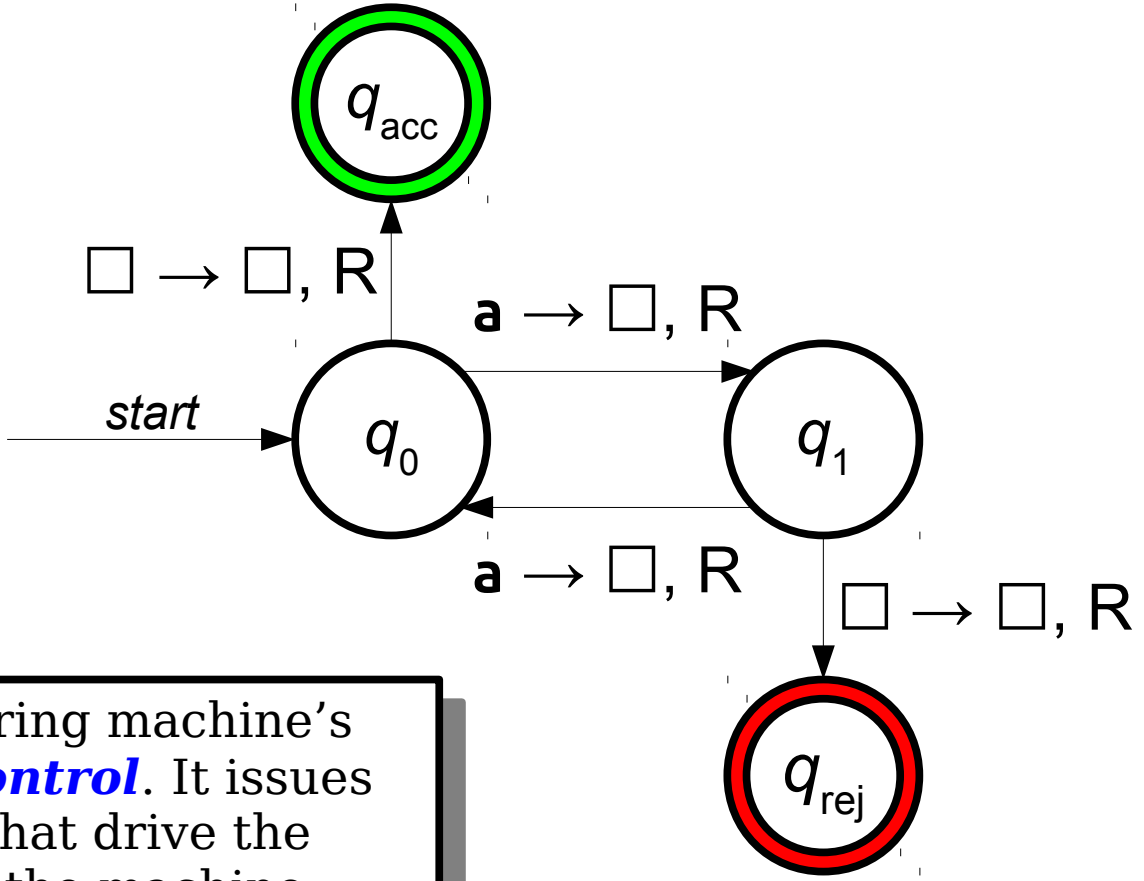
Turing Machines

Part Two

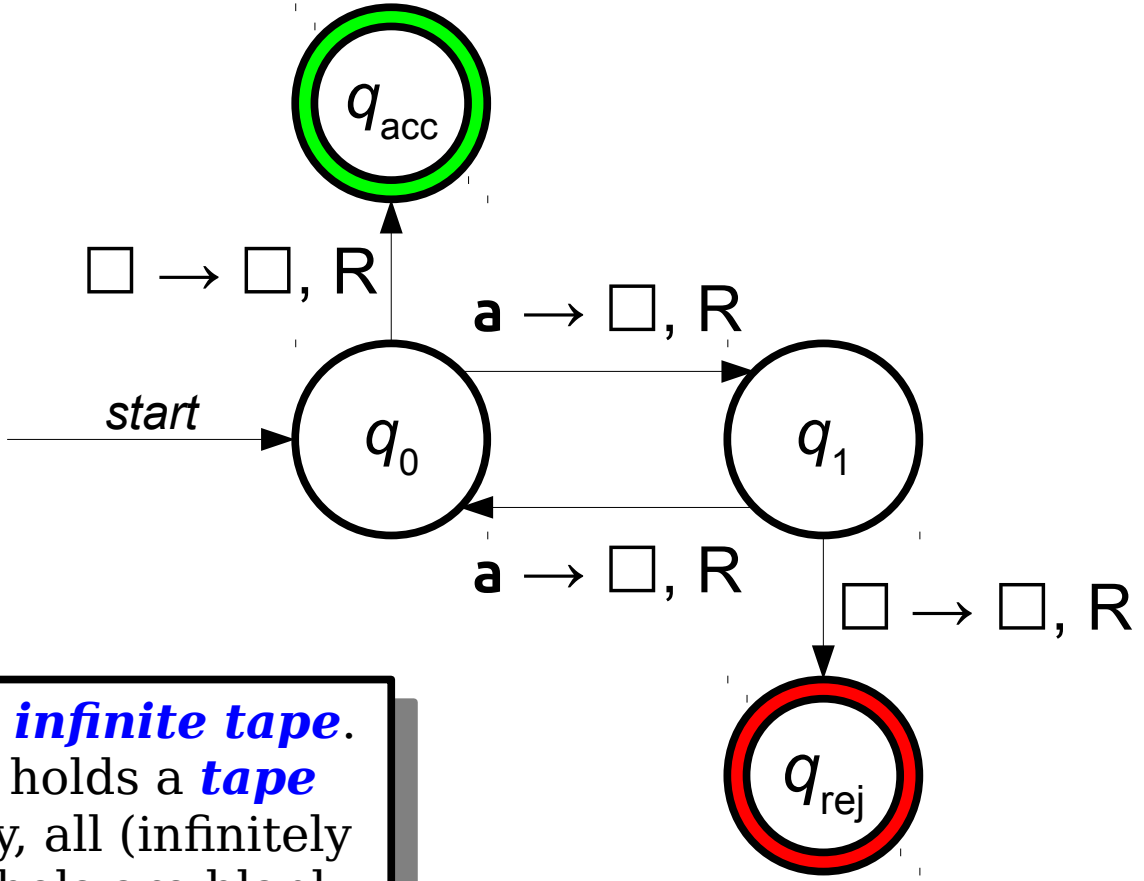
Recap from Last Time

Our First Turing Machine

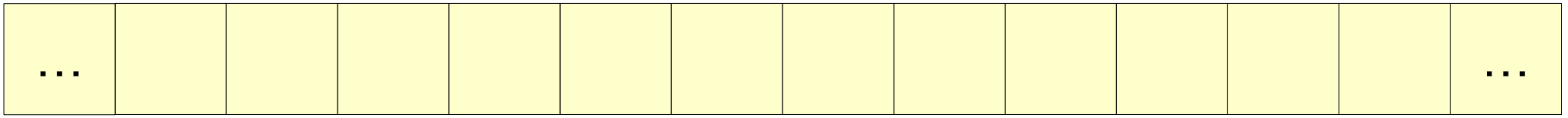


This is the Turing machine's ***finite state control***. It issues commands that drive the operation of the machine.

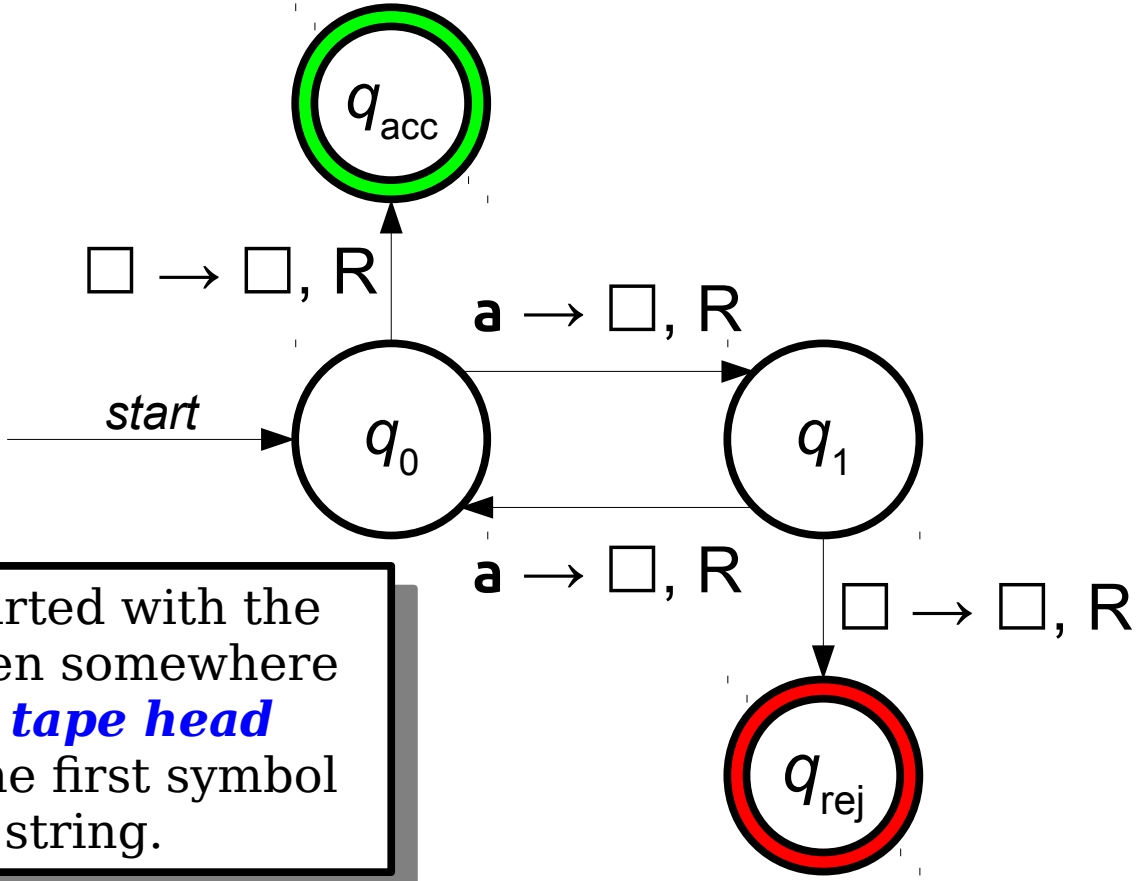
Our First Turing Machine



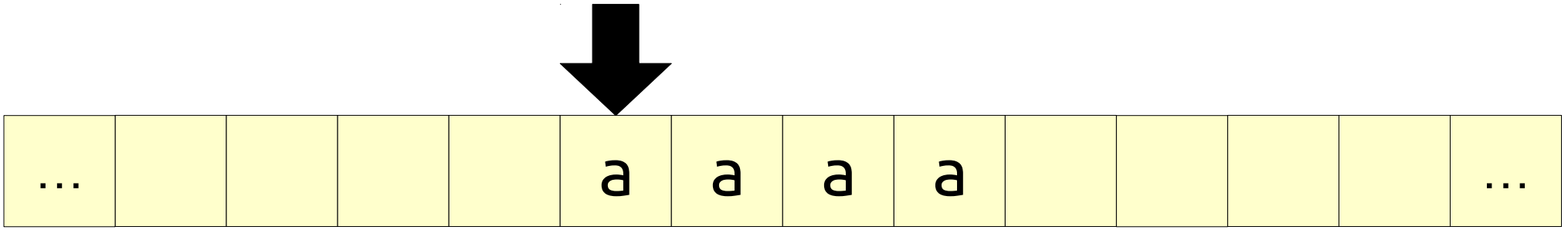
This is the TM's ***infinite tape***. Each tape cell holds a ***tape symbol***. Initially, all (infinitely many) tape symbols are blank.



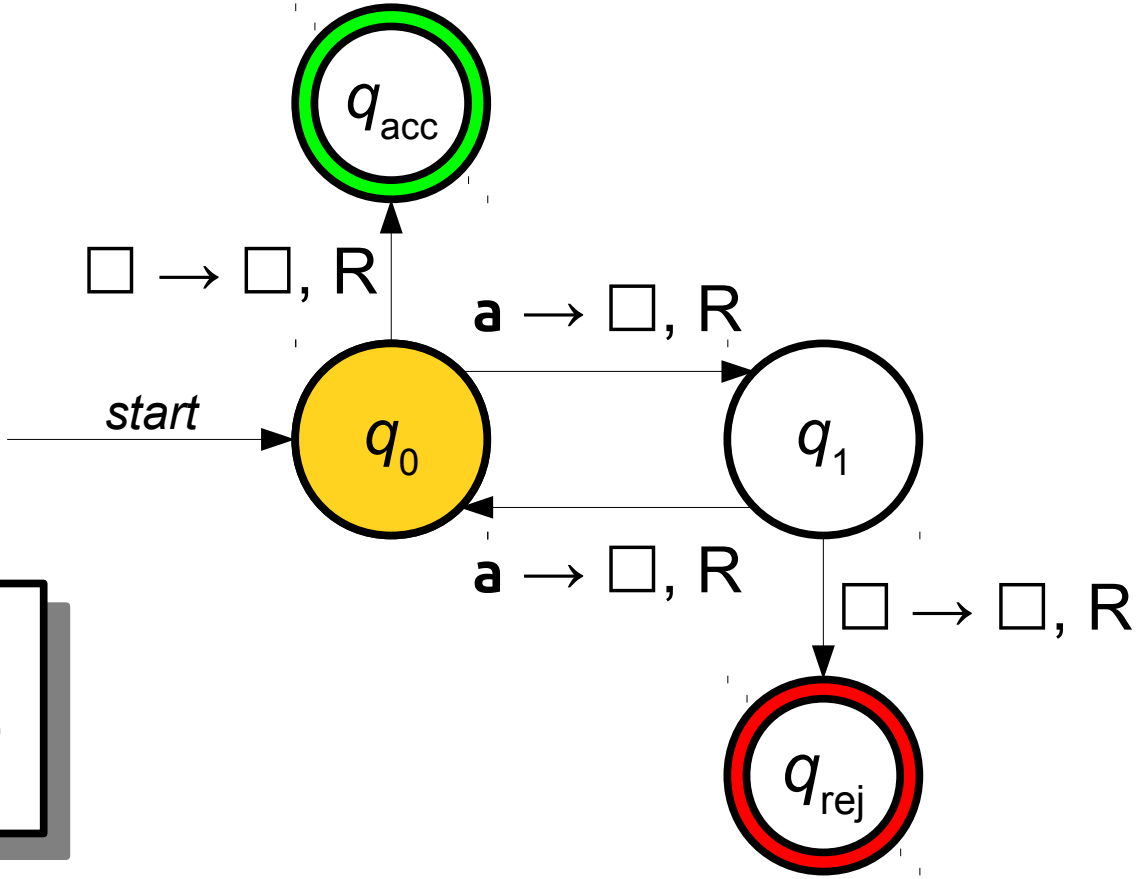
Our First Turing Machine



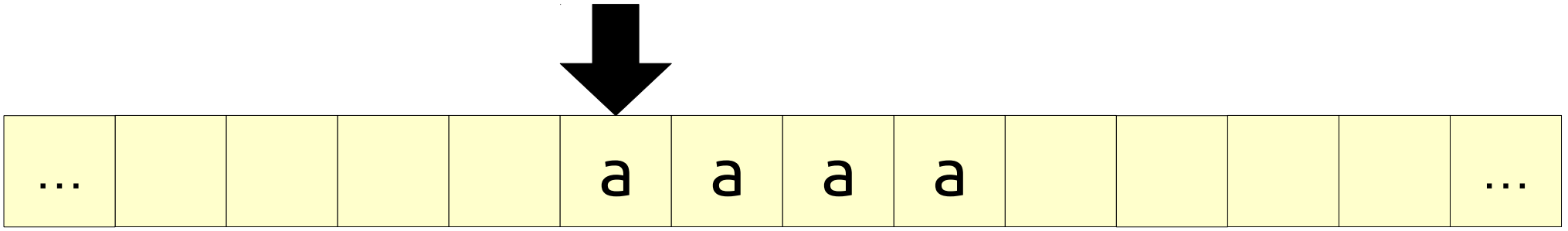
The machine is started with the ***input string*** written somewhere on the tape. The ***tape head*** initially points to the first symbol of the input string.



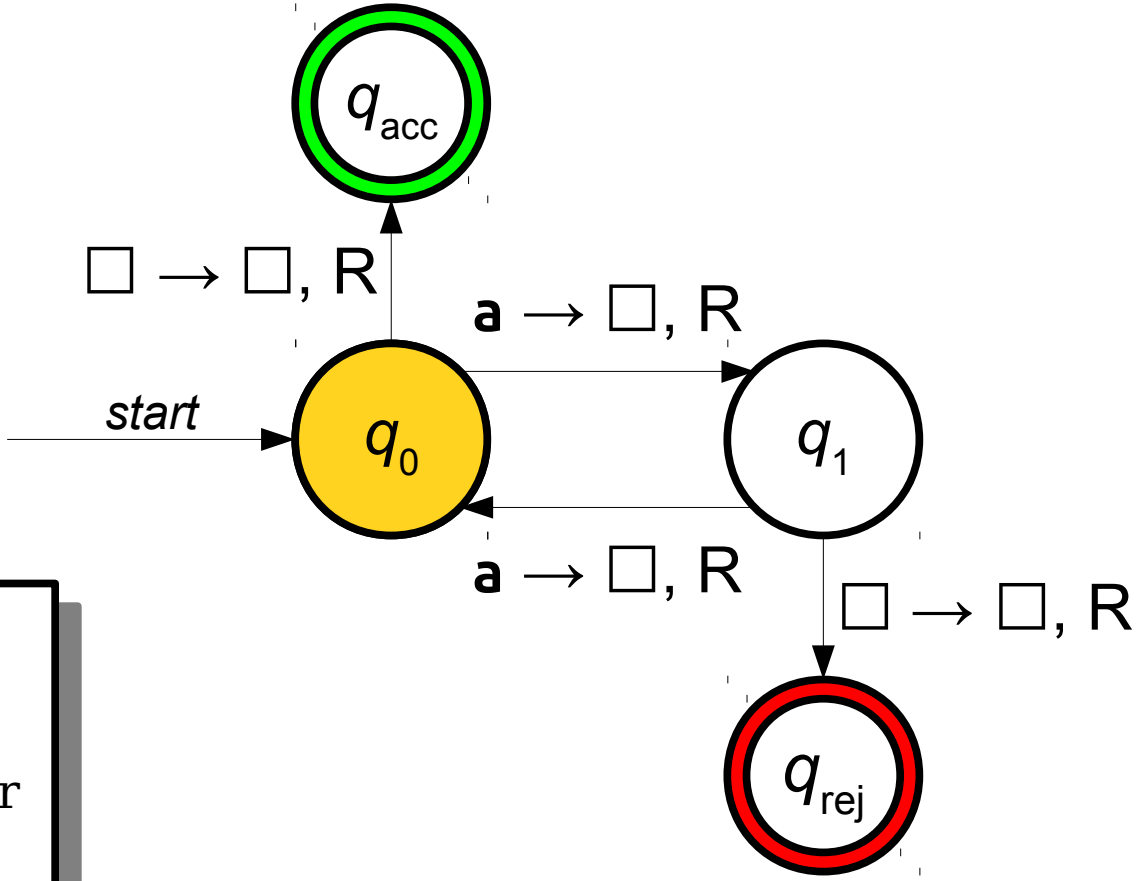
Our First Turing Machine



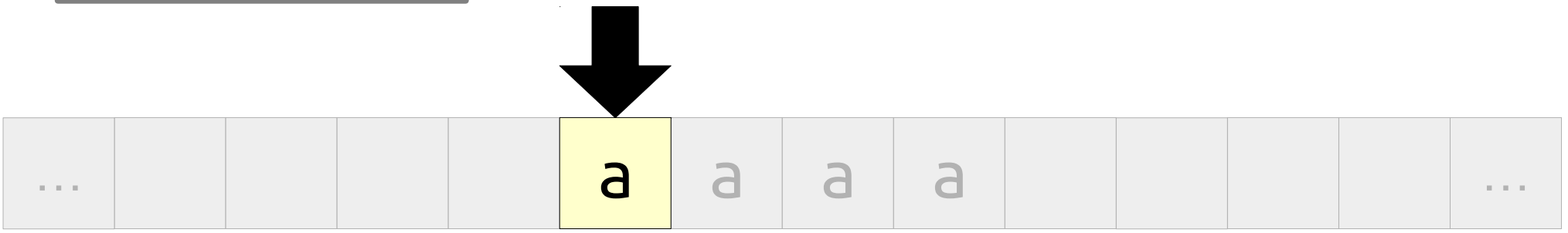
Like DFAs and NFAs, TMs begin execution in their ***start state***.



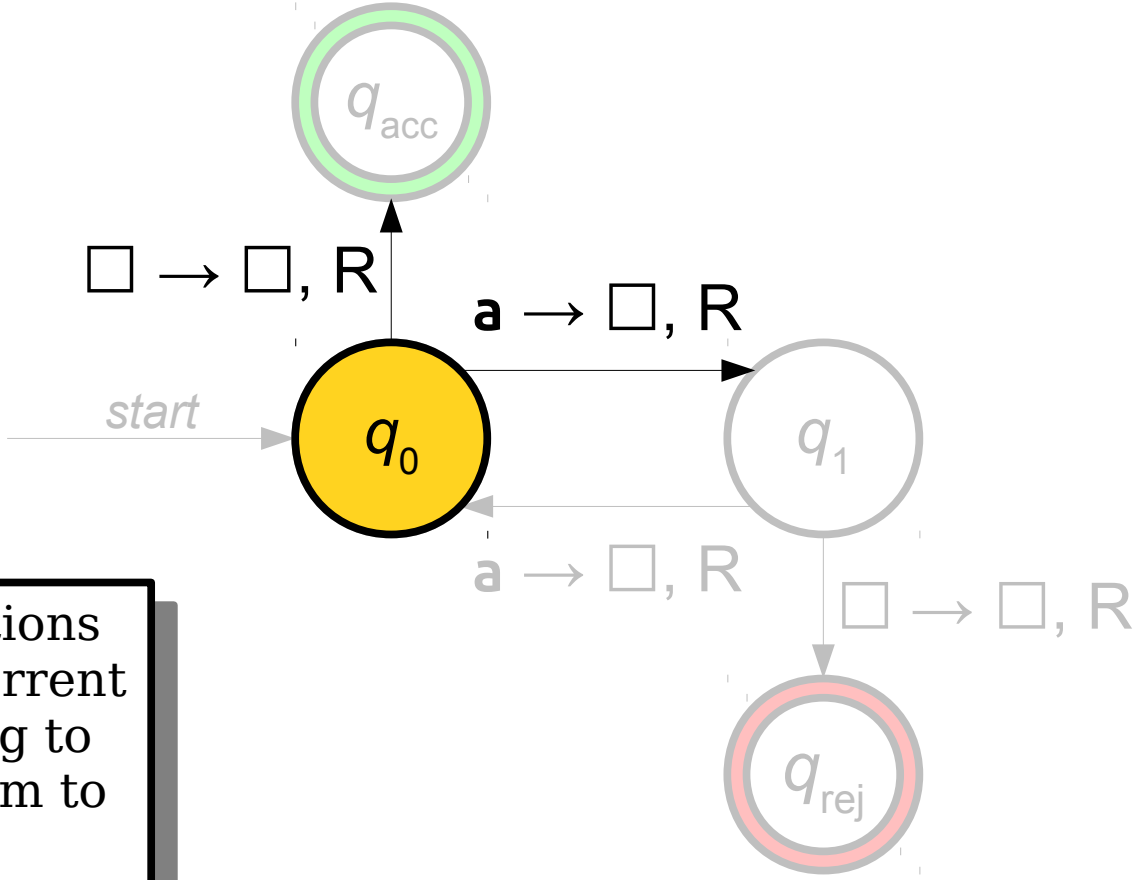
Our First Turing Machine



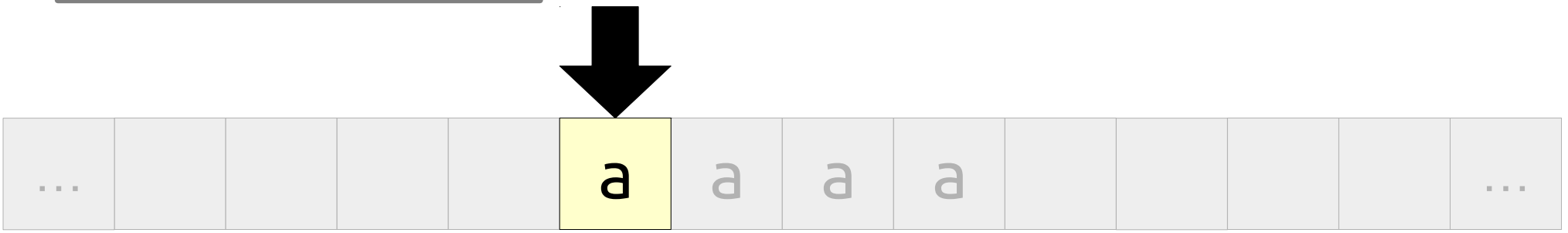
At each step, the TM only looks at the symbol immediately under the ***tape head***.



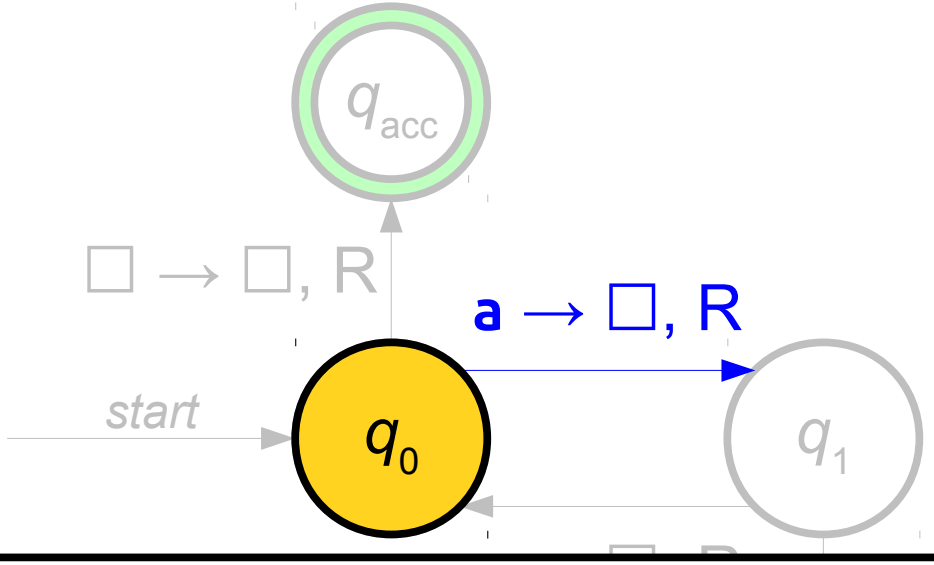
Our First Turing Machine



These two transitions originate at the current state. We're going to choose one of them to follow.



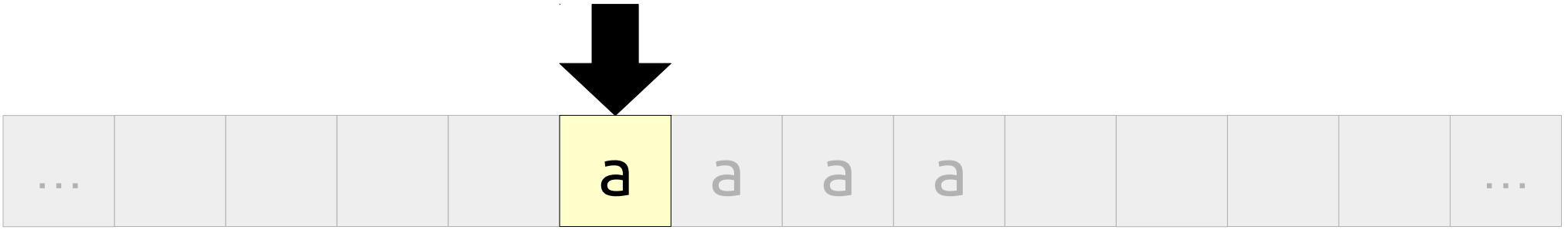
Our First Turing Machine



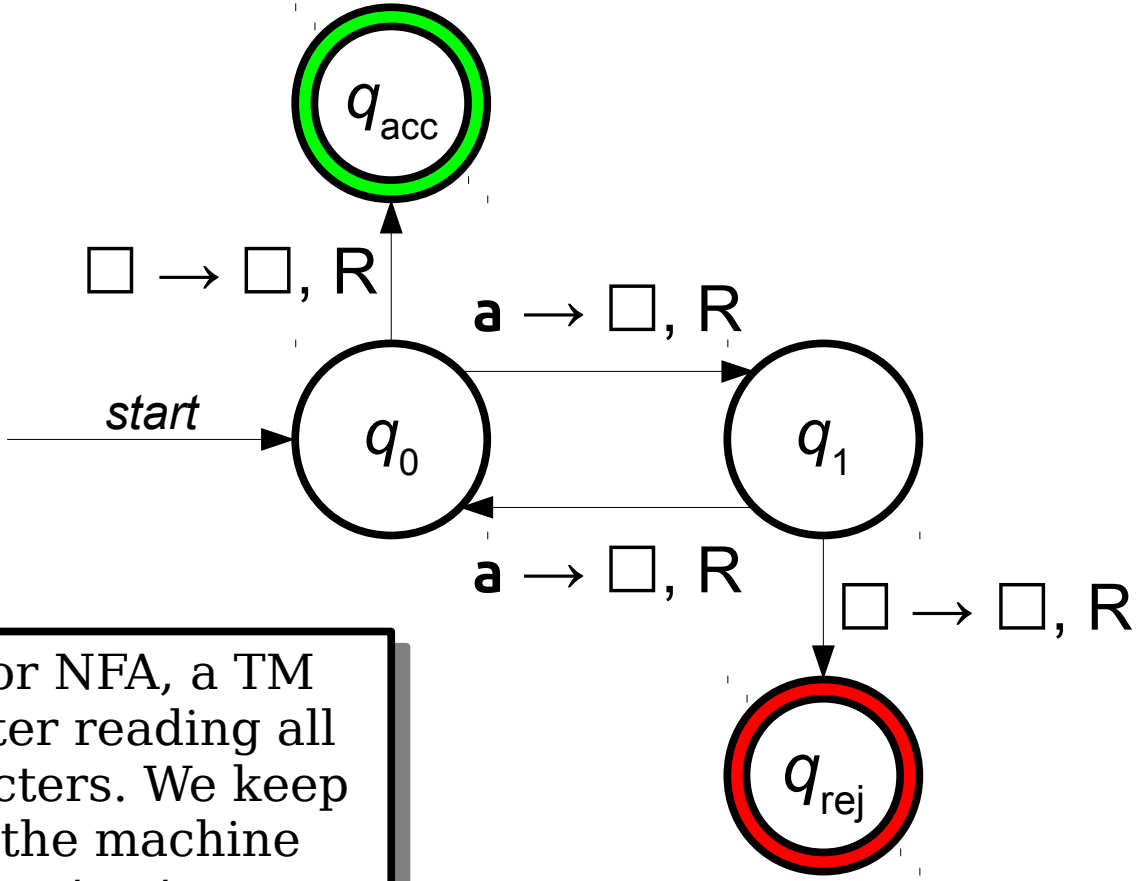
Each transition has the form

read → write, dir

and means "if symbol ***read*** is under the tape head, replace it with ***write*** and move the tape head in direction ***dir*** (L or R). The □ symbol denotes a blank cell.

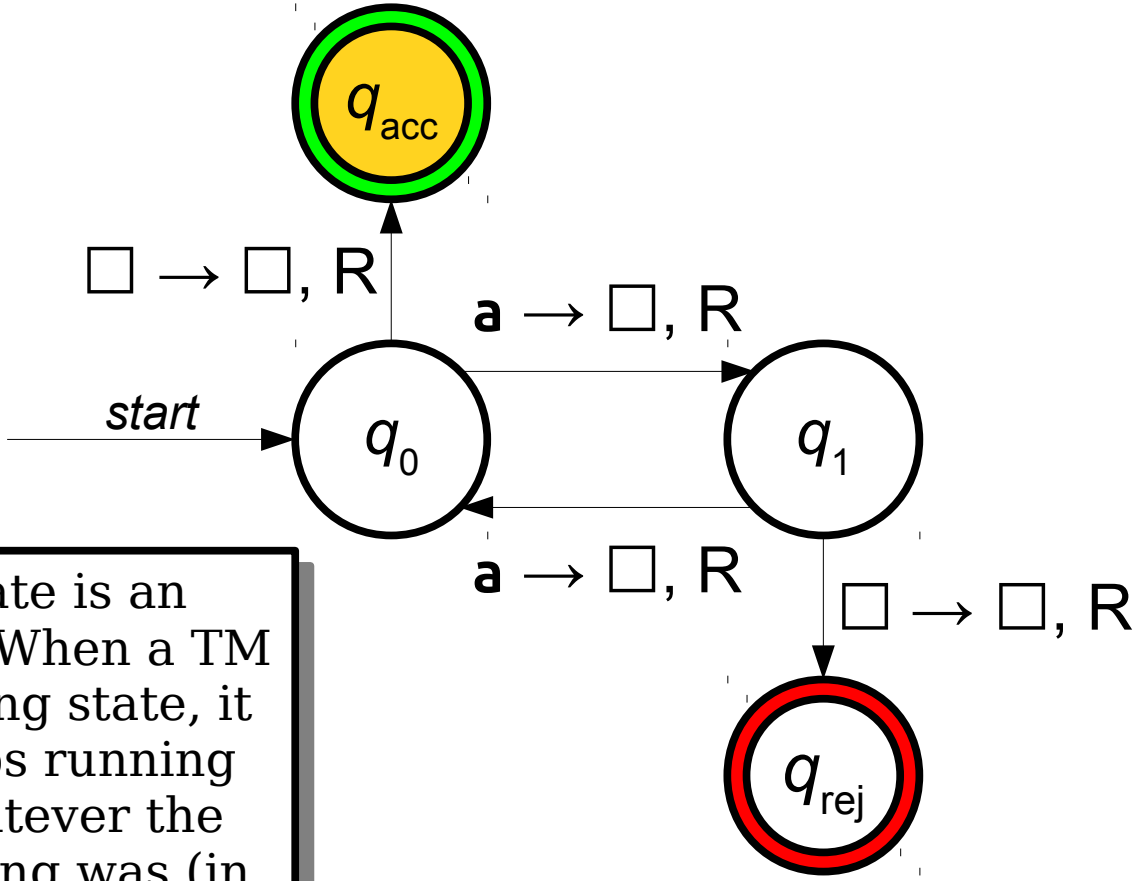


Our First Turing Machine



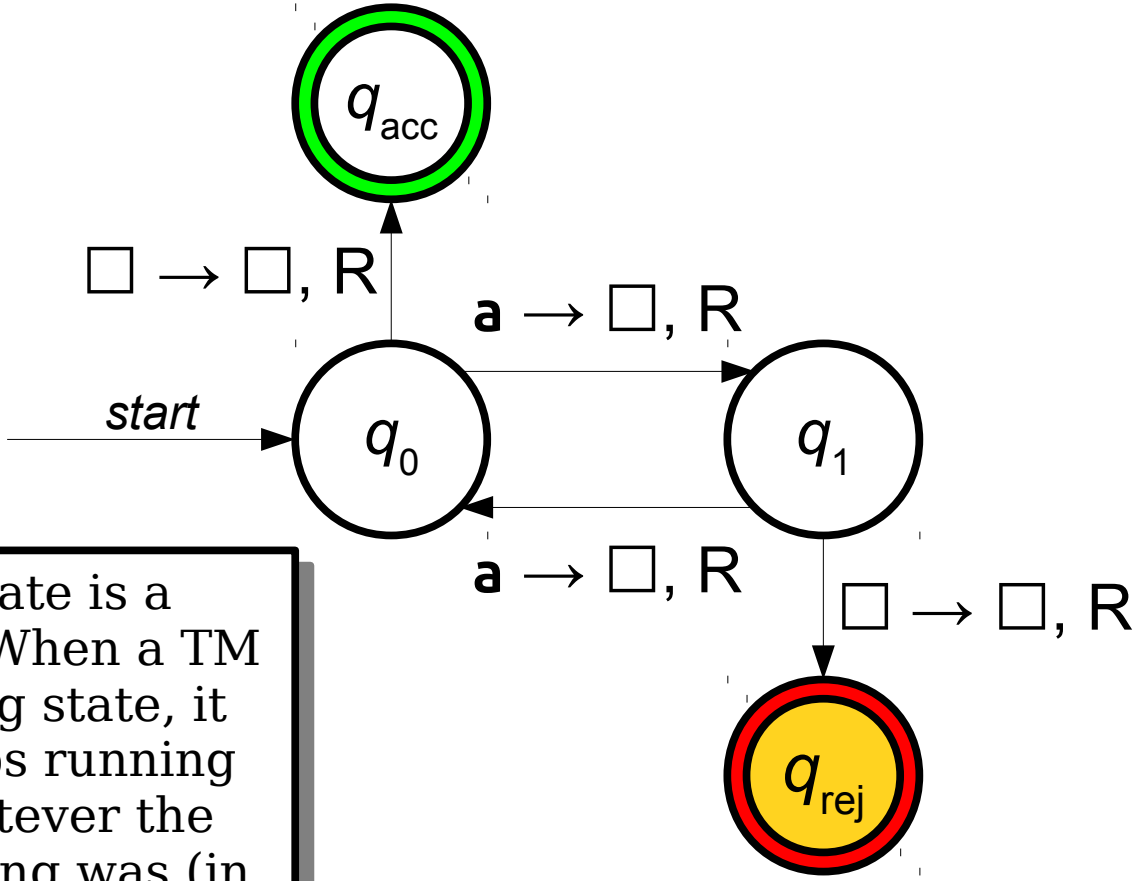
Unlike a DFA or NFA, a TM doesn't stop after reading all the input characters. We keep running until the machine explicitly says to stop.

Our First Turing Machine



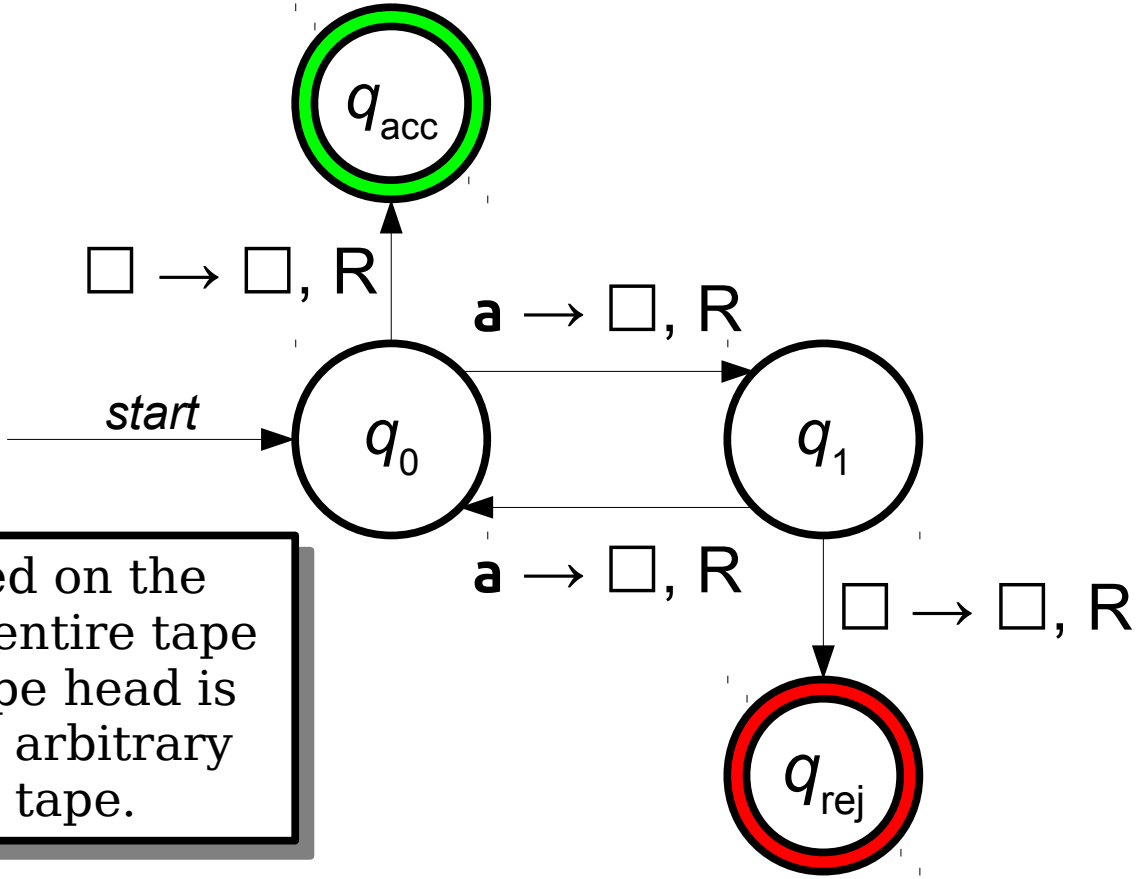
This special state is an ***accepting state***. When a TM enters an accepting state, it *immediately* stops running and accepts whatever the original input string was (in this case, **aaaa**).

Our First Turing Machine

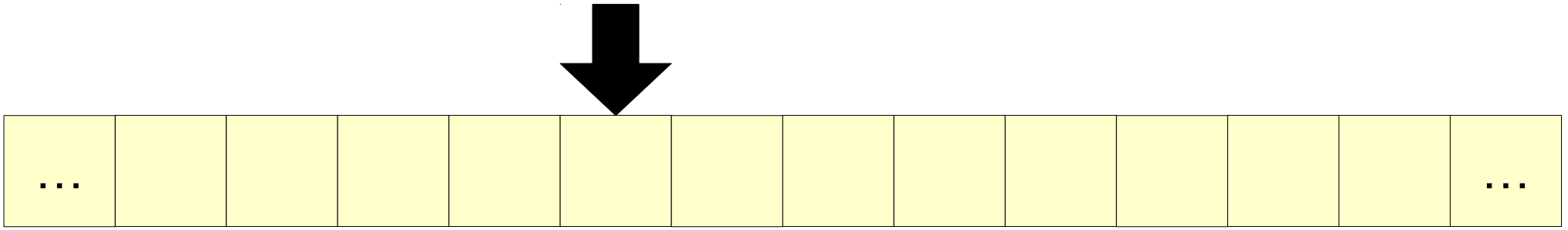


This special state is a ***rejecting state***. When a TM enters a rejecting state, it *immediately* stops running and rejects whatever the original input string was (in this case, **aaaaa**).

Our First Turing Machine

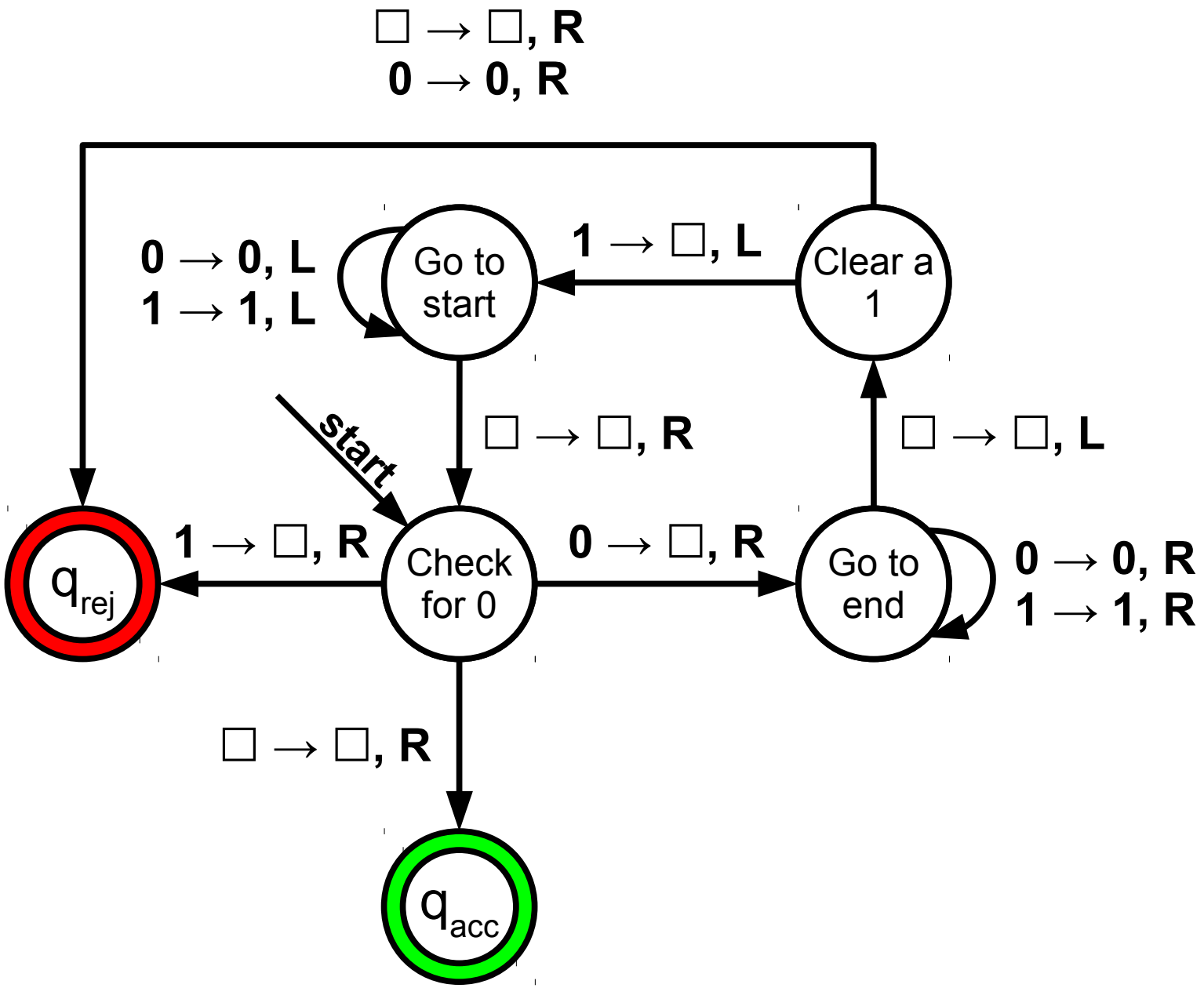


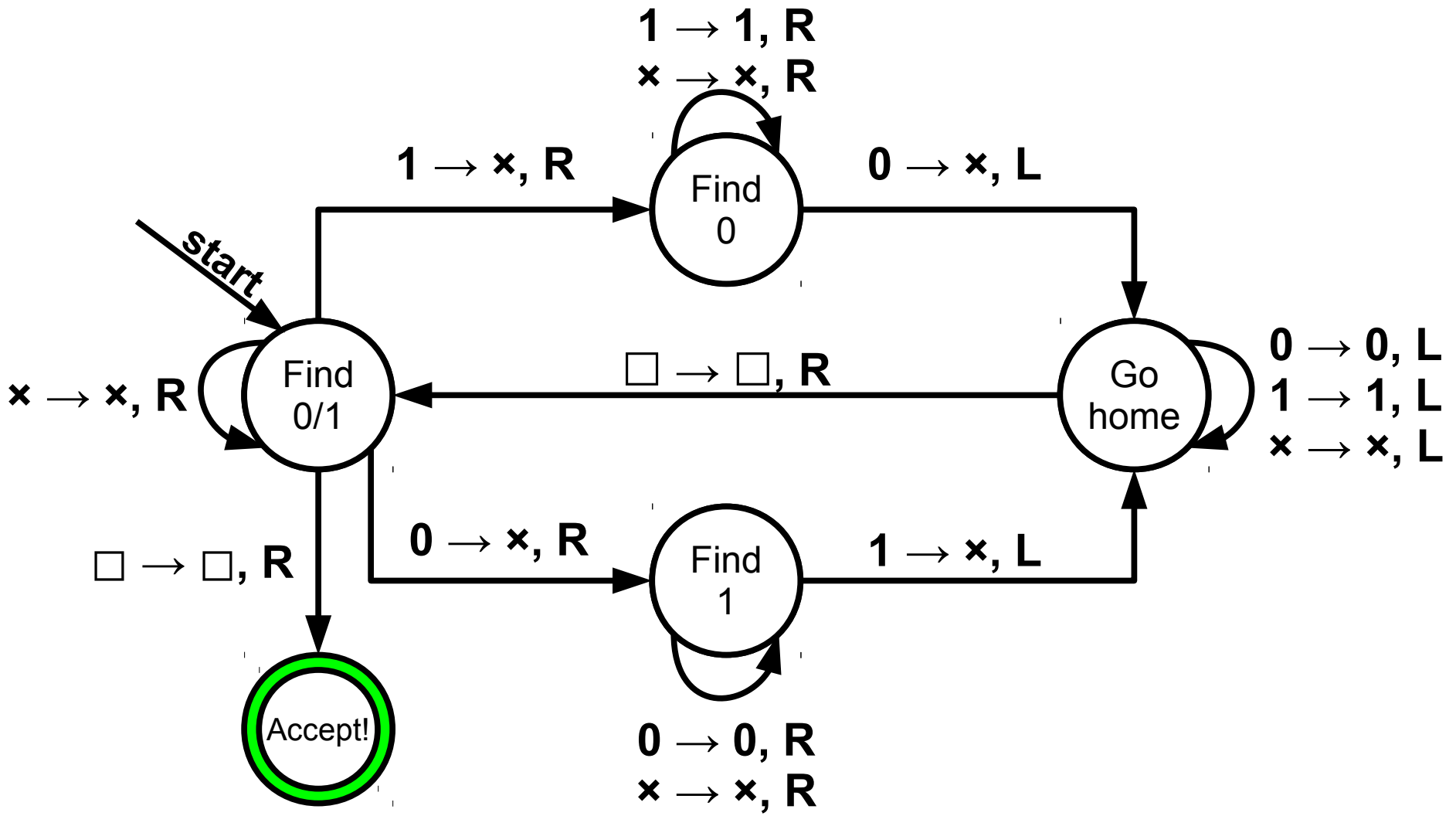
If the TM is started on the empty string ε, the entire tape is blank and the tape head is positioned at some arbitrary location on the tape.



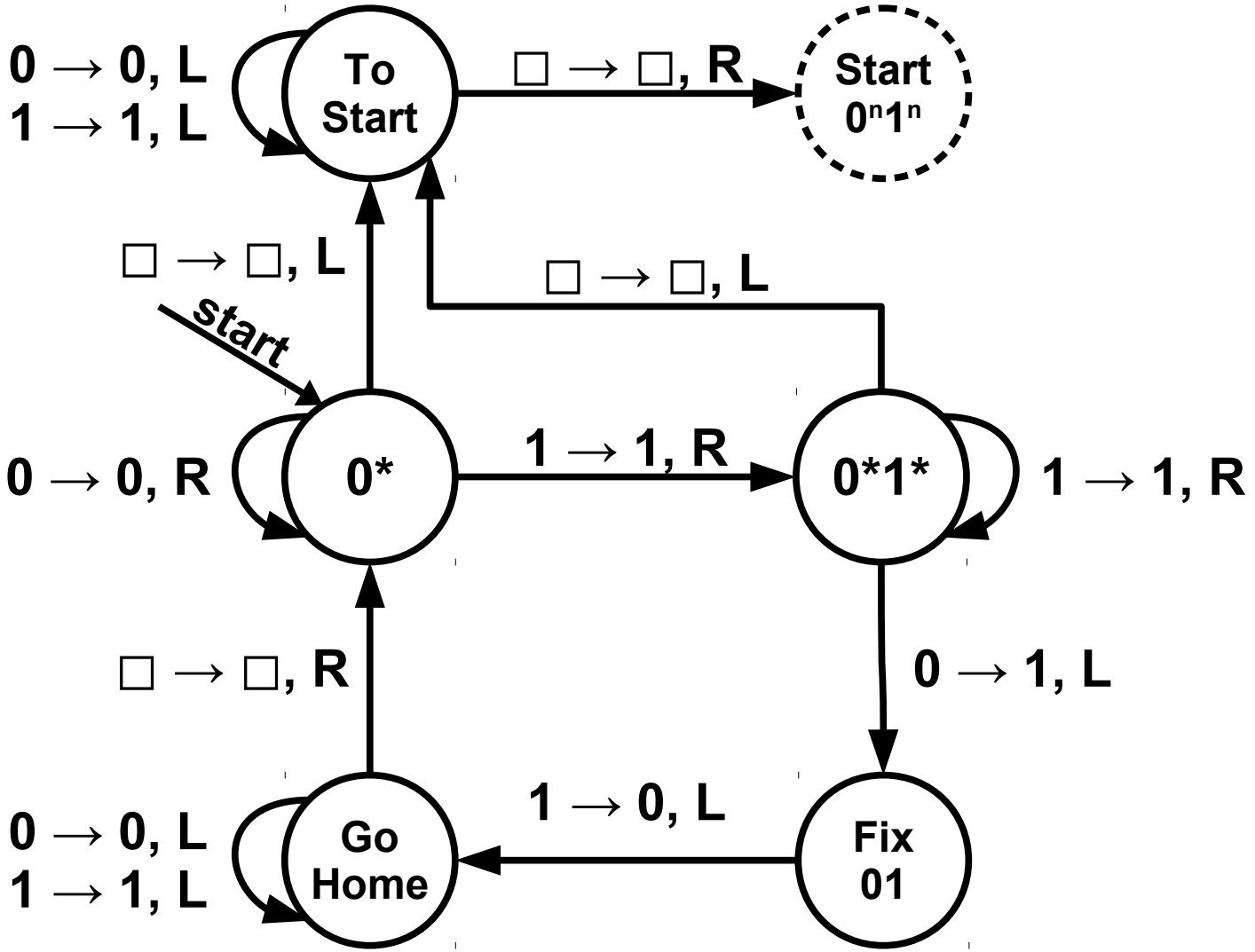
Input and Tape Alphabets

- A Turing machine has two alphabets:
  - An ***input alphabet*** Σ. All input strings are written in the input alphabet.
  - A ***tape alphabet*** Γ, where Σ ⊆ Γ. The tape alphabet contains all symbols that can be written onto the tape.
- The tape alphabet Γ can contain any number of symbols, but always contains at least one ***blank symbol***, denoted □. You are guaranteed □ ∉ Σ.
- At startup, the Turing machine begins with an infinite tape of □ symbols with the input written at some location. The tape head is positioned at the start of the input.


□ → □, R
0 → 0, R
0 → 0, L
1 → 1, L
Go to start
1 → □, L
Clear a 1
start
□ → □, R
□ → □, L
1 → □, R
Check for 0
0 → □, R
Go to end
0 → 0, R
1 → 1, R
q_rej
□ → □, R
q_acc



Remember that all missing transitions implicitly reject.


0 → 0, L
1 → 1, L
To Start
□ → □, R
Start
0n1n
□ → □, L
□ → □, L
start
0 → 0, R
0*
1 → 1, R
0*1*
1 → 1, R
□ → □, R
0 → 1, L
0 → 0, L
1 → 1, L
Go Home
1 → 0, L
Fix 01

New Stuff!

Main Question for Today:

Just how powerful are Turing machines?

TM Arithmetic

- Let's design a TM that, given a tape that looks like this:

…				1	3	7		4	2			…

ends up having the tape look like this:

…				1	7	9		0	0			…

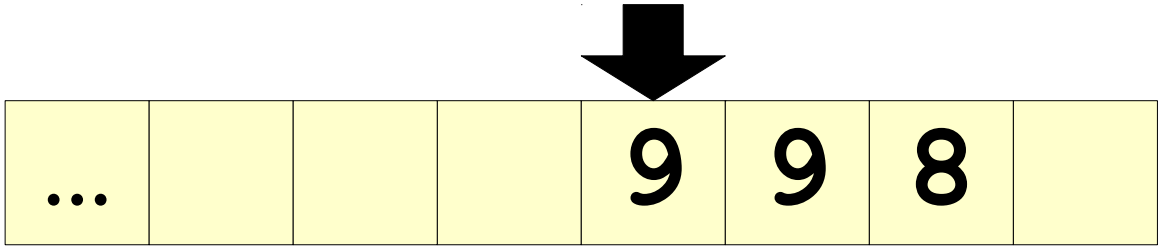
- In other words, we want to build a TM that can add two numbers.

TM Arithmetic

- There are many ways we could in principle design this TM.
- We're going to take the following approach:
  - First, we'll build a TM that increments a number.
  - Next, we'll build a TM that decrements a number.
  - Then, we'll combine them together, repeatedly decrementing the second number and adding one to the first number.

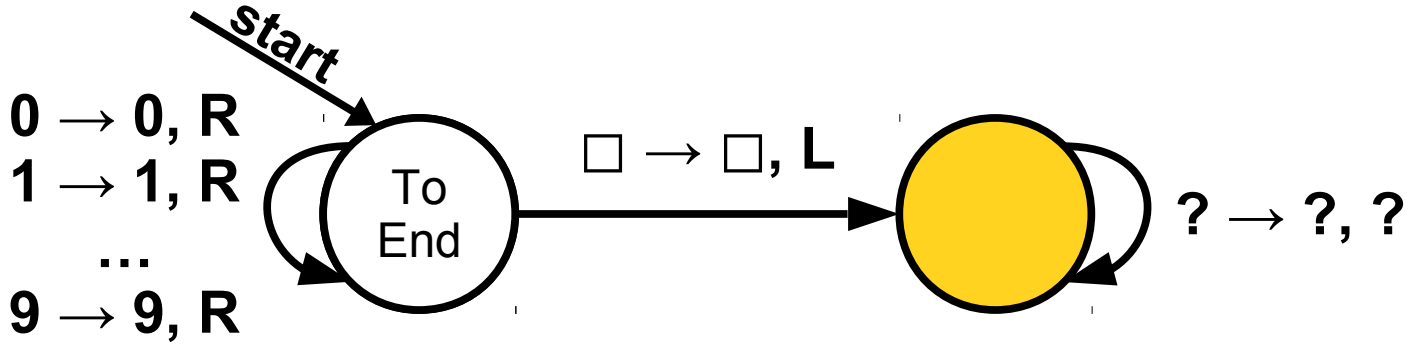
Incrementing Numbers

- Let's begin by building a TM that increments a number.
- We'll assume that
  - the tape head points at the start of a number,
  - there is are at least two blanks to the left of the number, and
  - that there's at least one blank at the start of the number.
- The tape head will end at the start of the number after incrementing it.



Incrementing Numbers

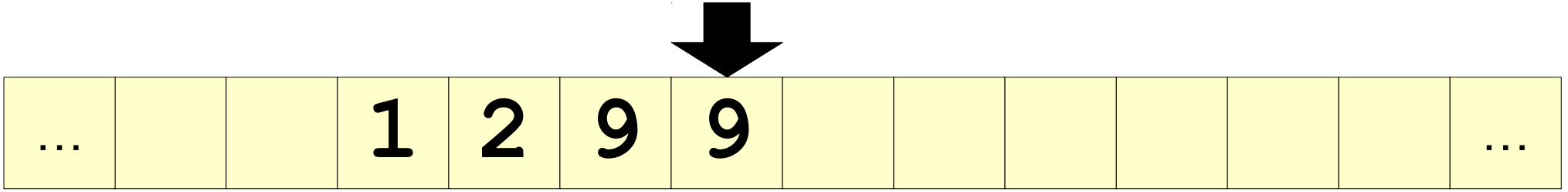
```
go to the end of the number;
while (the current digit is 9) {
    set the current digit to 0;
    back up one digit;
}
increment the current digit;
go to the start of the number;
```

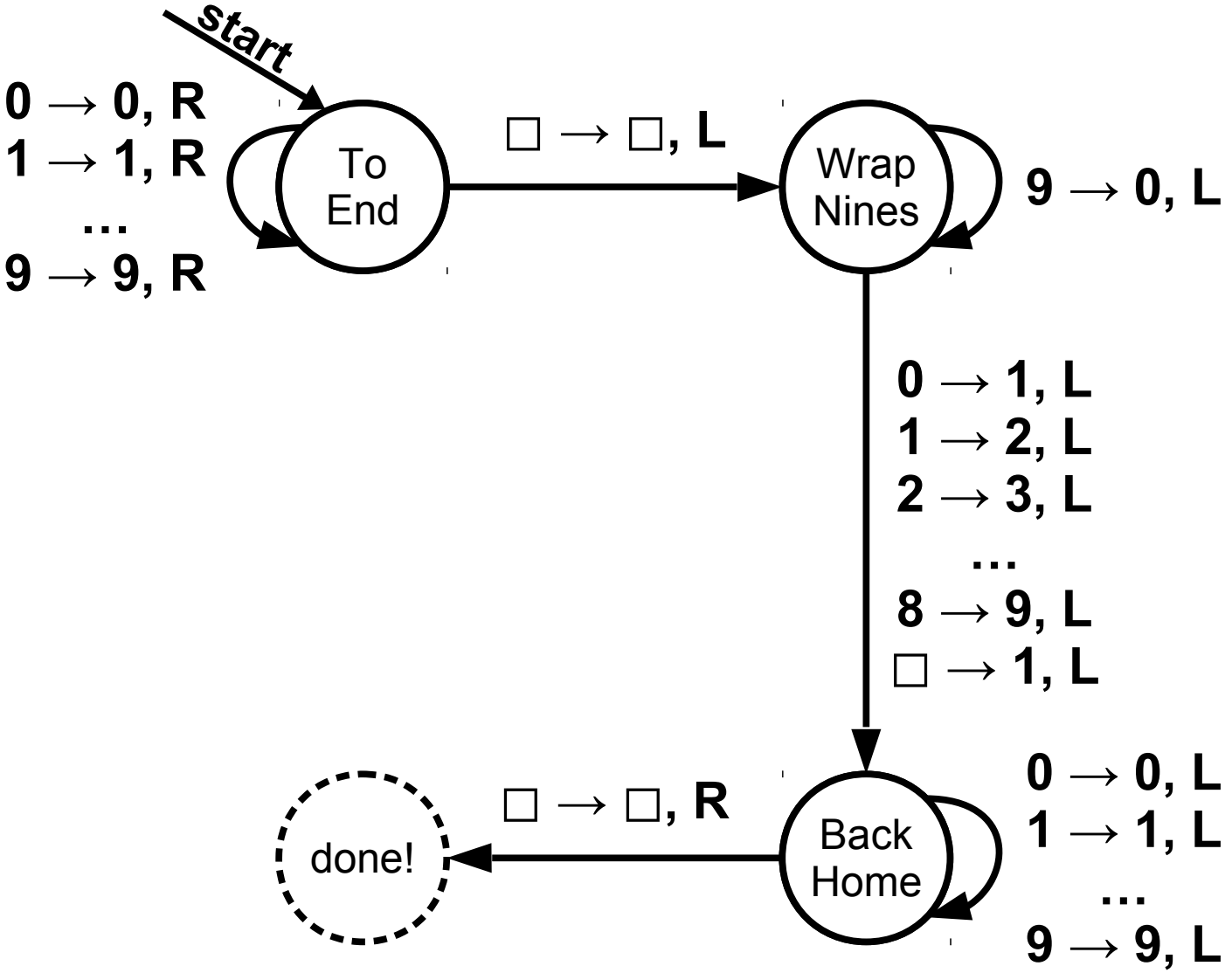


Based on we want this TM to do, what should this transition say?

A. **0** → **9**, R
B. **0** → **9**, L
C. **9** → **0**, R
D. **9** → **0**, L
E. None of these, or two or more of these.

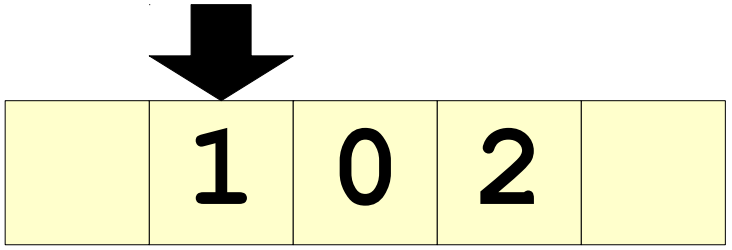
Answer at **PollEv.com/cs103** or
text **CS103** to **22333** once to join, then ***A***, ***B***, ***C***, ***D***, or ***E***.




start
0 → 0, R
1 → 1, R
…
9 → 9, R
To End
□ → □, L
Wrap Nines
9 → 0, L
0 → 1, L
1 → 2, L
2 → 3, L
…
8 → 9, L
□ → 1, L
Back Home
0 → 0, L
1 → 1, L
…
9 → 9, L
□ → □, R
done!

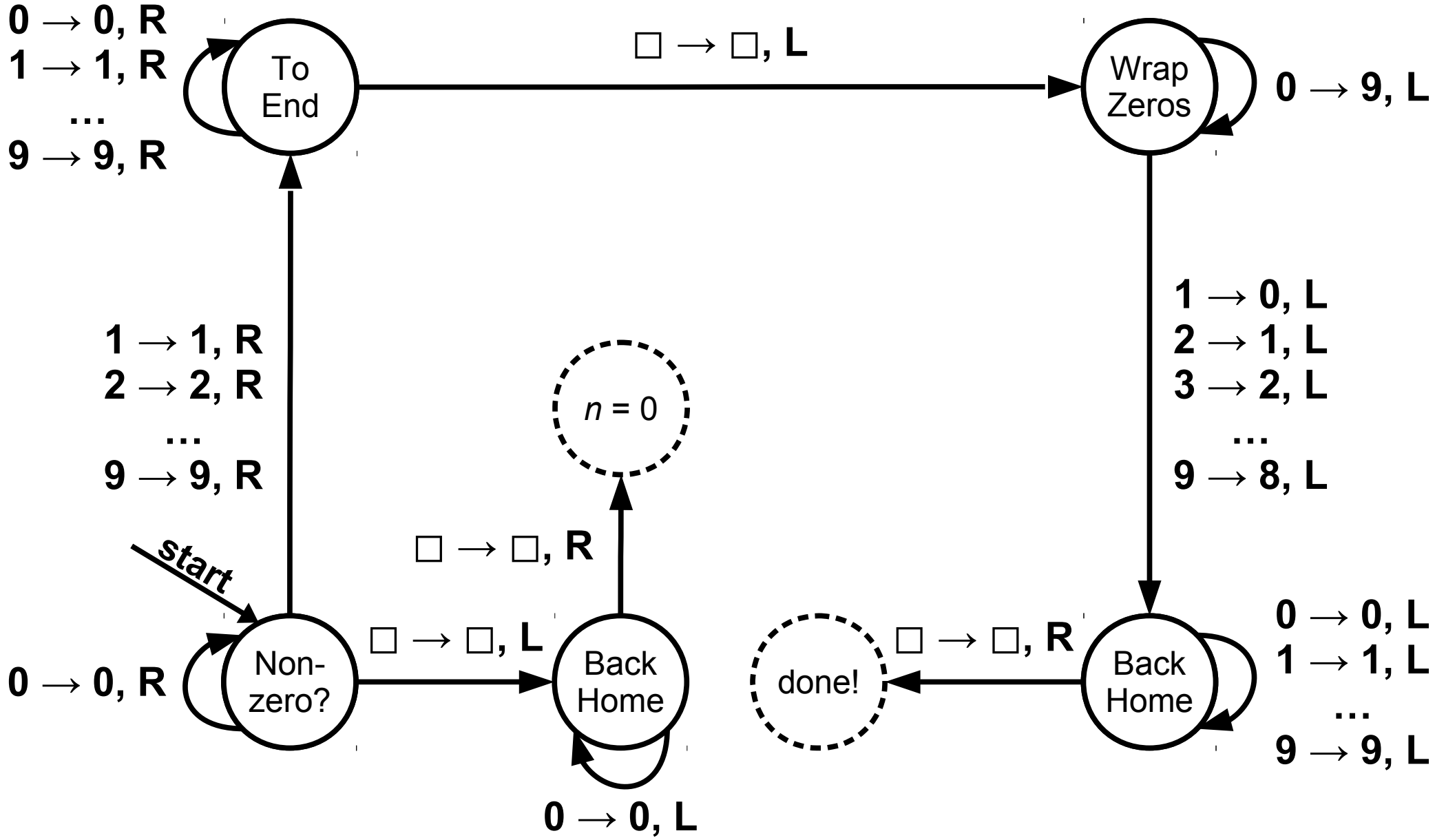
Decrementing Numbers

- Now, let's build a TM that decrements a number.
- We'll assume that
  - the tape head points at the start of a number,
  - there is at least one blank on each side of the number.
- The tape head will end at the start of the number after decrementing it.
- If the number is 0, then the subroutine should somehow signal this rather than making the number negative.



Decrementing Numbers

```
go to the end of the number;
if (every digit was 0) {
    signal that we're done;
}
while (the current digit is 0) {
    set the current digit to 9;
    back up one digit;
}
decrement the current digit;
go to the start of the number;
```

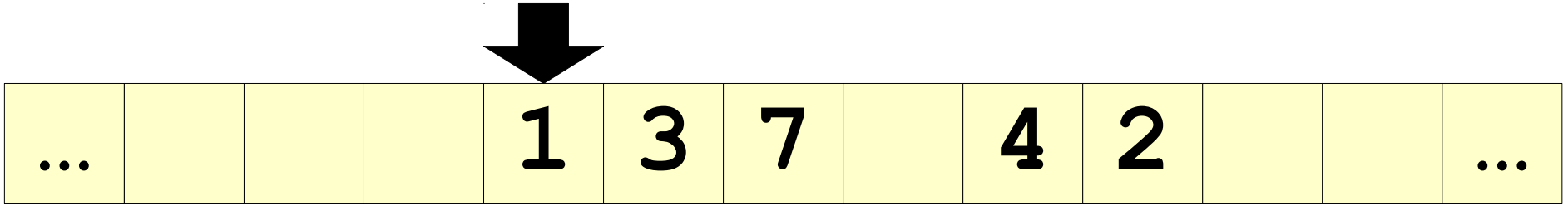

0 → 0, R
1 → 1, R
…
9 → 9, R
To End
□ → □, L
Wrap Zeros
0 → 9, L
1 → 0, L
2 → 1, L
3 → 2, L
…
9 → 8, L
1 → 1, R
2 → 2, R
…
9 → 9, R
n = 0
□ → □, R
start
Non-zero?
□ → □, L
Back Home
0 → 0, R
0 → 0, L
done!
□ → □, R
Back Home
0 → 0, L
1 → 1, L
…
9 → 9, L

TM Subroutines

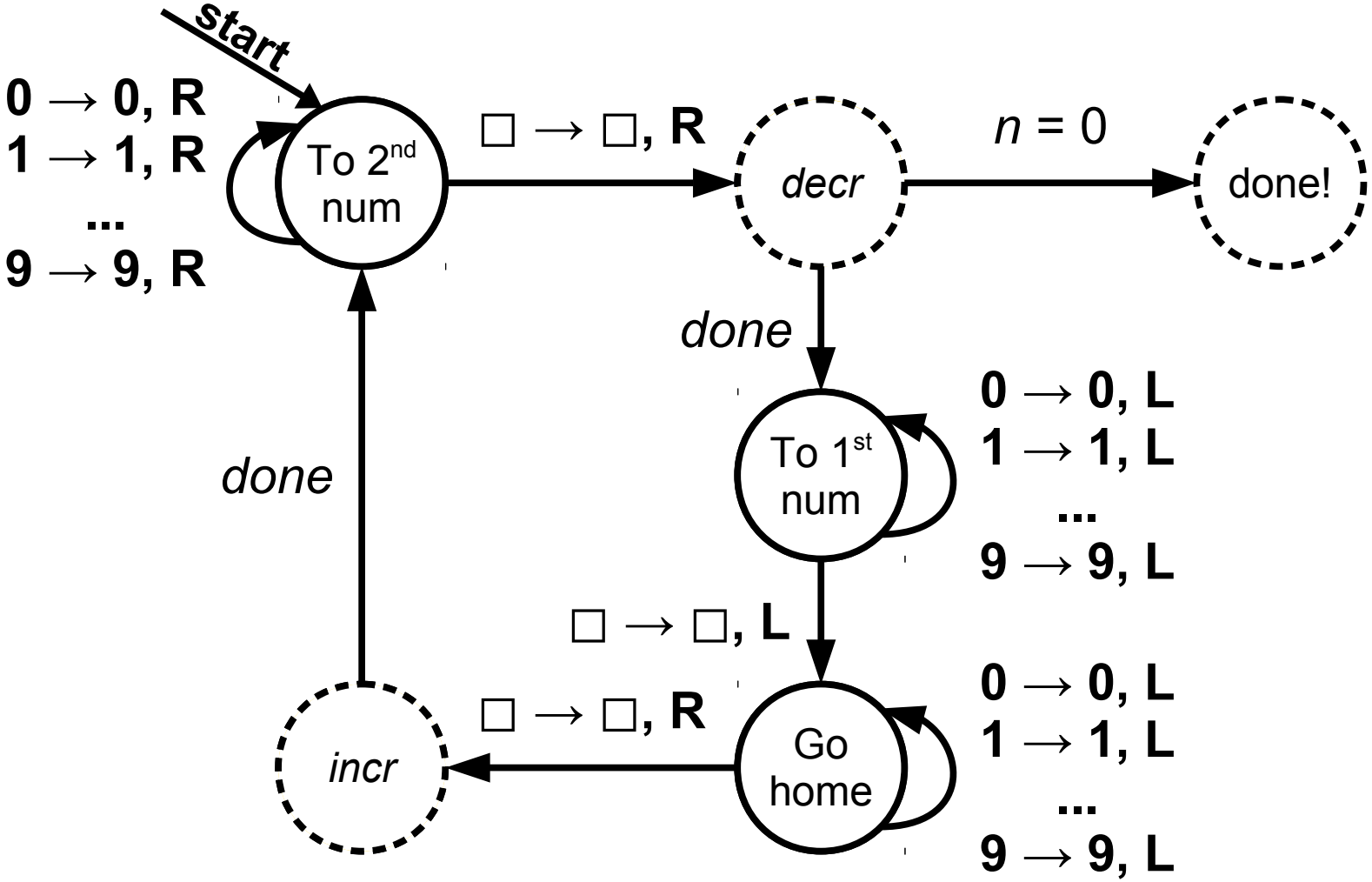
- Sometimes, a subroutine needs to report back some information about what happened.
- Just as a function can return multiple different values, we'll allow subroutines to have different “done” states.
- Each state can then be wired to a different state, so a TM using the subroutine can control what happens next.

Putting it All Together

- Our goal is to build a TM that, given two numbers, adds those numbers together.
- Before:

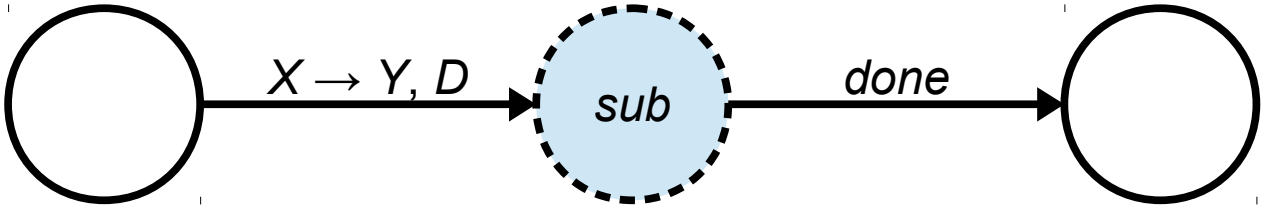


- After:


start
0 → 0, R
1 → 1, R
...
9 → 9, R
To 2nd num
□ → □, R
decr
n = 0
done!
done
To 1st num
0 → 0, L
1 → 1, L
...
9 → 9, L
□ → □, L
Go home
0 → 0, L
1 → 1, L
...
9 → 9, L
□ → □, R
incr
done

Using Subroutines

- Once you've built a subroutine, you can wire it into another TM with something that, schematically, looks like this:



- Intuitively, this corresponds to transitioning to the start state of the subroutine, then replacing the “done” state of the subroutine with the state at the end of the transition.

Time-Out for Announcements!

Problem Sets

- Problem Set Seven was due at 2:30PM today.
  - Using late days, you can extend the deadline to this Sunday at 2:30PM.
- Problem Set Eight goes out today. It's due the Friday at 2:30PM.
  - Play around with CFGs and Turing machines!
  - We have online tools you'll use to design, test, and submit your grammars and TMs. Hope this helps!
  - Depending on what we manage to get through today, there's a chance that a few topics on the problem set will require content from Monday's lecture. Those problems are clearly marked as such.

Extra Practice Problems 3

- Later this evening, we'll post a (massive!) set of practice problems (EPP3) on the course website, with solutions.
- These problems cover all the topics we'll explore this quarter. Questions involving topics we will cover next week are marked with a star.
- Feel free to work through any of the problems that seem interesting and to ask questions!

Back to CS103!

Main Question for Today:

Just how powerful are Turing machines?

How Powerful are TMs?

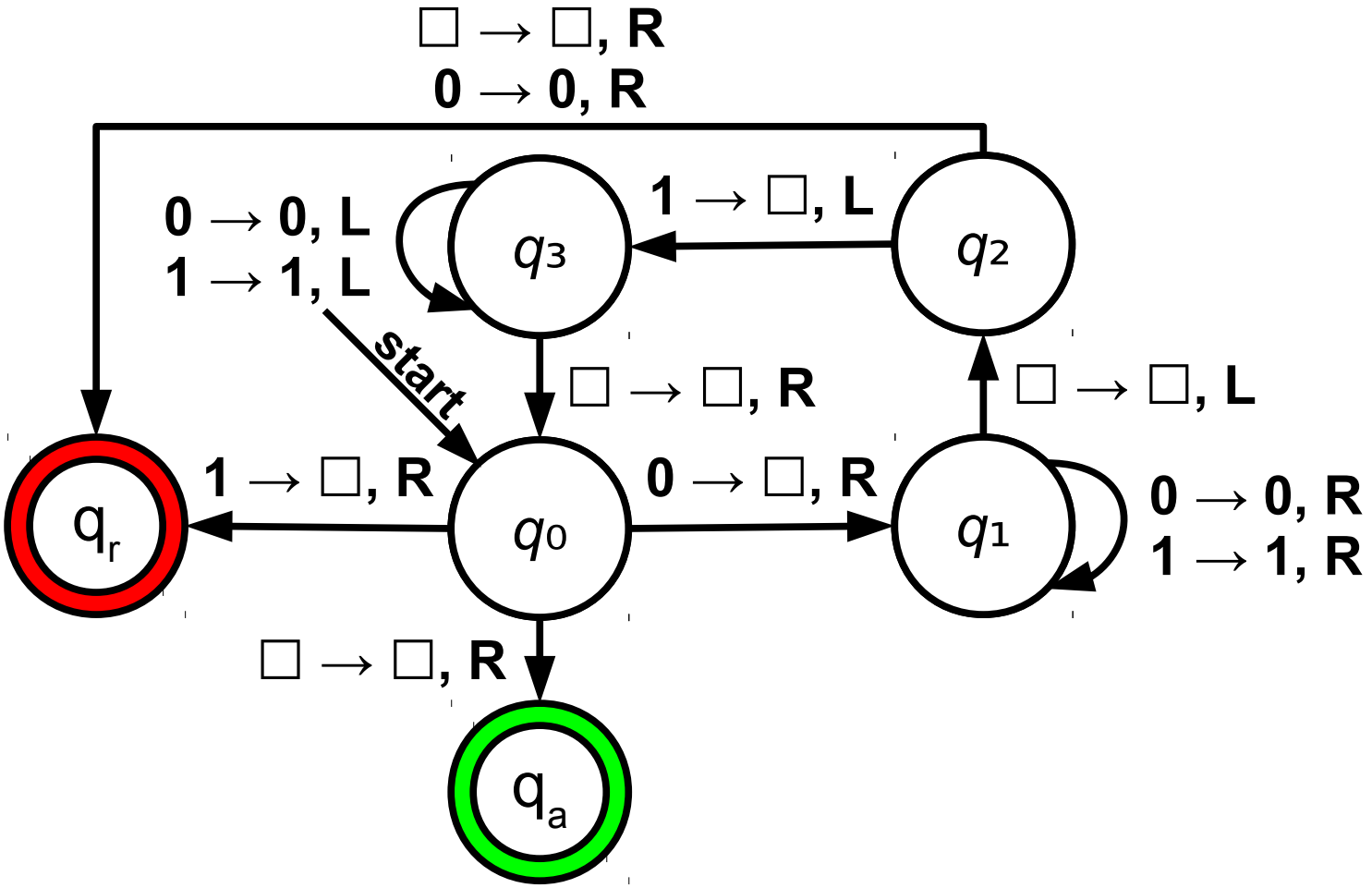
- Regular languages, intuitively, are as powerful as computers with finite memory.
- TMs by themselves seem like they can do a fair number of tasks, but it's unclear specifically what they can do.
- Let's explore their expressive power.

Real and “Ideal” Computers

- A real computer has memory limitations: you have a finite amount of RAM, a finite amount of disk space, etc.
- However, as computers get more and more powerful, the amount of memory available keeps increasing.
- An ***idealized computer*** is like a regular computer, but with unlimited RAM and disk space. It functions just like a regular computer, but never runs out of memory.

Claim 1: Idealized computers can simulate Turing machines.

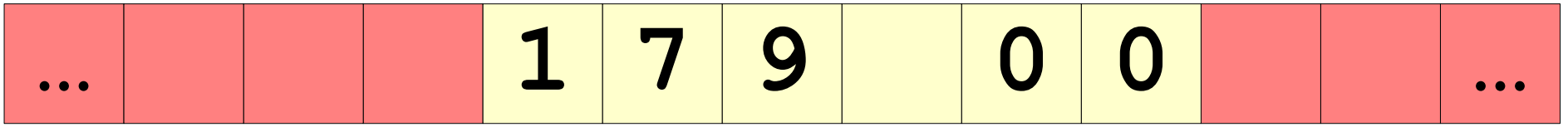
"Anything that can be done with a TM can also be done with an unbounded-memory computer."



	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

Simulating a TM

- To simulate a TM, the computer would need to be able to keep track of
  - the finite-state control,
  - the current state,
  - the position of the tape head, and
  - the tape contents.
- The tape contents are infinite, but that's because there are infinitely many blanks on both sides.
- We only need to store the “interesting” part of the tape (the parts that have been read from or written to so far.)



Claim 2: Turing machines can simulate idealized computers.

"Anything that can be done with an unbounded-memory computer can be done with a TM."

What We've Seen

- TMs can
  - implement loops (basically, every TM we've seen).
  - make function calls (subroutines).
  - keep track of natural numbers (written in unary or in decimal on the tape).
  - perform elementary arithmetic (equality testing, multiplication, addition, increment, decrement, etc.).
  - perform if/else tests (different transitions based on different cases).

What Else Can TMs Do?

- Maintain variables.
  - Have a dedicated part of the tape where the variables are stored.
  - We've seen this before: take a look at our machine for increment/decrement.
- Maintain arrays and linked structures.
  - Divide the tape into different regions corresponding to memory locations.
  - Represent arrays and linked structures by keeping track of the ID of one of those regions.

A CS107 Perspective

- Internally, computers execute by using basic operations like
  - simple arithmetic,
  - memory reads and writes,
  - branches and jumps,
  - register operations,
  - etc.
- Each of these are simple enough that they could be simulated by a Turing machine.

A Leap of Faith

- It may require a leap of faith, but anything you can do a computer (excluding randomness and user input) can be performed by a Turing machine.
- The resulting TM might be colossal, or really slow, or both, but it would still faithfully simulate the computer.
- We're going to take this as an article of faith in CS103. If you curious for more specific details of how to do this, come talk to me after class.

This is quite a bold claim to make. What's an example of something your computer can do that you're convinced a Turing machine couldn't do?

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

Just how powerful ***are*** Turing machines?

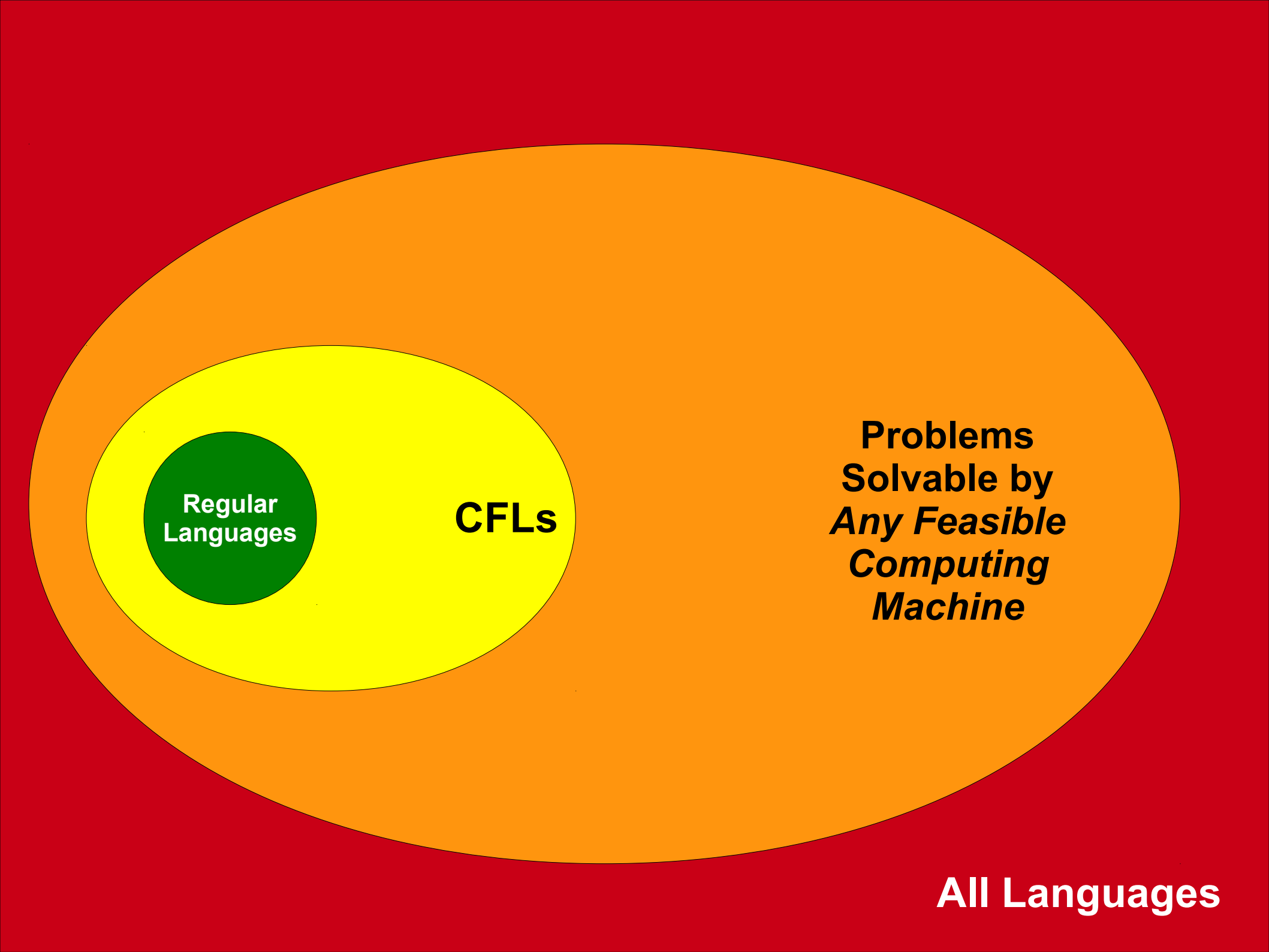
Effective Computation

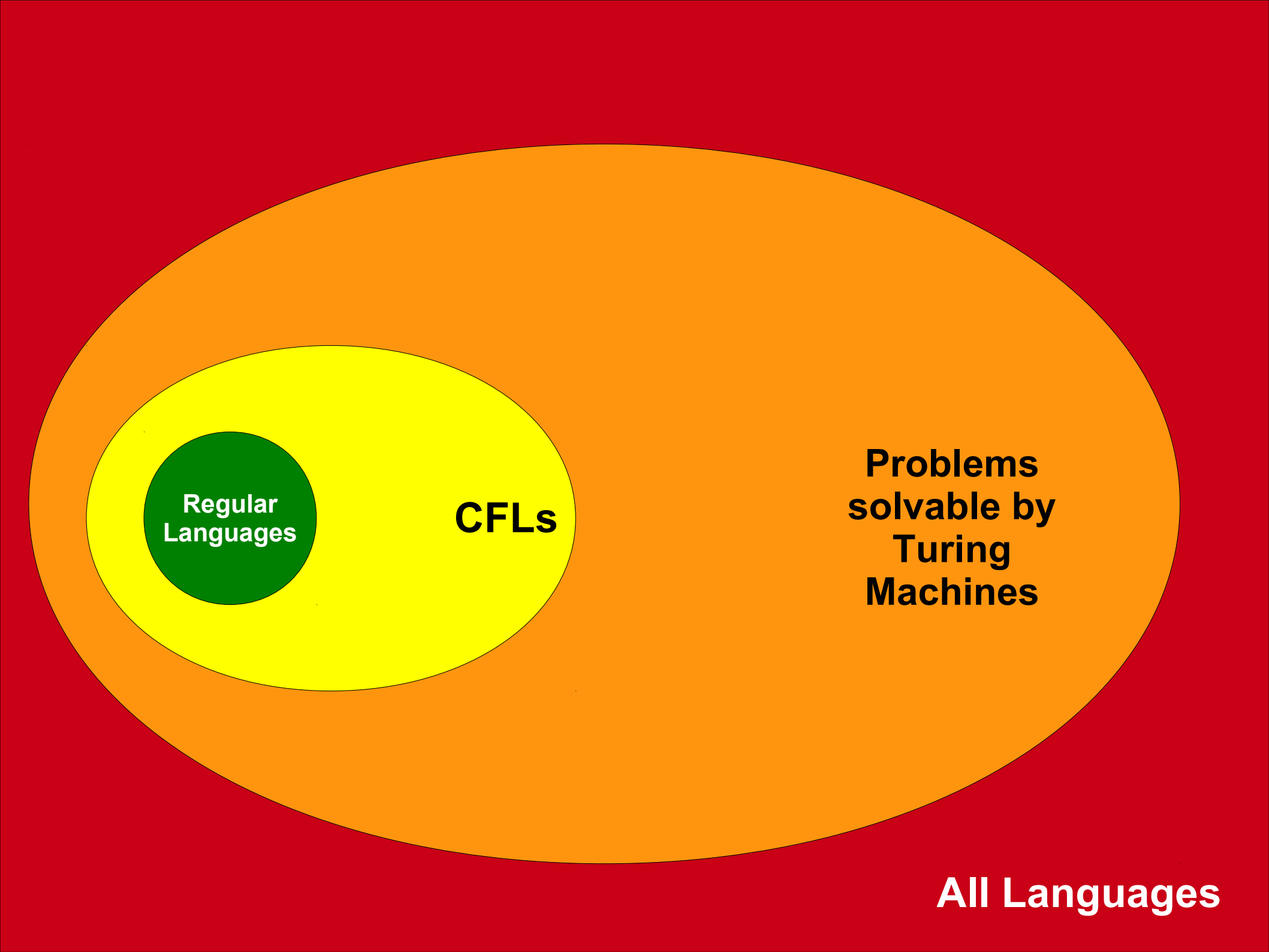
- An ***effective method of computation*** is a form of computation with the following properties:
  - The computation consists of a set of steps.
  - There are fixed rules governing how one step leads to the next.
  - Any computation that yields an answer does so in finitely many steps.
  - Any computation that yields an answer always yields the correct answer.
- This is not a formal definition. Rather, it's a set of properties we expect out of a computational system.

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
CFLs
Problems Solvable by *Any Feasible Computing Machine*
All Languages

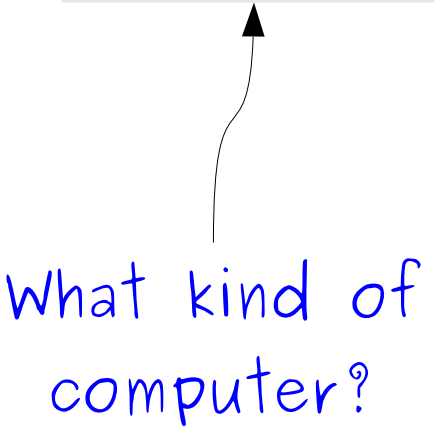

Regular Languages
CFLs
Problems solvable by Turing Machines
All Languages

TMs ≈ Computers

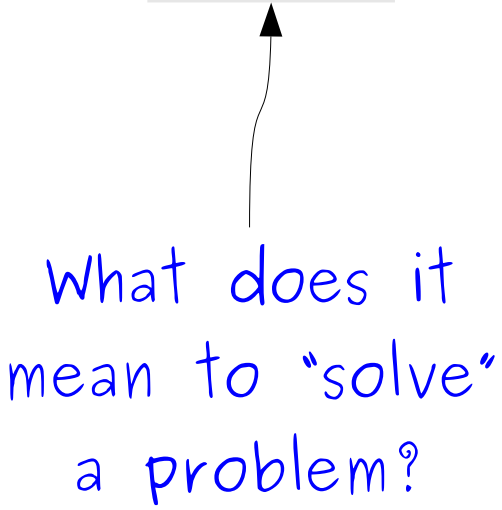
- Because Turing machines have the same computational powers as regular computers, we can (essentially) reason about Turing machines by reasoning about actual computer programs.
- Going forward, we're going to switch back and forth between TMs and computer programs based on whatever is most appropriate.
- In fact, our eventual proofs about the existence of impossible problems will involve a good amount of pseudocode. Stay tuned for details!

Strings, Languages, Encodings, and Problems

What problems can we solve with a computer?

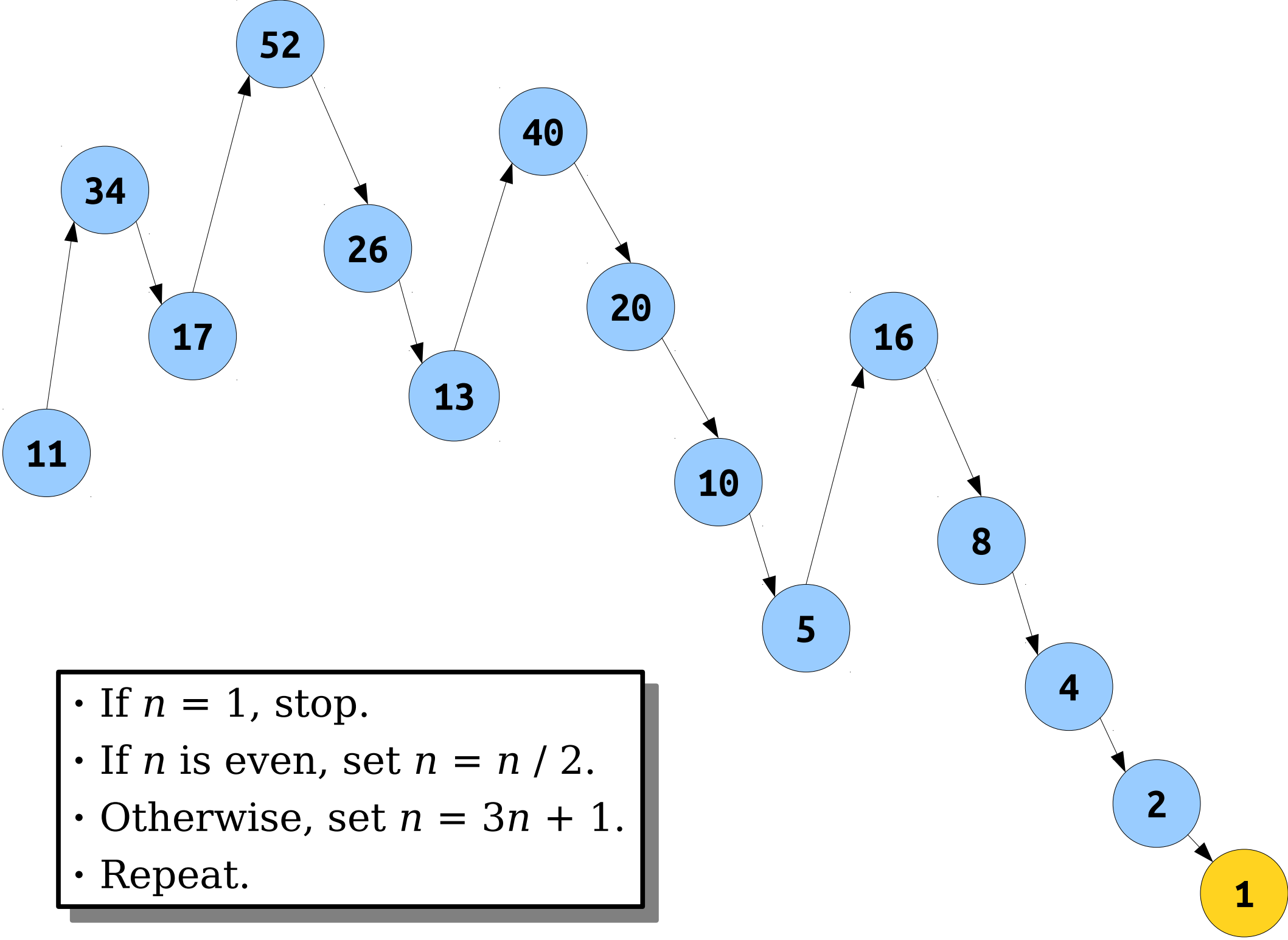


What problems can we solve with a computer?



The Hailstone Sequence

- Consider the following procedure, starting with some $n \in \mathbb{N}$, where $n > 0$:
  - If $n = 1$, you are done.
  - If n is even, set $n = n / 2$.
  - Otherwise, set $n = 3n + 1$.
  - Repeat.
- ***Question:*** Given a number n, does this process terminate?



- If $n = 1$, stop.
- If n is even, set $n = n / 2$.
- Otherwise, set $n = 3n + 1$.
- Repeat.

The Hailstone Sequence

- Let $\Sigma = \{\mathtt{1}\}$ and consider the language

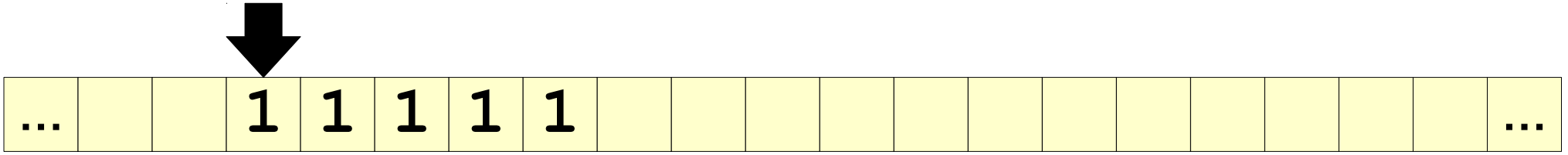
  $L = \{ \mathtt{1}^n \mid n > 0$ and the hailstone sequence terminates for $n \}$.

- Could we build a TM for L?

The Hailstone Turing Machine

- We can build a TM that works as follows:
  - If the input is ε, reject.
  - While the string is not **1**:
    - If the input has even length, halve the length of the string.
    - If the input has odd length, triple the length of the string and append a **1**.
  - Accept.

The Hailstone Turing Machine



If the input is ε, reject.

While the input is not 1:

- If the input has even length, halve the length of the string.
- If the input has odd length, triple the length of the string and append a 1.

Accept.

Does this Turing machine accept all nonempty strings?

The Collatz Conjecture

- It is *unknown* whether this process will terminate for all natural numbers.
- In other words, ***no one knows whether the TM described in the previous slides will always stop running!***
- The conjecture (unproven claim) that this always terminates is called the ***Collatz Conjecture***.

The Collatz Conjecture

“Mathematics may not be ready for such problems.” - Paul Erdős

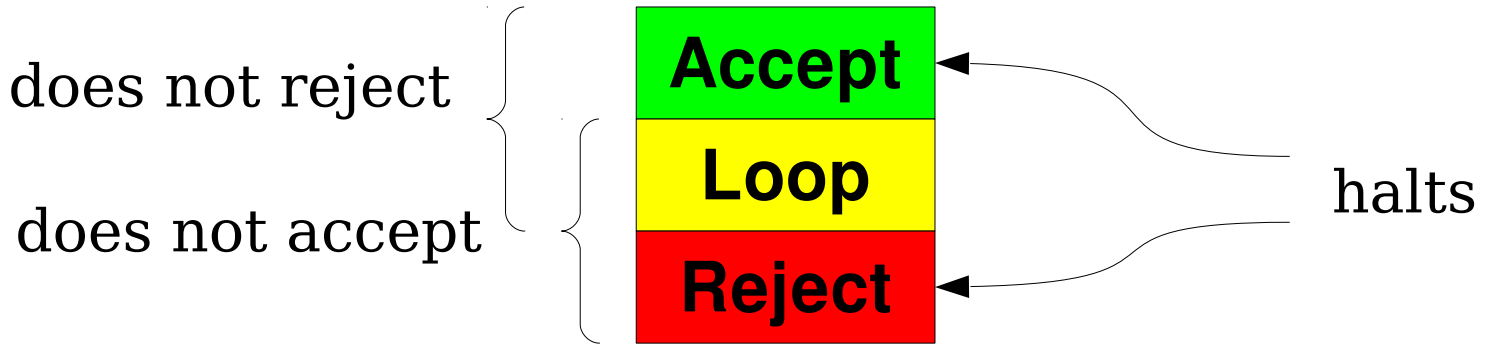
- The fact that the Collatz Conjecture is unresolved is useful later on for building intuitions. Keep this in mind!

An Important Observation

- Unlike finite automata, which automatically halt after all the input is read, TMs keep running until they explicitly enter an accept or reject state.
- As a result, it’s possible for a TM to run forever without accepting or rejecting.
- This leads to several important questions:
  - How do we formally define what it means to build a TM for a language?
  - What implications does this have about problem-solving?

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.
- M ***does not accept w*** if it either rejects w or loops infinitely 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.



The Language of a TM

- The language of a Turing machine M, denoted $\mathscr{L}(M)$, is the set of all strings that M accepts:

$$\mathscr{L}(M) = \{ w \in \Sigma^* \mid M \text{ accepts } w \}$$

Let M be a TM with input alphabet Σ and let $w \in \Sigma^*$ be a string where $w \notin \mathscr{L}(M)$. How many of the following are ***not*** possible?

M accepts w.
M rejects w.
M loops on w.
M does not accept w.
M does not reject w.
M halts on w.

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

The Language of a TM

- The language of a Turing machine M, denoted $\mathscr{L}(M)$, is the set of all strings that M accepts:

$$\mathscr{L}(M) = \{ w \in \Sigma^* \mid M \text{ accepts } w \}$$

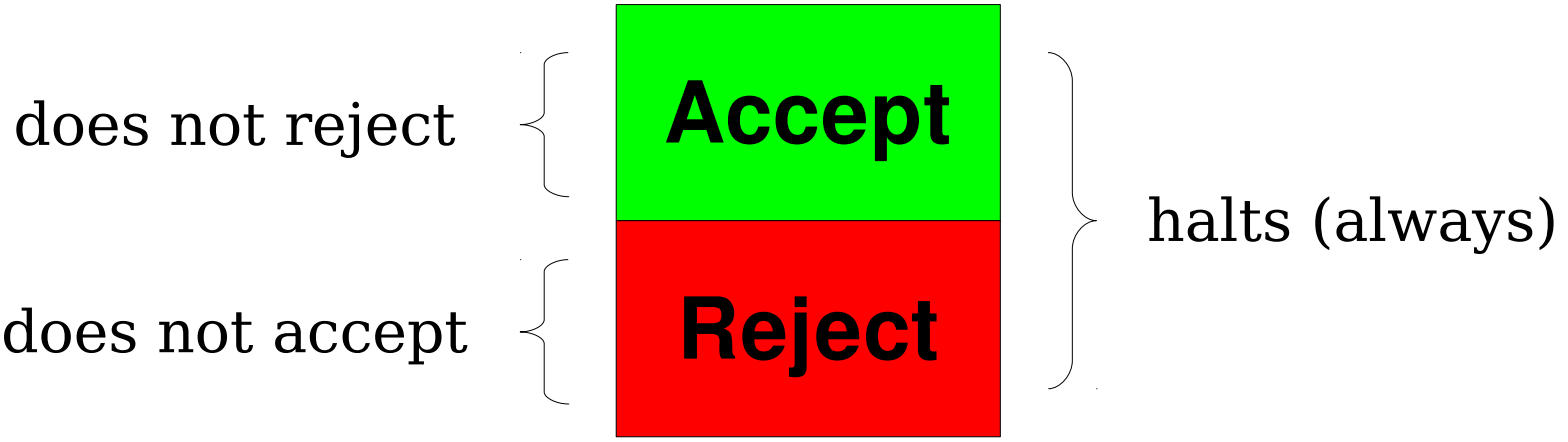
- For any $w \in \mathscr{L}(M)$, M accepts w.
- For any $w \notin \mathscr{L}(M)$, M does not accept w.
  - M might reject, or it might loop forever.
- A language is called ***recognizable*** if it is the language of some TM.
- A TM M where $\mathscr{L}(M) = L$ is called a ***recognizer*** for L.
- Notation: the class **RE** is the set of all recognizable languages.

$$L \in \mathbf{RE} \quad \leftrightarrow \quad L \text{ is recognizable}$$

What do you think? Does that correspond to what you think it means to solve a problem?

Deciders

- Some Turing machines always halt; they never go into an infinite loop.
- If M is a TM and M halts on every possible input, then we say that M is a ***decider***.
- For deciders, accepting is the same as not rejecting and rejecting is the same as not accepting.



Decidable Languages

- A language L is called ***decidable*** if there is a decider M such that $\mathscr{L}(M) = L$.
- Equivalently, a language L is decidable if there is a TM M such that
  - If $w \in L$, then M accepts w.
  - If $w \notin L$, then M rejects w.
- The class **R** is the set of all decidable languages.

$$L \in \mathbf{R} \quad \leftrightarrow \quad L \text{ is decidable}$$

Examples of **R** Languages

- All regular languages are in **R**.
  - See Problem Set Eight!
- $\{ \texttt{0}^n\texttt{1}^n \mid n \in \mathbb{N} \}$ is in **R**.
  - The TM we built is a decider.
- All CFLs are in **R**.
  - Proof is tricky; check Sipser for details.
  - (This is why it's possible to build the CFG tool online!)

Why **R** Matters

- If a language is in **R**, there is an algorithm that can decide membership in that language.
  - Run the decider and see what it says.
- If there is an algorithm that can decide membership in a language, that language is in **R**.
  - By the Church-Turing thesis, any effective model of computation is equivalent in power to a Turing machine.
  - Therefore, if there is *any* algorithm for deciding membership in the language, there is a decider for it.
  - Therefore, the language is in **R**.
- ***A language is in R if and only if there is an algorithm for deciding membership in that language.***

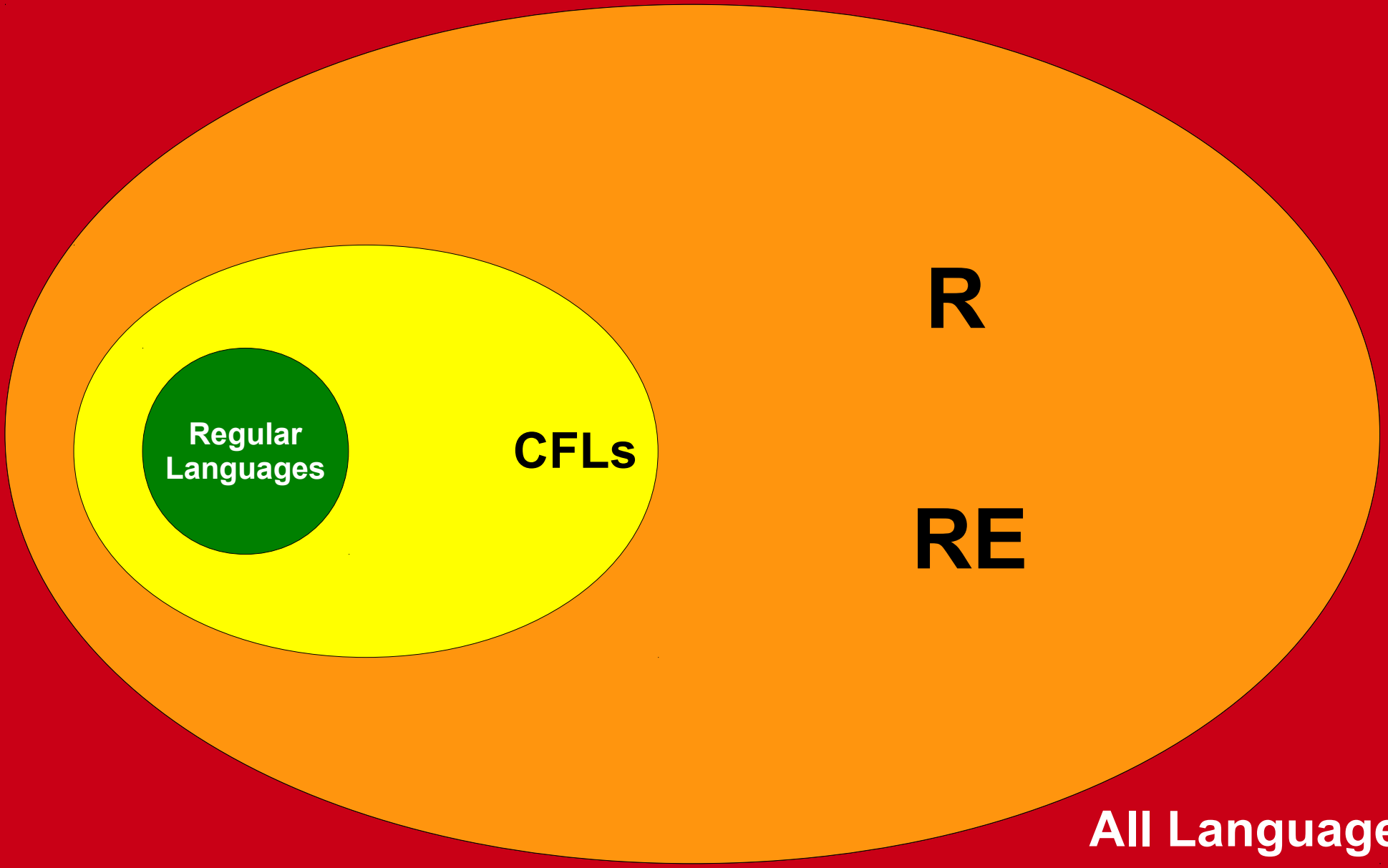
R and **RE** Languages

- Every decider is a Turing machine, but not every Turing machine is a decider.
- This means that $\mathbf{R} \subseteq \mathbf{RE}$.
- Hugely important theoretical question:

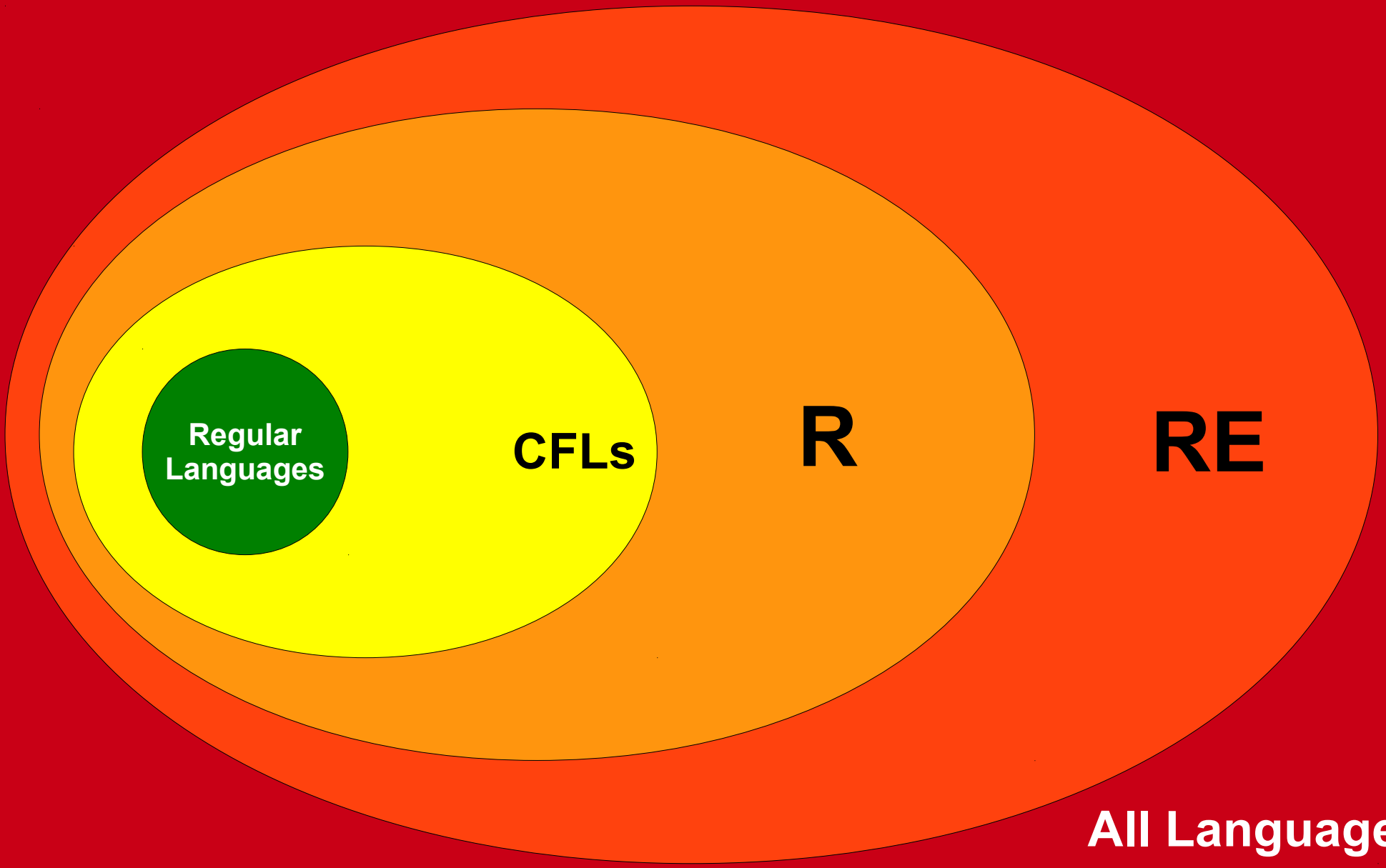
$$\mathbf{R} \stackrel{?}{=} \mathbf{RE}$$

- That is, if you can just confirm "yes" answers to a problem, can you necessarily *solve* that problem?

Which Picture is Correct?



Which Picture is Correct?



Unanswered Questions

- Why exactly is **RE** an interesting class of problems?
- What does the $\mathbf{R} \stackrel{?}{=} \mathbf{RE}$ question mean?
- Is $\mathbf{R} = \mathbf{RE}$?
- What lies beyond **R** and **RE**?
- We'll see the answers to each of these in due time.