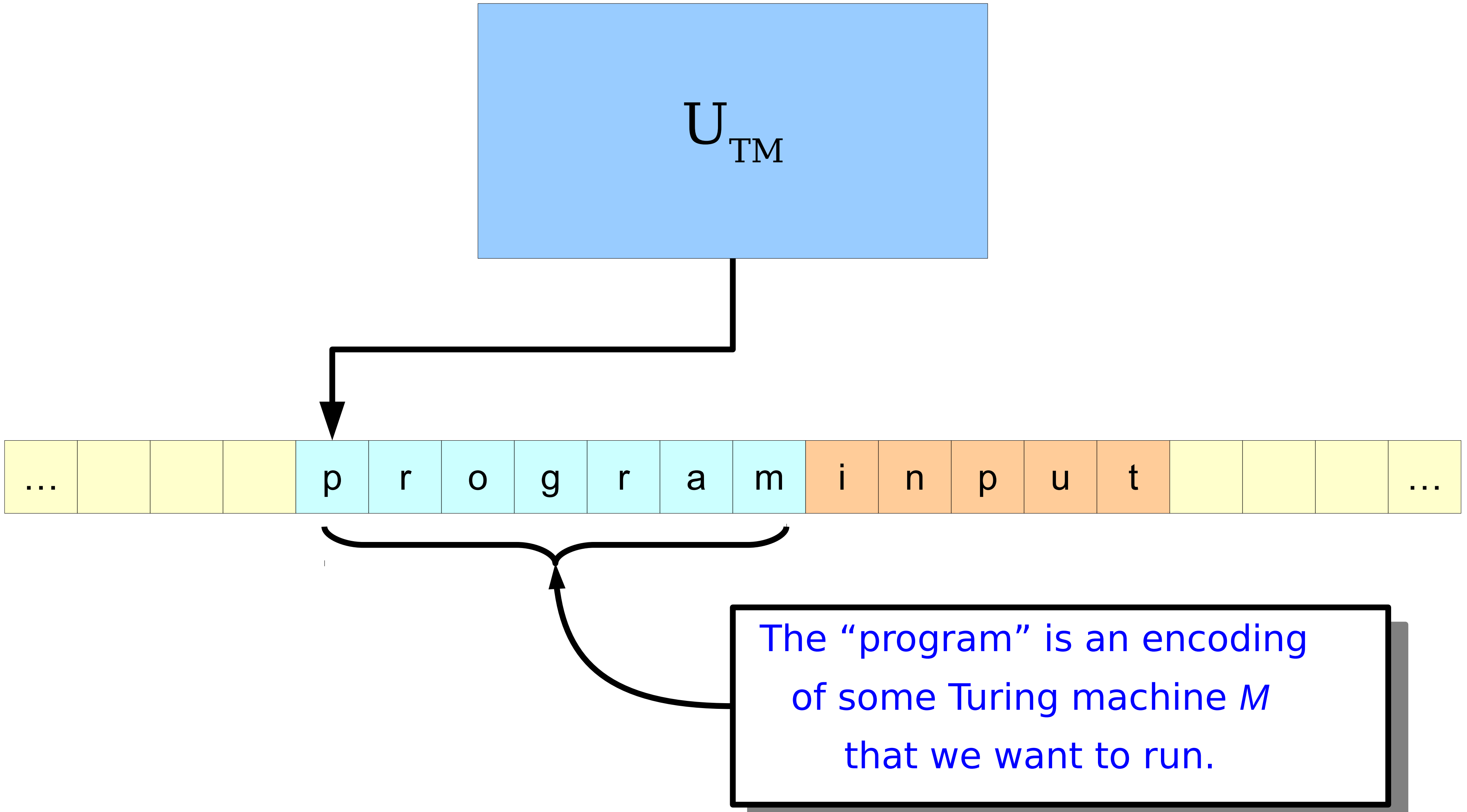
A Universal Machine



A Universal Machine

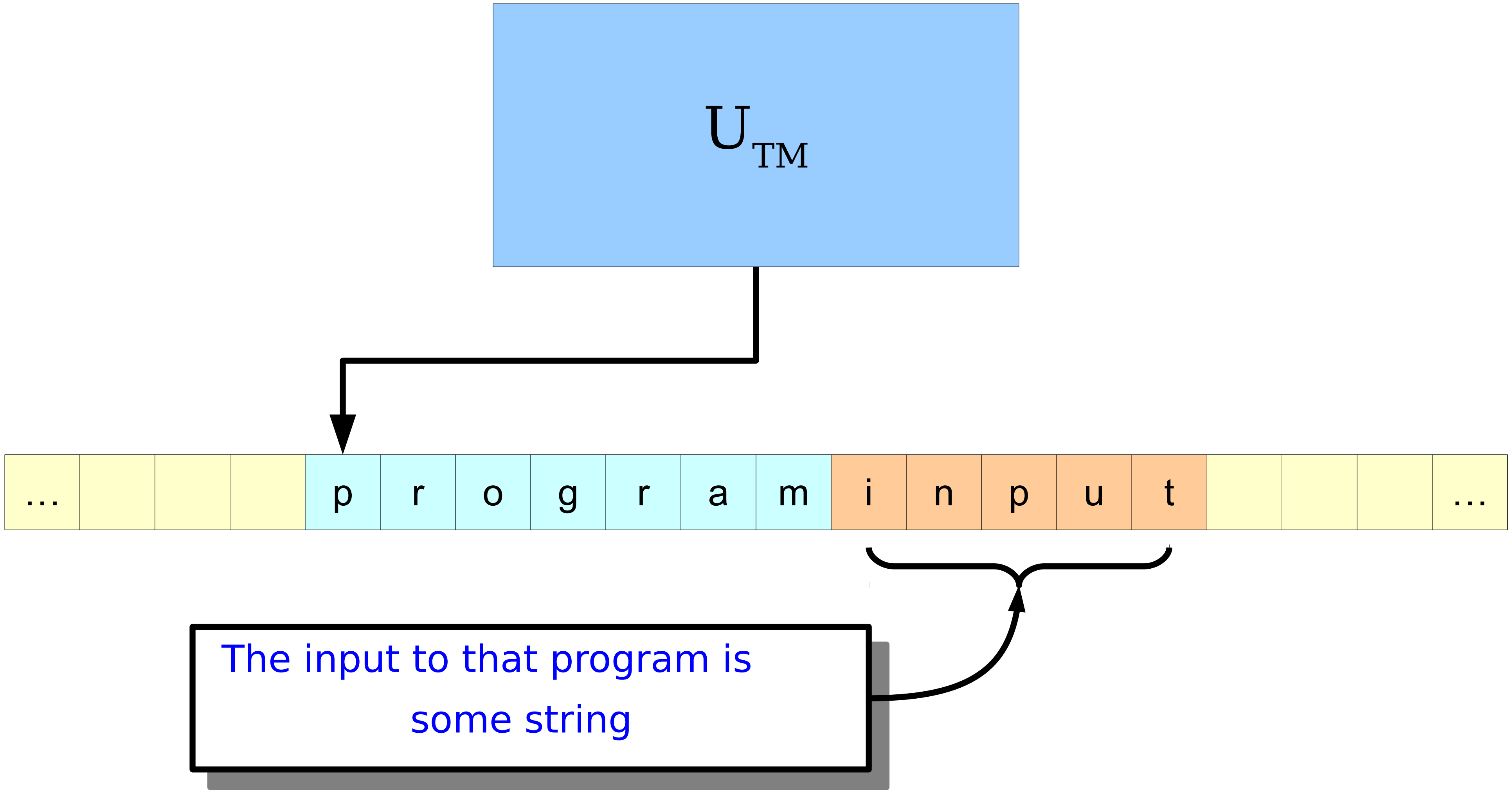


A Universal Machine



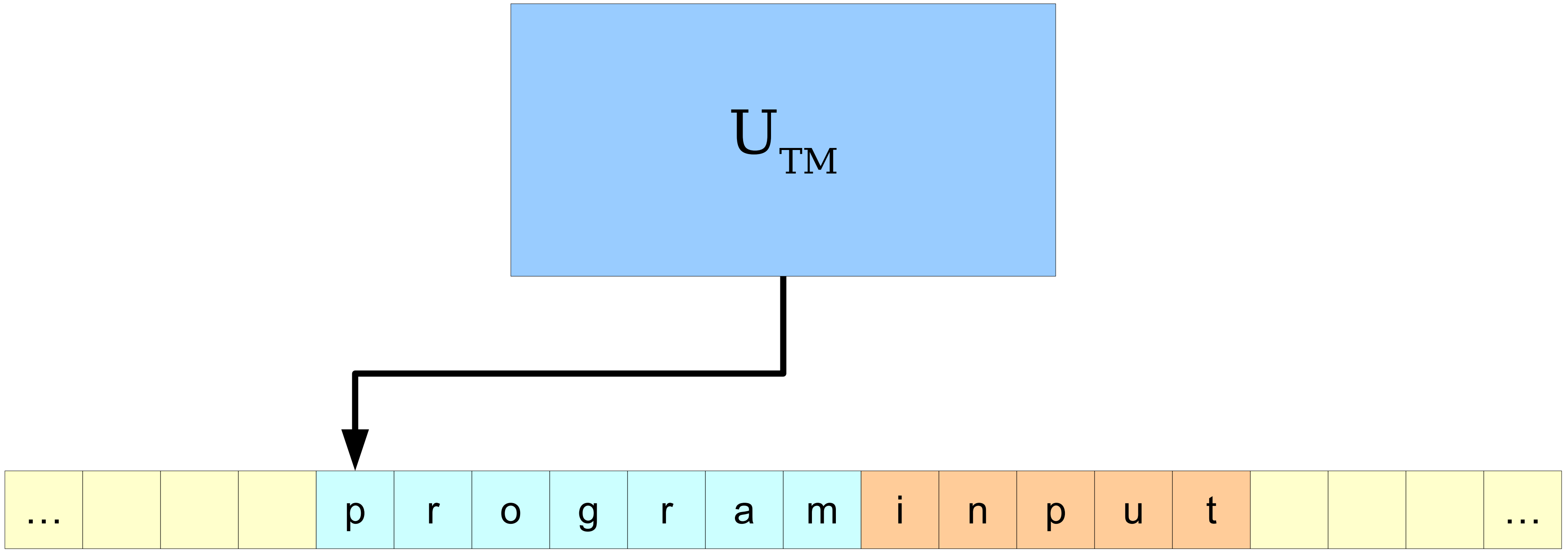
The “program” is an encoding of some Turing machine *M* that we want to run.

A Universal Machine



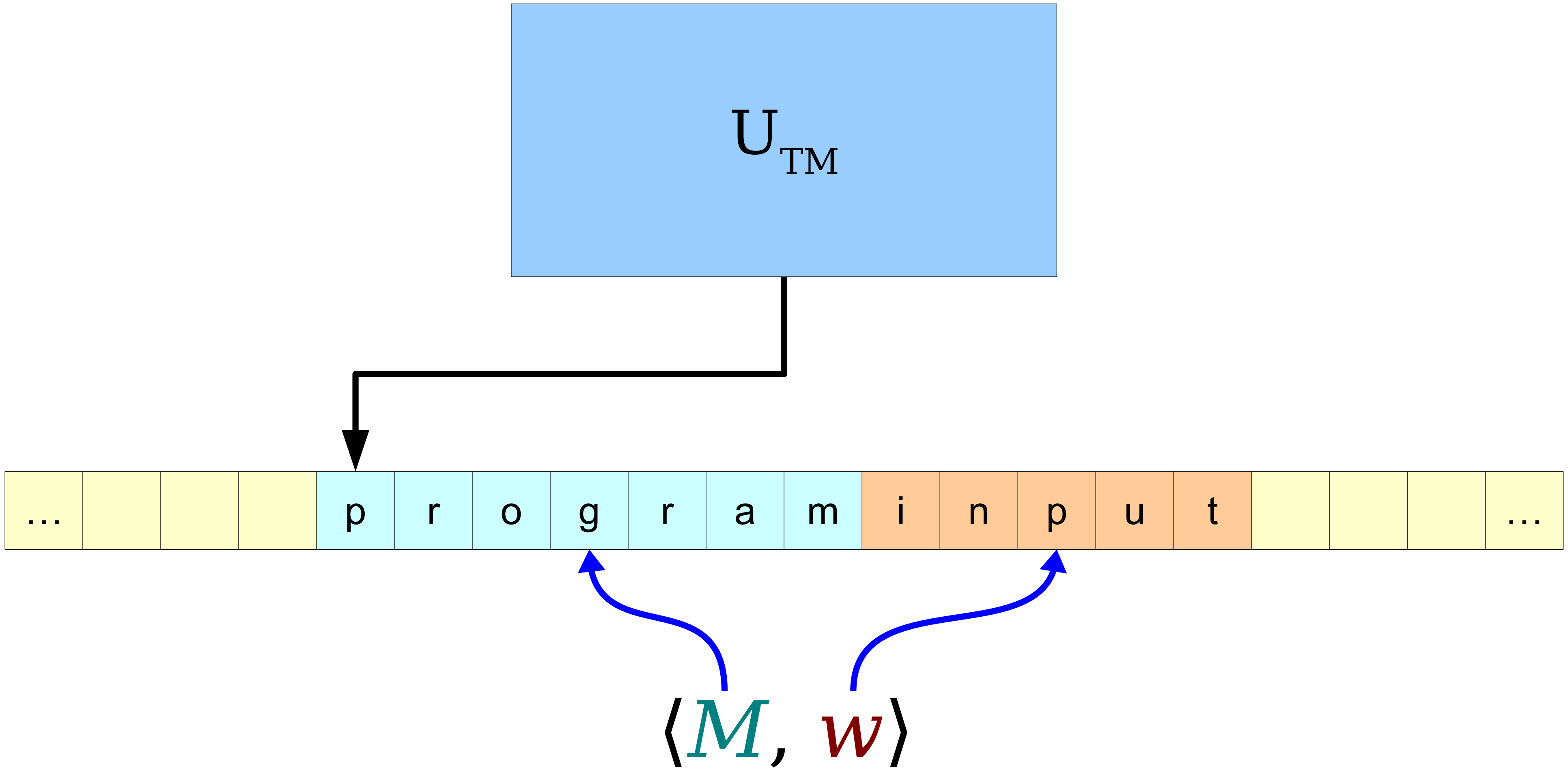
The input to that program is some string

A Universal Machine



The input has the form ⟨*M*, *w*⟩, where *M* is some TM and *w* is some string.

A Universal Machine



Since U_{TM} is a TM, it has a language.

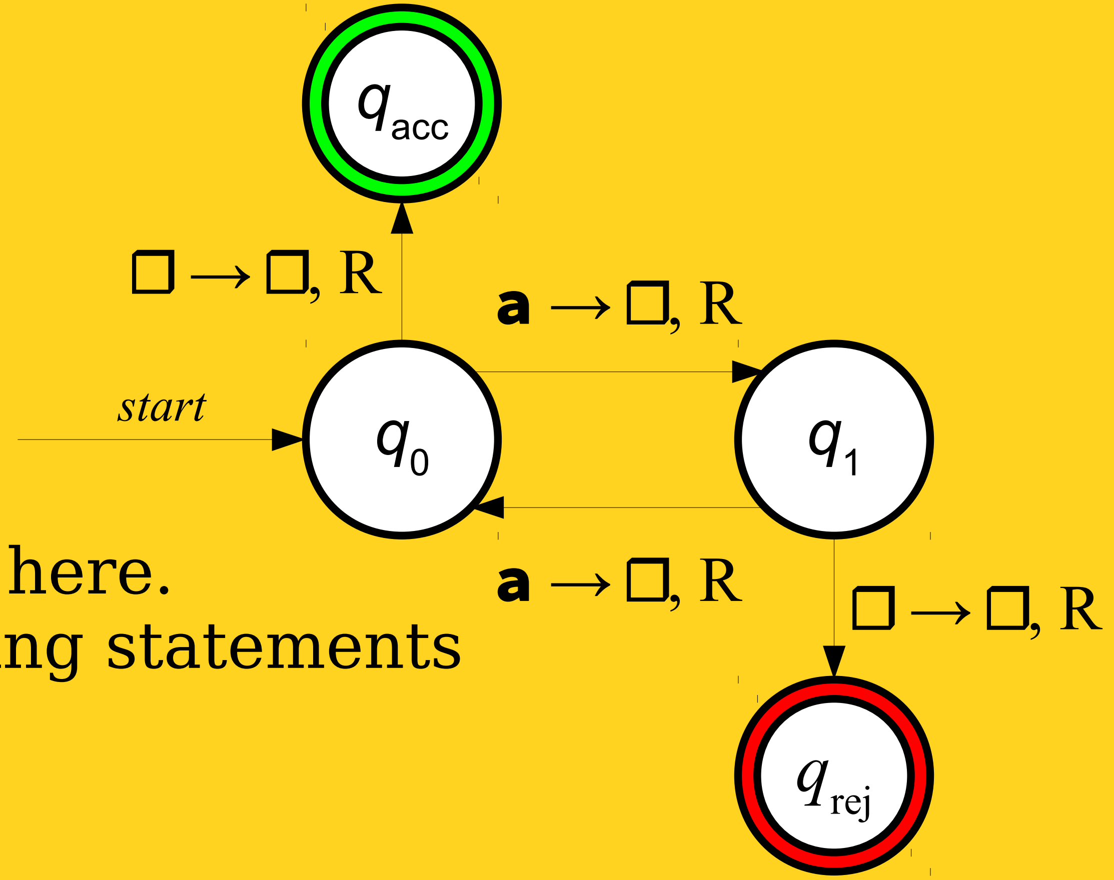
What is the language of the universal Turing machine?

The Language of U_{TM}

- U_{TM} accepts $\langle M, w \rangle$ iff M is a TM that accepts w.
- Therefore:

  $\mathscr{L}(\mathbf{U_{TM}}) = \{ \langle M, w \rangle \mid M \textbf{ is a TM and } M \textbf{ accepts } w \}$

- For simplicity, define $\mathbf{A_{TM}} = \mathscr{L}(\mathbf{U_{TM}})$. This is an important language and we'll see it many times.


q_{acc}
$\square \rightarrow \square$, R
a $\rightarrow \square$, R
start
q_0
q_1
a $\rightarrow \square$, R
$\square \rightarrow \square$, R
q_{rej}

Let M be the TM shown here.
How many of the following statements are true?

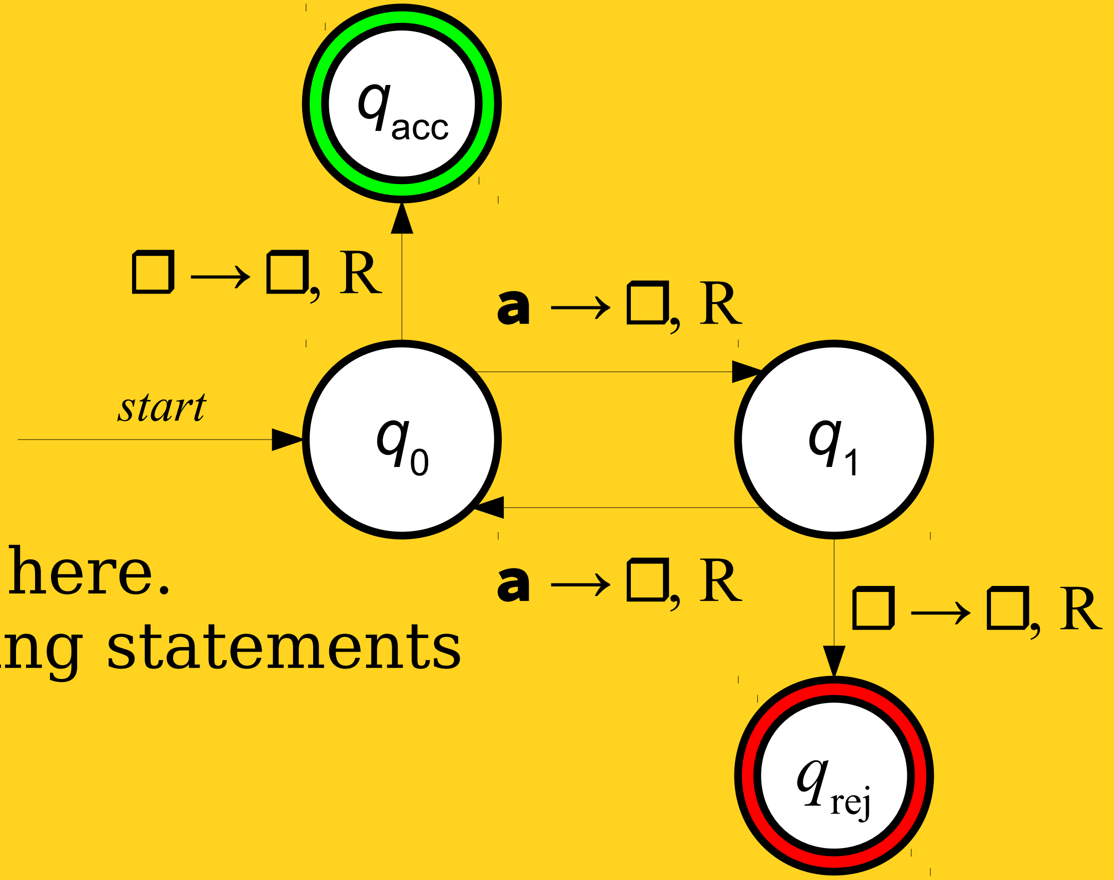
M accepts **aa**

U_{TM} accepts $\langle M,$ **aa**$\rangle$

U_{TM} accepts $\langle M,$ **ε**$\rangle$

U_{TM} accepts $\langle M,$ **a**$\rangle$

Answer at **PollEv.com/cs103** or
text **CS103** to **22333** once to join, then **a number**.



Let M be the TM shown here.
How many of the following statements are true?

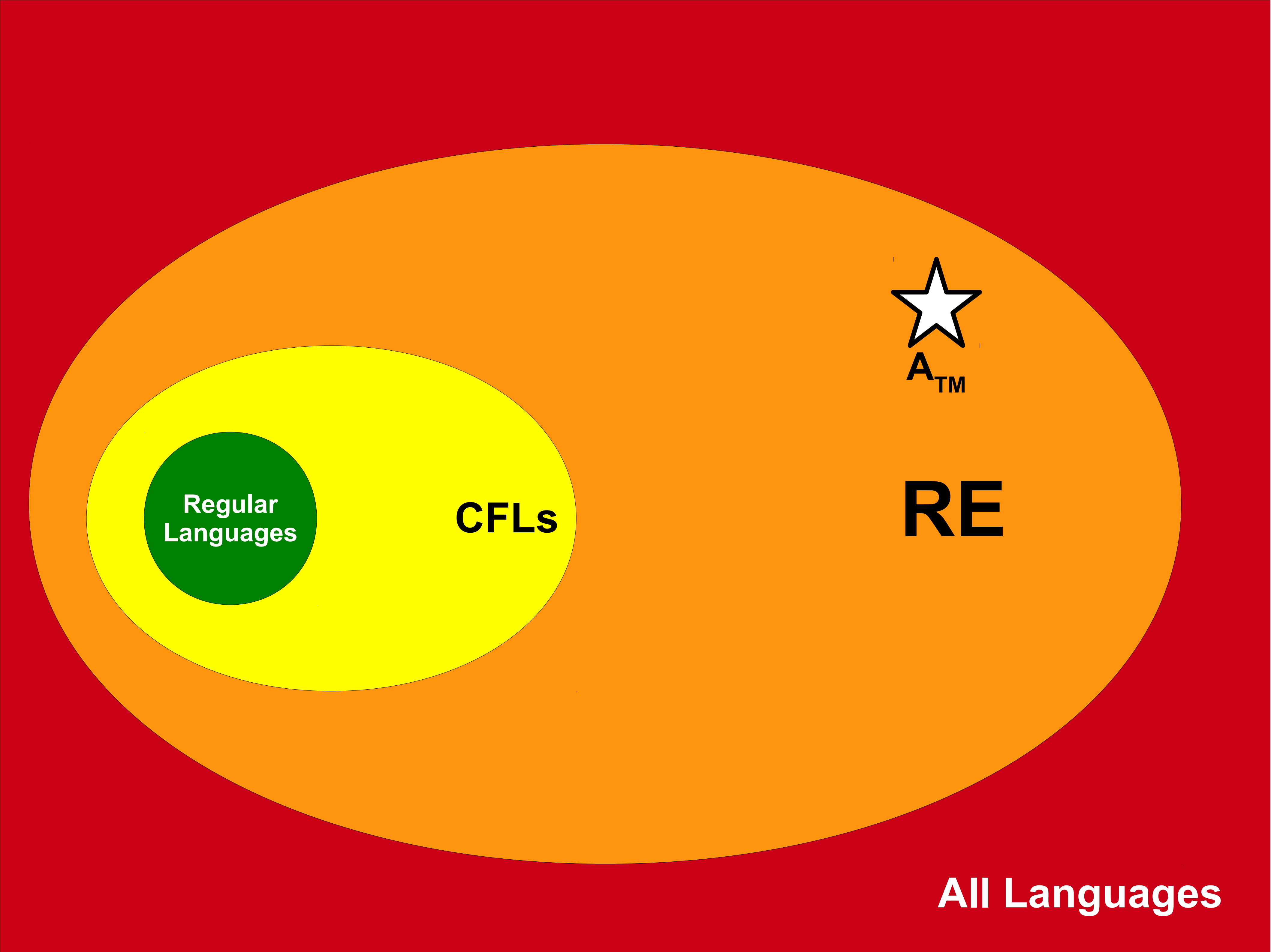
M accepts **aa**

U_{TM} accepts $\langle M, \textbf{aa} \rangle$

U_{TM} accepts $\langle M, \boldsymbol{\varepsilon} \rangle$ is equivalent to $\langle M, \varepsilon \rangle \in A_{TM}$

U_{TM} accepts $\langle M, \textbf{a} \rangle$

Answer at **PollEv.com/cs103** or
text **CS103** to **22333** once to join, then **a number**.


Regular Languages
CFLs
A_{TM}
RE
All Languages

3. That TM, or any TM that takes other TMs as input, could take itself as input. ("Self-Reference" property)

Equivalence of TMs and Programs

- Here's a sample program we might use to model a Turing machine for { w ∈ {**a**, **b**}* | w has the same number of **a**'s and **b**'s }:

```
int main() {
    string input = getInput();
    int difference = 0;

    for (char ch: input) {
        if (ch == 'a') difference++;
        else if (ch == 'b') difference--;
        else reject();
    }

    if (difference == 0) accept();
    else reject();
}
```

Equivalence of TMs and Programs

- Now, a new fact: it’s possible to build a method `mySource()` into a program, which returns the source code of the program.
- For example, here's a narcissistic program:

```
  int main() {
     string me = mySource();
     string input = getInput();

     if (input == me) accept();
     else reject();
}
```

Equivalence of TMs and Programs

- Sometimes, TMs use other TMs as subroutines.
- We can think of a decider for a language as a method that takes in some number of arguments and returns a boolean.
- For example, a decider for $\{ \mathbf{a}^n\mathbf{b}^n \mid n \in \mathbb{N} \}$ might be represented in software as a method with this signature:

```
bool isAnBn(string w);
```

- Similarly, a decider for $\{ \langle m, n \rangle \mid m, n \in \mathbb{N}$ and m is a multiple of $n \}$ might be represented in software as a method with this signature:

```
bool isMultipleOf(int m, int n);
```

Side note: what does equivalence of TMs and programs mean for YOU?

- In this class (starting Problem Set Nine, and unless directed otherwise) you can write proofs about TMs by just writing normal code (e.g., Java or C++), and never have to painstakingly draw an actual TM ever again!

:: rejoicing ::

Self-Referential :Danger:

- So, I hope we’ve convinced you that there’s nothing magic, impossible, or scary about a program getting a string version of its own code.

- But, there are some dragons in the land of self-referential things….

True or false:

`"This string is 34 characters long."`

True or false:

```
"This string is 34 characters long."
 1234567890123456789012345678901234
```

True or false:

`"This sentence is written in blue."`

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.

Answer at **PollEv.com/cs103** or
text **CS103** to **22333** once to join, then **your response**.

MY NOSE WILL GROW NOW!

True or false:

`"This sentence is false."`

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.
  - They can even contain themselves (“The set of all sets” and “The set of all sets with infinite cardinality” both 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?

"The set of all sets that do *not* contain themselves contains itself."

Answer at **PollEv.com/cs103** or
text **CS103** to **22333** once to join, then **your answer**.

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. 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

- One way to think abo
  they lead to a kind of
  similar to the parado
  - **Thm.** There is no gre
  - **Proof, by contradict**
    contradiction that the
  - *[Now we will use g to*
    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. ■

Observation: there is other math we could have done on g that would **not** have led to an impossible situation. (For example, "Let $x = g - 1$.") "Fixing the bug" in the math doesn't actually fix anything, because it wasn't the math that was buggy. It was g itself. The math did what it needed to do to *expose* the problem with g.

More Self-Reference!

(this time with Turing Machines)

The Problem of Looping TMs

- Suppose we have a TM M and a string w.
- If we run M on w, we may never find out whether $w \in \mathscr{L}(M)$ because M might loop on w.
- Is there some algorithm we can use to determine whether M is eventually going to accept w?

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 a TM *M* and string *w*, then
  - accepts if *M* accepts *w*, and
  - rejects if *M* does not accept *w*.

A Decider for `willAccept`?

- To make the previous discussion more concrete, let's explore the analog for computer programs.
- If A_{TM} is decidable, we could construct a function

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

  that takes in as input a program and a string, then returns true if the program will accept the input and false otherwise.
- What could we do with this?

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

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

that takes in as input a program and a string, then returns true if the program will accept the input and false otherwise.

How many of the following statements are true?

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

Answer at **PollEv.com/cs103** or
text **CS103** to **22333** once to join, then **a number**.

What does this program do?

```
bool willAccept(string program, string input)
{
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (willAccept(me, input)) {
      reject();
   } else {
      accept();
   }
}
```

What does this program do?

```
bool willAccept(string program, string input)
{
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (willAcce
      reject();
   } else {
      accept();
   }
}
```

How many of the following statements are true?

This program accepts at least one input.
This program rejects at least one input.
This program loops on at least one input.

Answer at **PollEv.com/cs103** or
text **CS103** to **22333** once to join, then **a number**.

What does this program do?

```
bool willAccept(string program, string input)
{
   /* … some implementation … */
}

int main() {
   string me = mySource();
   string input = getInput();

   if (willAccept(me, inp
      reject();
   } else {
      accept();
   }
}
```

Try running this program on any input.
What happens if

… this program accepts its input?

What does this program do?

```
bool willAccept(string program, string input)
{
    /* … some implementation … */
}

int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, inp
        reject();
    } else {
        accept();
    }
}
```

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?

What does this program do?

```
bool willAccept(string program, string input)
{
   /* … some implementation … */
}


int main() {
   string me = mySource();
   string input = getInput();

   if (willAccept(me, inp
      reject();
   } else {
      accept();
   }
}
```

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!

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 Time

- Today we explored in code how assuming $\mathbf{A_{TM}}$ is decidable leads to an impossible situation with only one resolution: A_{TM} must not be decidable!
- ***Next time we will write this up as an actual proof by contradiction.***