

Turing Machines

Part Two

Recap from Last Time

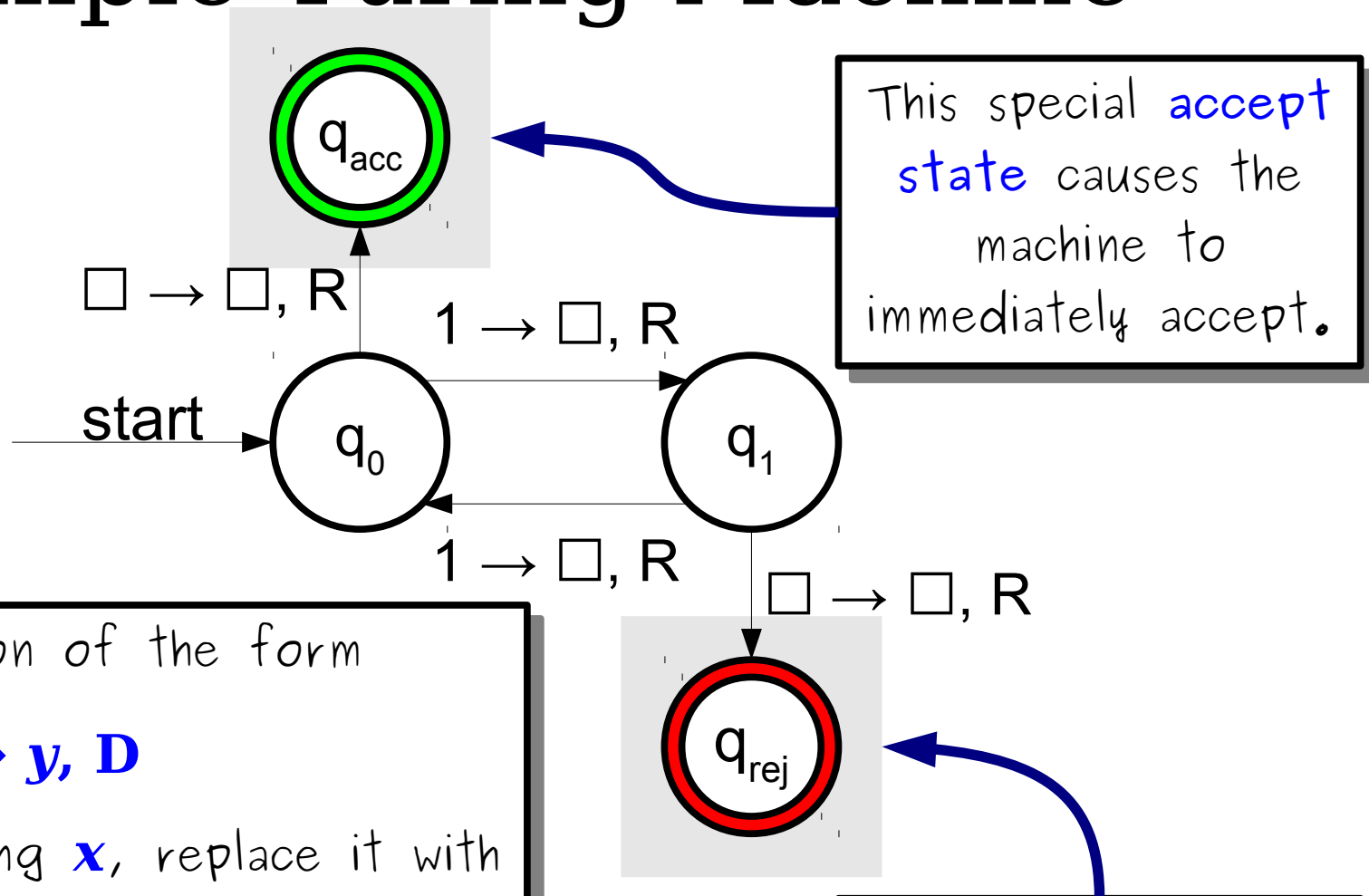
The Turing Machine

- A Turing machine consists of three parts:
 - A ***finite-state control*** that issues commands,
 - an ***infinite tape*** for input and scratch space, and
 - a ***tape head*** that can read and write a single tape cell.
- At each step, the Turing machine
 - writes a symbol to the tape cell under the tape head,
 - changes state, and
 - moves the tape head to the left or to the right.

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 $\Sigma \subseteq \Gamma$. 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 \square . You are guaranteed $\square \notin \Sigma$.
- At startup, the Turing machine begins with an infinite tape of \square symbols with the input written at some location. The tape head is positioned at the start of the input.

A Simple Turing Machine



This special **accept state** causes the machine to immediately accept.

Each transition of the form

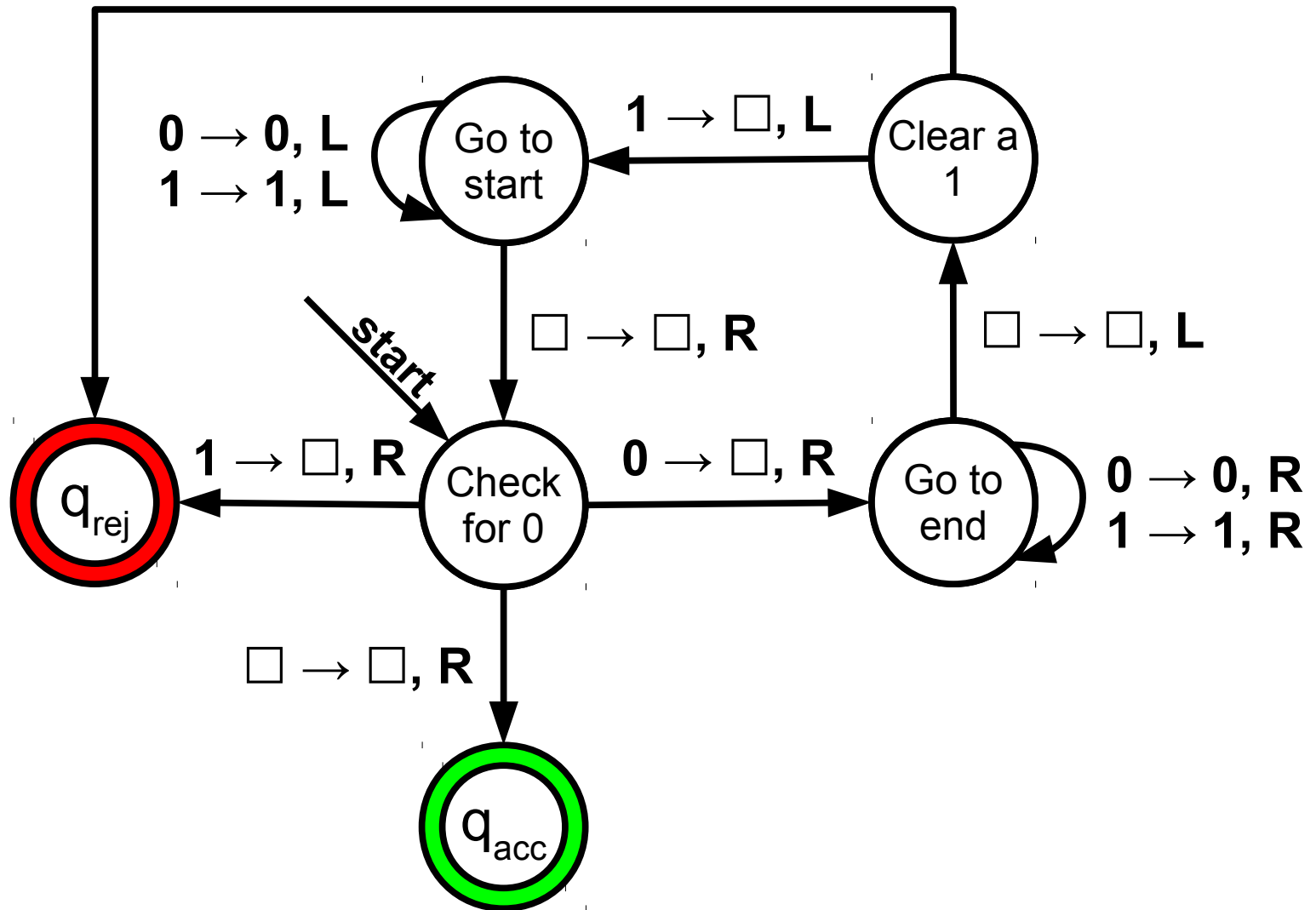
$$x \rightarrow y, D$$

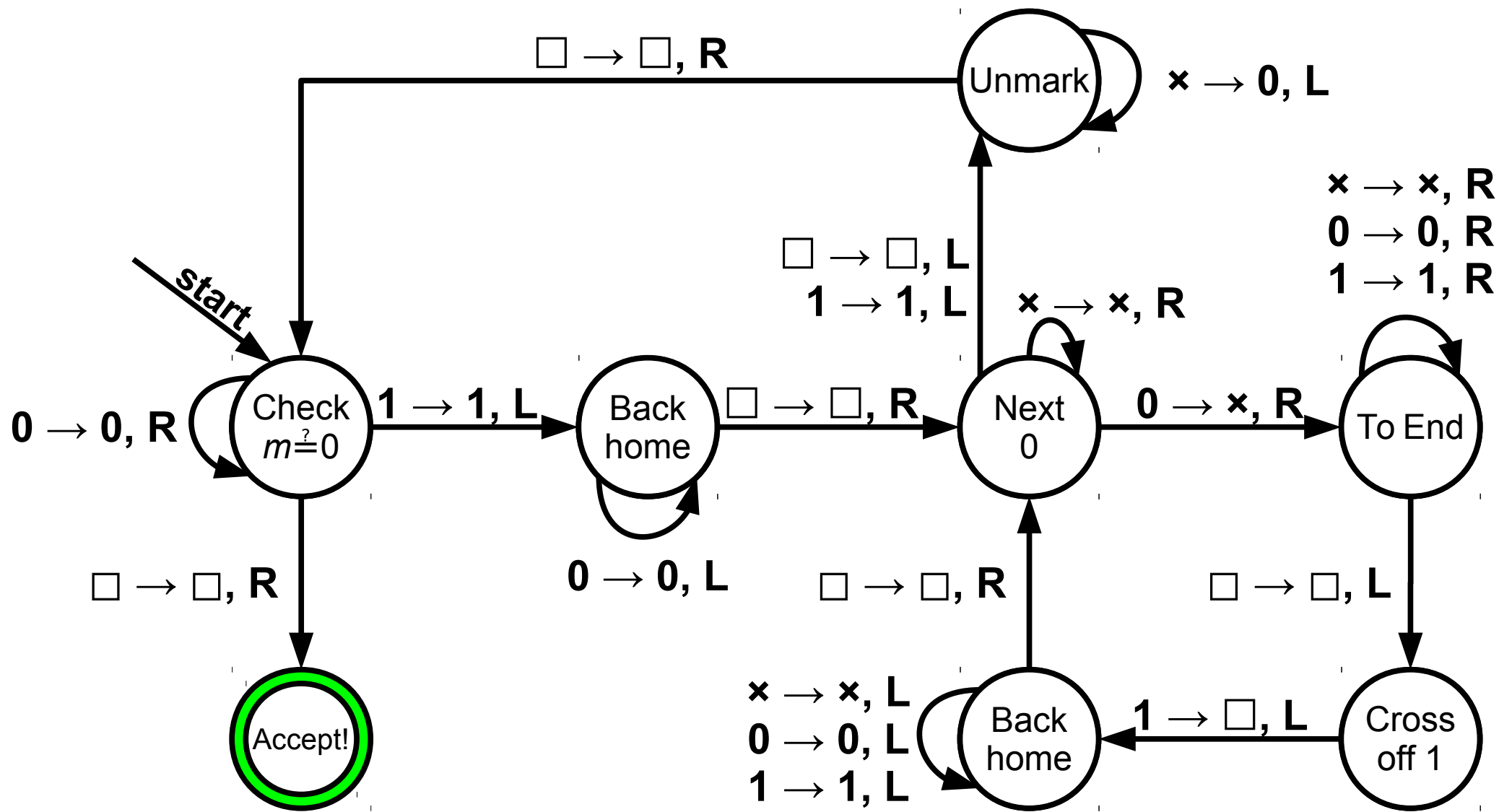
means "upon reading x , replace it with symbol y and move the tape head in direction D (which is either **L** or **R**).

The symbol \square represents the **blank symbol**.

This special **reject state** causes the machine to immediately reject.

$\square \rightarrow \square, R$
 $0 \rightarrow 0, R$





Main Question for Today:

Just how powerful are Turing machines?

A Sample Problem

Composite Numbers

- A natural number $n \geq 2$ is **composite** if it is the product of two natural numbers p and q , neither of which is n .
- Example:
 - $6 = 3 \cdot 2$ is composite.
 - $42 = 7 \cdot 6$ is composite
 - 5 is not composite.
- Programming problem: write a method that determines whether a natural number is composite.

```
bool isComposite(int n) {  
    if (n ≤ 1) return false;  
  
    int divisor = 2;  
    while (divisor ≠ n) {  
        if (n is a multiple of divisor) {  
            return true;  
        }  
        divisor++;  
    }  
  
    return false;  
}
```

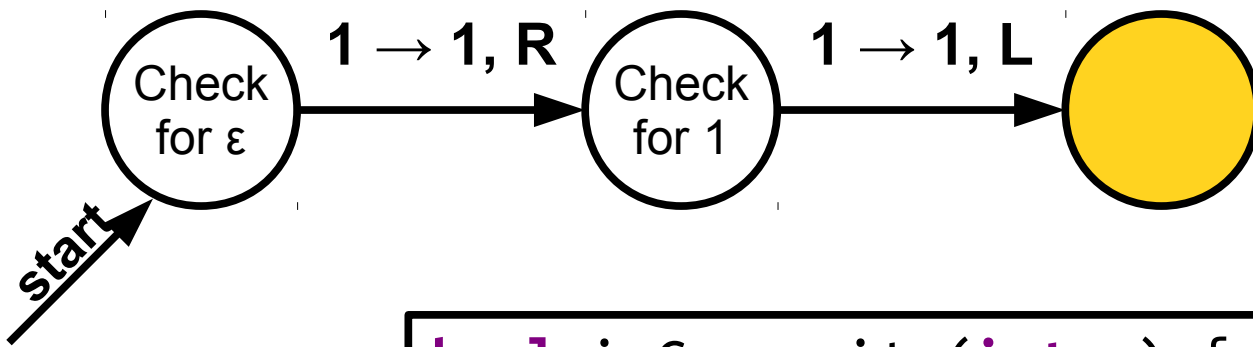
Question: Can we build a TM that determines whether a number is composite?

Back to Languages

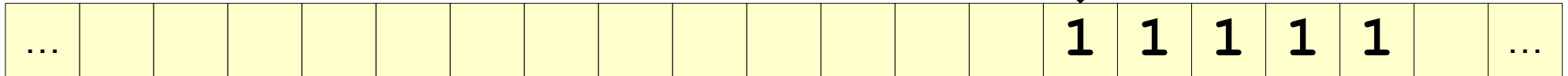
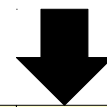
- Let $\Sigma = \{1\}$ and consider the following language L over Σ :

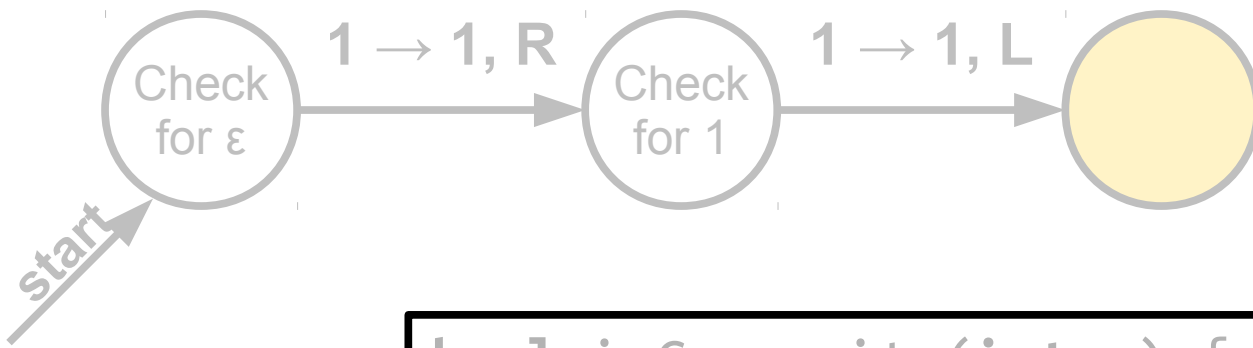
$$L = \{ 1^n \mid n \text{ is composite} \}$$

- This language is not regular (think about why this is).
- It's also not context-free (don't worry if you don't see why - we didn't discuss this in CS103. ☺)
- Can we build a TM for it?



```
bool isComposite(int n) {  
    if (n ≤ 1) return false;  
  
    int divisor = 2;  
    while (divisor ≠ n) {  
        if (n is a multiple of divisor) {  
            return true;  
        }  
        divisor++;  
    }  
  
    return false;  
}
```





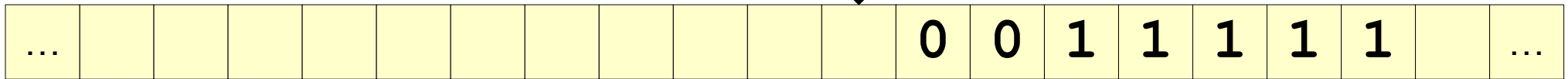
```
bool isComposite(int n) {  
    if (n ≤ 1) return false;  
  
    int divisor = 2;  
    while (divisor ≠ n) {  
        if (n is a multiple of divisor) {  
            return true;  
        }  
        divisor++;  
    }  
  
    return false;  
}
```



Reusing our TMs

- What does our TM need to be able to do?
 - Check if `divisor = n`
 - Check if `n` is a multiple of `divisor`.
- What TMs did we build last time?
 - One that checks for $0^n 1^n$
 - One that checks for $0^m 1^n$, where n is a multiple of m .
- Coincidence? I think not!

A Sketch of the TM



```
bool isComposite(int n) {  
    if (n ≤ 1) return false;  
    int divisor = 2;  
    while (divisor ≠ n) {  
        if (n is a multiple of divisor) {  
            return true;  
        }  
        divisor++;  
    }  
    return false;  
}
```

A Sketch of the TM



... 0 0 1 1 1 1 1 ...

```
bool isComposite(int n) {  
    if (n ≤ 1) return false;  
  
    int divisor = 2;  
    while (divisor ≠ n) {  
        if (n is a multiple of divisor) {  
            return true;  
        }  
        divisor++;  
    }  
  
    return false;  
}
```

Somehow, run our TM from last time to see if the string is of the form 0^n1^n .

A Sketch of the TM

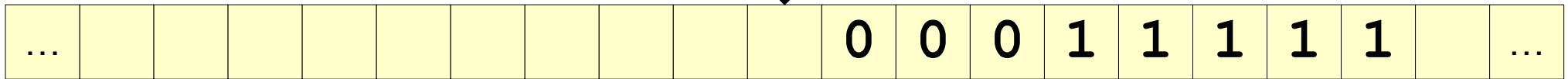
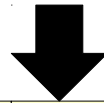


... 0 0 1 1 1 1 1 ...

```
bool isComposite(int n) {  
    if (n ≤ 1) return false;  
  
    int divisor = 2;  
    while (divisor ≠ n) {  
        if (n is a multiple of divisor) {  
            return true;  
        }  
        divisor++;  
    }  
  
    return false;  
}
```

Somehow, run our TM from last time to see if the string is of the form 0^m1^n , where n is a multiple of m .

A Sketch of the TM



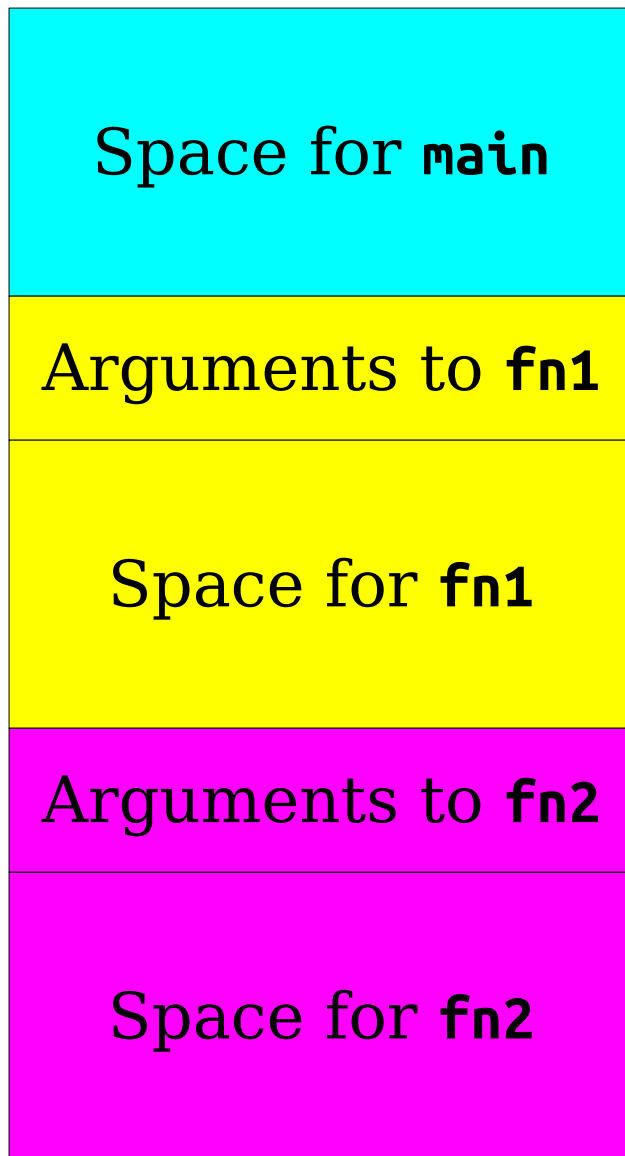
```
bool isComposite(int n) {  
    if (n ≤ 1) return false;  
  
    int divisor = 2;  
    while (divisor ≠ n) {  
        if (n is a multiple of divisor) {  
            return true;  
        }  
        divisor++;  
    }  
  
    return false;  
}
```

How can we get our new TM
to run our old TMs?

Function Call and Return

- Think about actual programming for a minute.
- If you write a function that performs a task, another part of the code can call that function to use the functionality.
- Could we do something like that here in our TM?

Function Call and Return

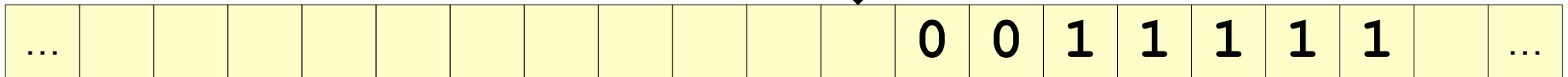
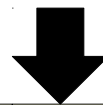


- Internally, the memory for each function call is allocated on the ***stack***:
 - Each call allocates more space.
 - Each return cleans up the space.
 - Arguments are copied by value.

Combining TMs

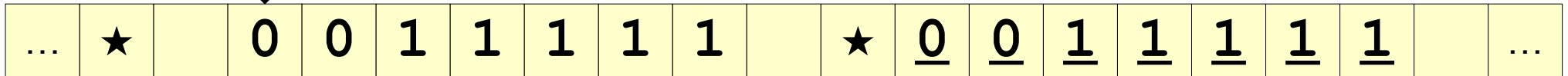
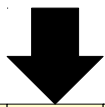
- ***Claim:*** We can use the TM's tape to simulate function call and return.
- This means that we can build very complex TMs by assembling smaller subroutines and combining them together.
- The details are icky; we'll see them in a second.

Our TM, in More Depth



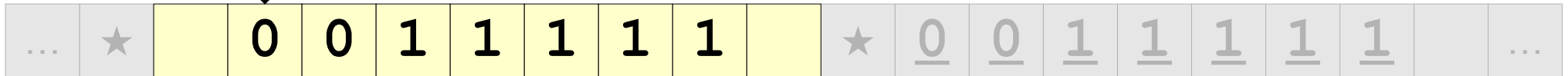
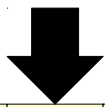
High-level idea: Pass the current string as a “parameter” to the TM for 0^n1^n .

Our TM, in More Depth



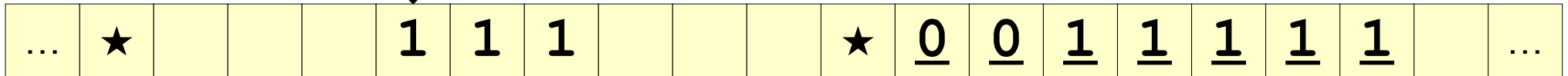
High-level idea: Pass the current string as a “parameter” to the TM for 0^n1^n .

Our TM, in More Depth



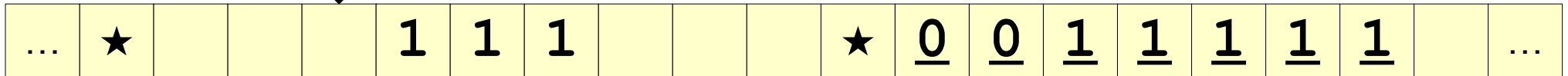
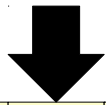
Now, go run the TM for $0^n 1^n$ by transitioning into its start state. It never takes more than one step off the string in either direction, so it has no idea there's anything else on the tape.

Our TM, in More Depth



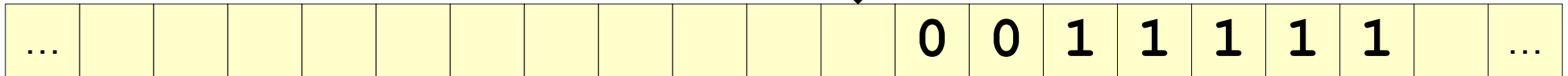
High-level idea:
Clean up the stack
space to return from
the function call.

Our TM, in More Depth



High-level idea:
Clean up the stack
space to return from
the function call.

Our TM, in More Depth



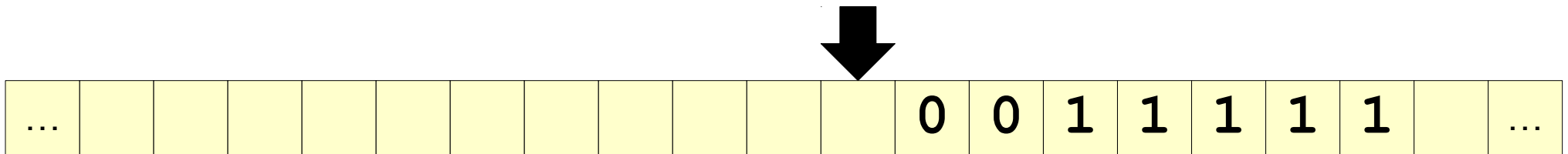
Now, you can imagine that we do the same thing, but for the TM that checks if the number of 1's is a multiple of the number of 0's.

Subroutines in TMs

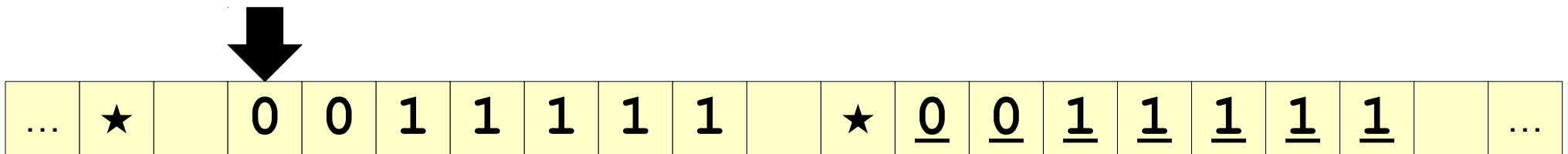
- Just as complex programs are often broken down into smaller functions and classes, complex TMs are often broken down into smaller “subroutines.”
- Each subroutine performs some task that helps in the overall task.
- The TM is then described by giving a collection of subroutines and showing how they link up.

The “Copy” Subroutine

- This subroutine starts with the tape head at the start of a string of 0s and 1s:



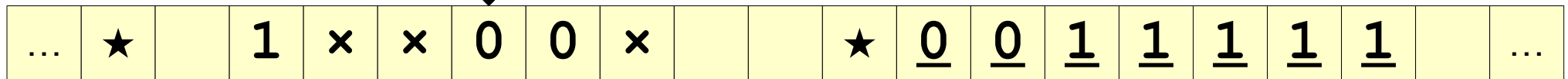
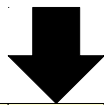
- It ends in this configuration:



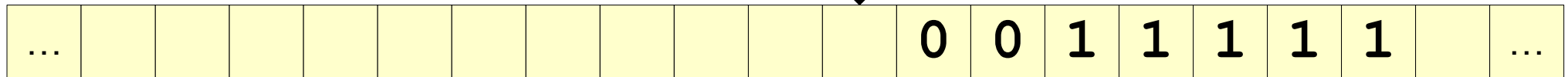
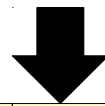
- We use the copy subroutine to let us run another TM on the current input without breaking it.

The “Cleanup” Subroutine

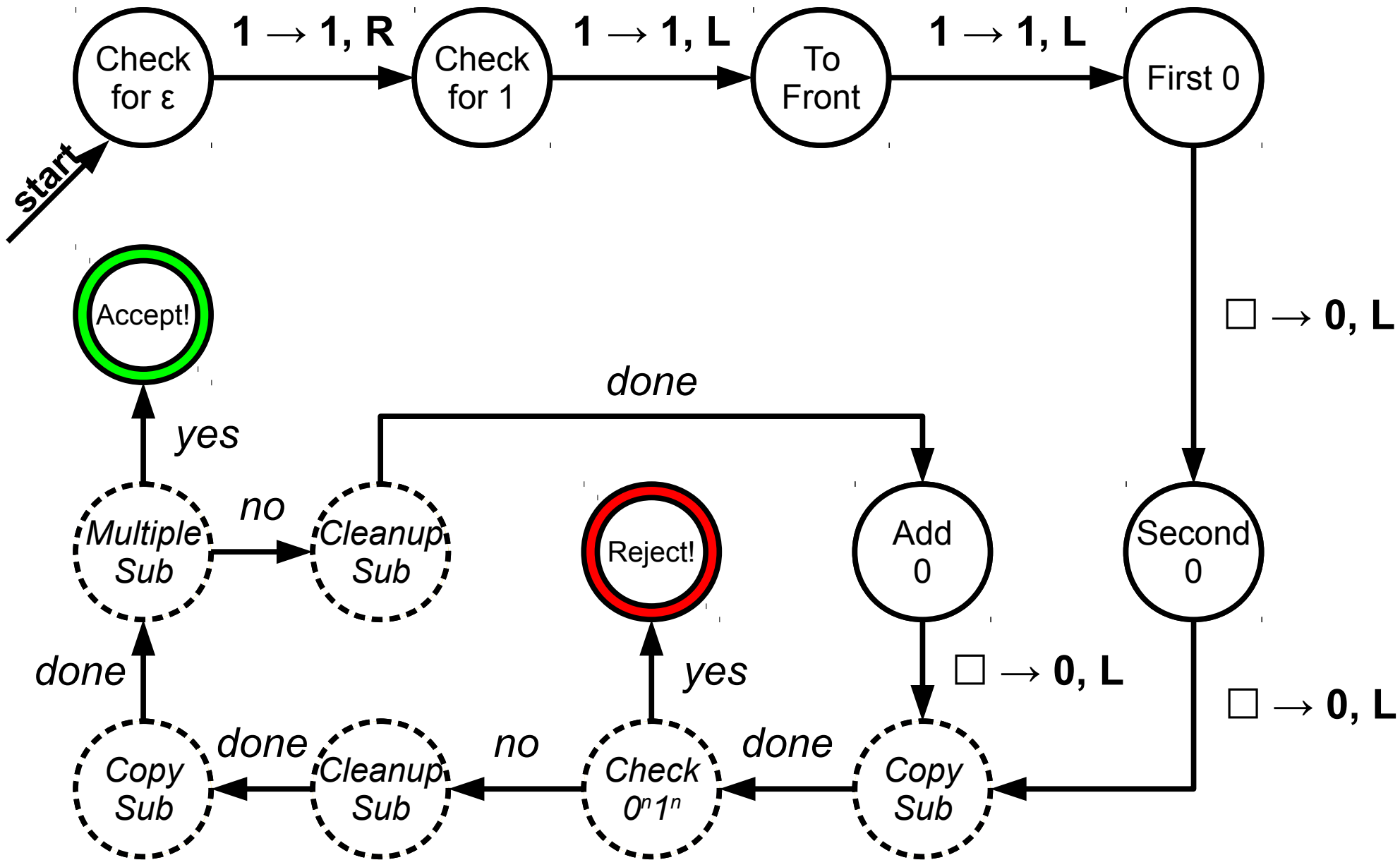
- This subroutine starts with the tape head between two ★ characters delimiting TM workspace:



- It ends in this configuration:



- We use the cleanup subroutine to recover from the end of running a sub-TM.

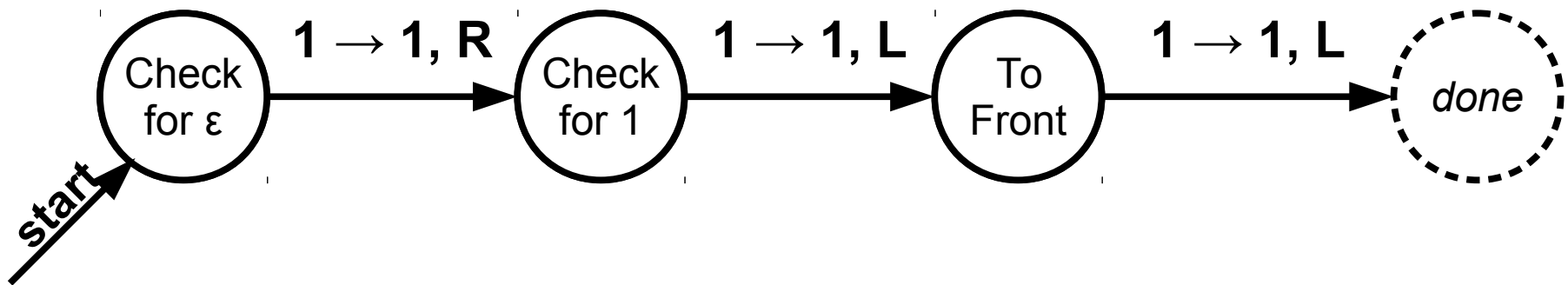


TM Subroutines

- To represent a subroutine call inside a TM, just add a state with dashed edges with the name of the subroutine.
- If the subroutine is just a “processing” subroutine, have a single exit arrow.
- If the subroutine may exit along different paths, have multiple exit arrows.

Defining a Subroutine

- If you want to (or are asked to) design a TM subroutine, design it like a normal TM with one change: have special dashed states representing the exit of the subroutine.



Time-Out for Announcements!

CFG Tool

- We have a tool available online you can use to develop and test out context-free grammars.
- Features include
 - Autogeneration of strings in the language of your grammar.
 - Automatic testing of whether a specific string can be generated.
- Doesn't support submission (sorry), but still highly recommended!

Midterm Logistics

- Midterm is next **Thursday, May 21** from **7PM - 10PM**, location TBA.
 - Cumulative, focusing primarily on topics from PS4 - PS6.
 - Covers material up through and including the lecture on CFGs; TMs will not be tested.
- Same format as last time: four questions, closed-book, closed-computer, open one double-sided 8.5" × 11" sheet of paper.
- Alternate exam: 4PM - 7PM on Thursday, May 21. Contact us ASAP if you'd like to take the exam at an alternate time.
- Practice exam next Monday, May 18th from 7PM - 10PM; location TBA.

Extra Practice Problems

- We've released another set of extra practice problems (Practice Problems 4) to help you review for the exam.
- We'll release solutions and another set of practice problems on Monday.

I wish my CS Professor had known...

“We are working to make the CS department here at Stanford a more welcoming environment for all students, especially those who are racial, gender, socioeconomic, or other minority students.

To assist with this, we are collecting responses from students that seek to share their experiences in CS courses with the members of the Stanford community. For this project, we are asking students who want to share things they wish their computer science professor knew about them or their own personal experiences in the classroom.

Please use the following **form** to anonymously submit your experience.

If you have any questions, comments, or concerns, contact us at ecortes@stanford.edu or vniu@stanford.edu.”

Midterms Regraded

- All midterm regrade requests have been processed. They're available for pickup in the filing cabinet where we normally returned exams.
- Going forward, please only submit regrades if we deducted points for something that's actually correct. Please don't use regrades as a way to ask for more partial credit.

Your Questions!

“I have no idea what all this 'language' stuff has to do with computation. I feel I understand it in isolation, but not in terms of how it ties in with figuring out what impossible problems are, how to find them, or how to even reason about them.”

We're just about
to talk about this -
hang in there!

“What implications do the limits of computation have on our understanding of the universe?”

We're also going to talk a bit about that soon. Some teasers:

1. Various problems in physics, chemistry, and biology can be solved by computers, but we suspect that they can't be solved efficiently at the necessary scale.
2. Certain economic models work if and only if specific computational problems can be solved efficiently.
3. There is a fundamental distinction between something being true and us being able to know that it's true.

“In powers of 2, how many atoms are in the universe?”

There are about 10^{80} atoms in the universe. Let's upper-bound that at 10^{100} .

Numbers you should know: $2^{10} \approx 10^3$

So $10^{100} \approx 2^{300}$.

This means that \log_2 (atoms in universe) ≈ 300 .

Back to CS103!

Main Question for Today:

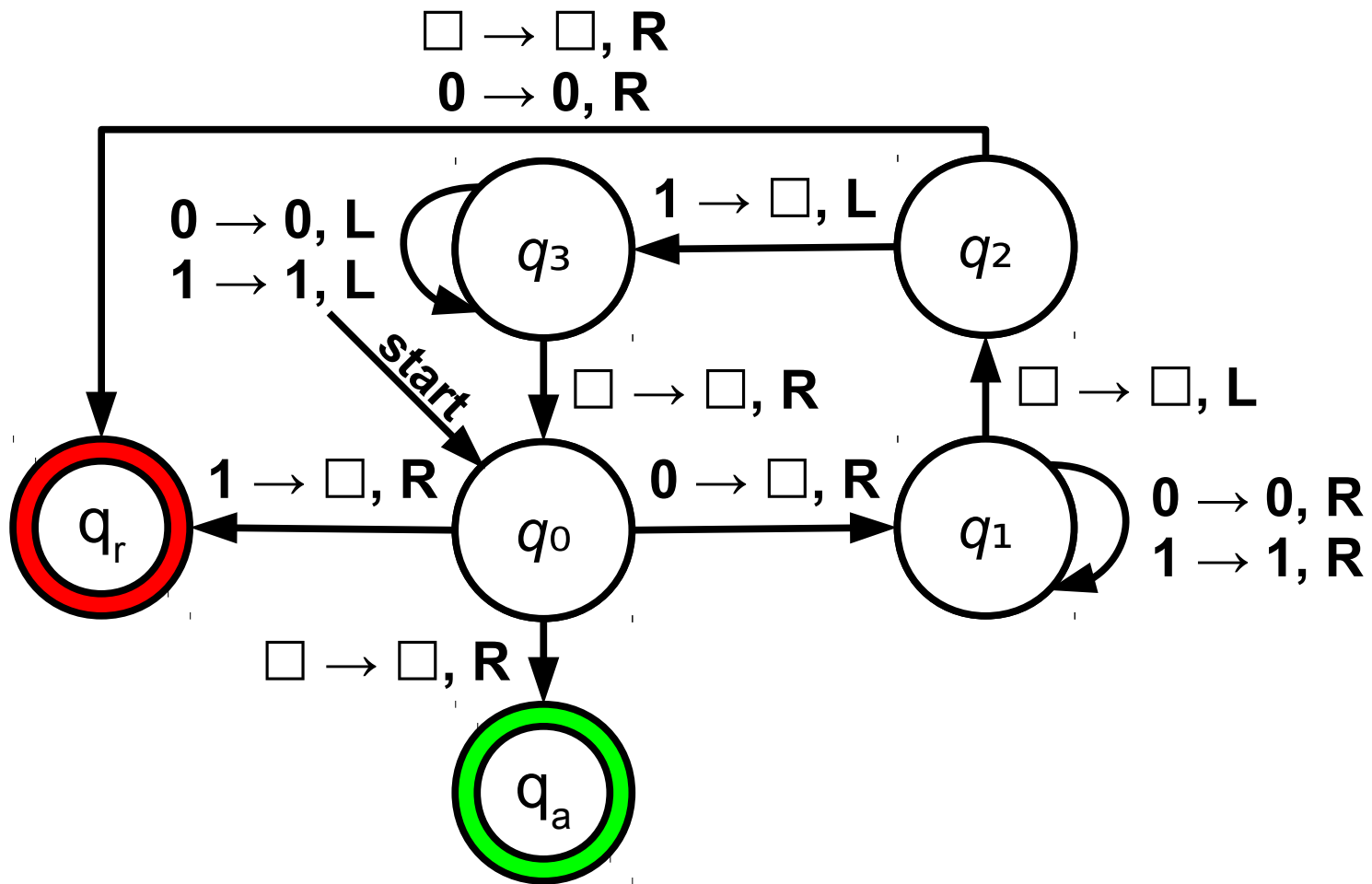
Just how powerful are Turing machines?

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.

Claim 1: Computers with unbounded memory can simulate Turing machines.

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



	0	1	\square
q_0	q_1 \square R	q_r \square R	q_a \square R
q_1	q_1 0 R	q_1 1 R	q_2 \square L
q_2	q_r 0 R	q_3 \square L	q_r \square R
q_3	q_3 0 L	q_3 1 L	q_0 \square 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 computers with unbounded memory.

“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 on the tape).
 - perform elementary arithmetic (equality testing, multiplication, addition, division, 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.
 - Each variable can be represented as a name (written one symbol at a time) followed by a value.
- 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 details, come talk to me after class.

Just how powerful *are* Turing machines?

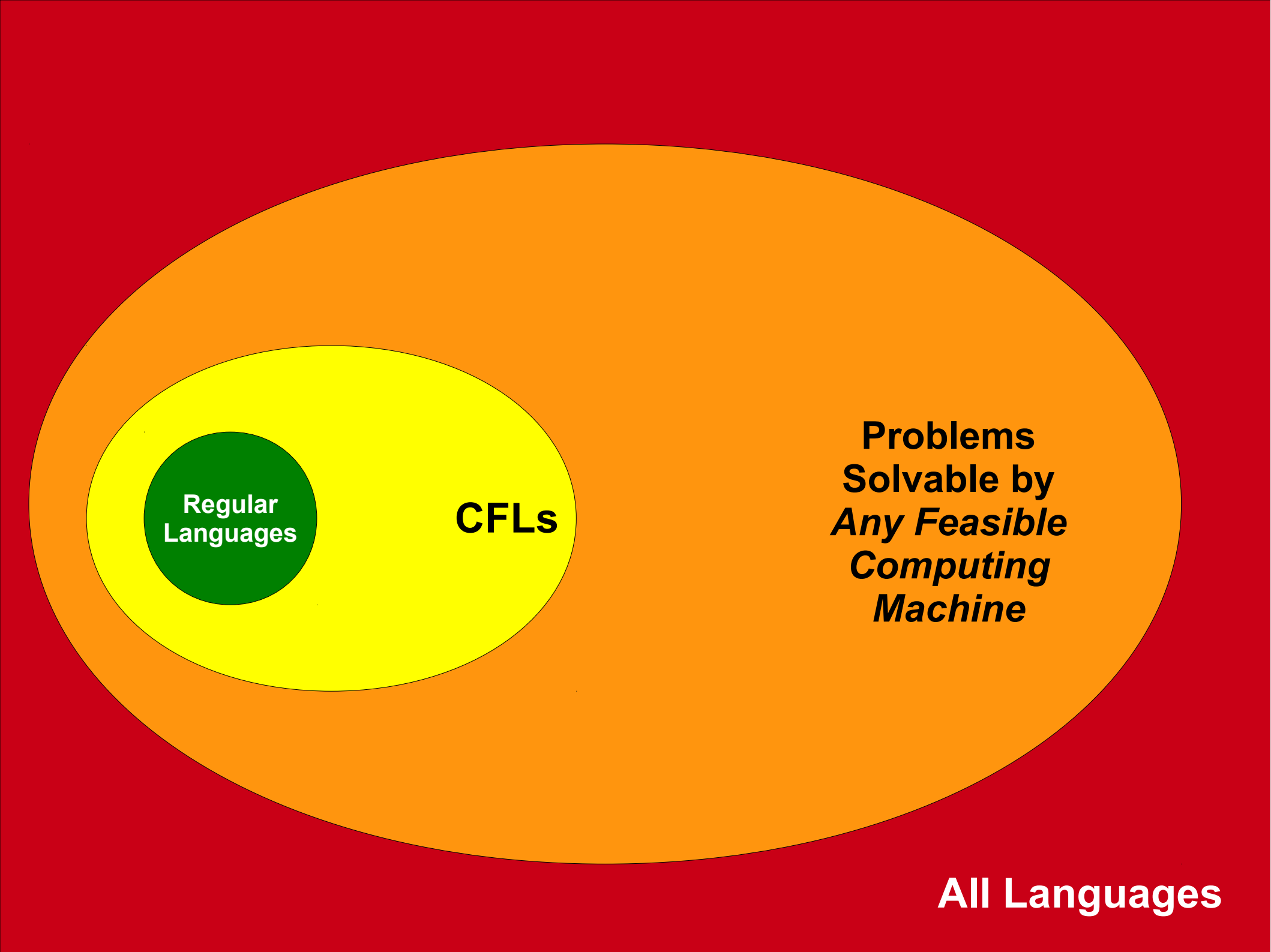
Effective Computation

- 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***

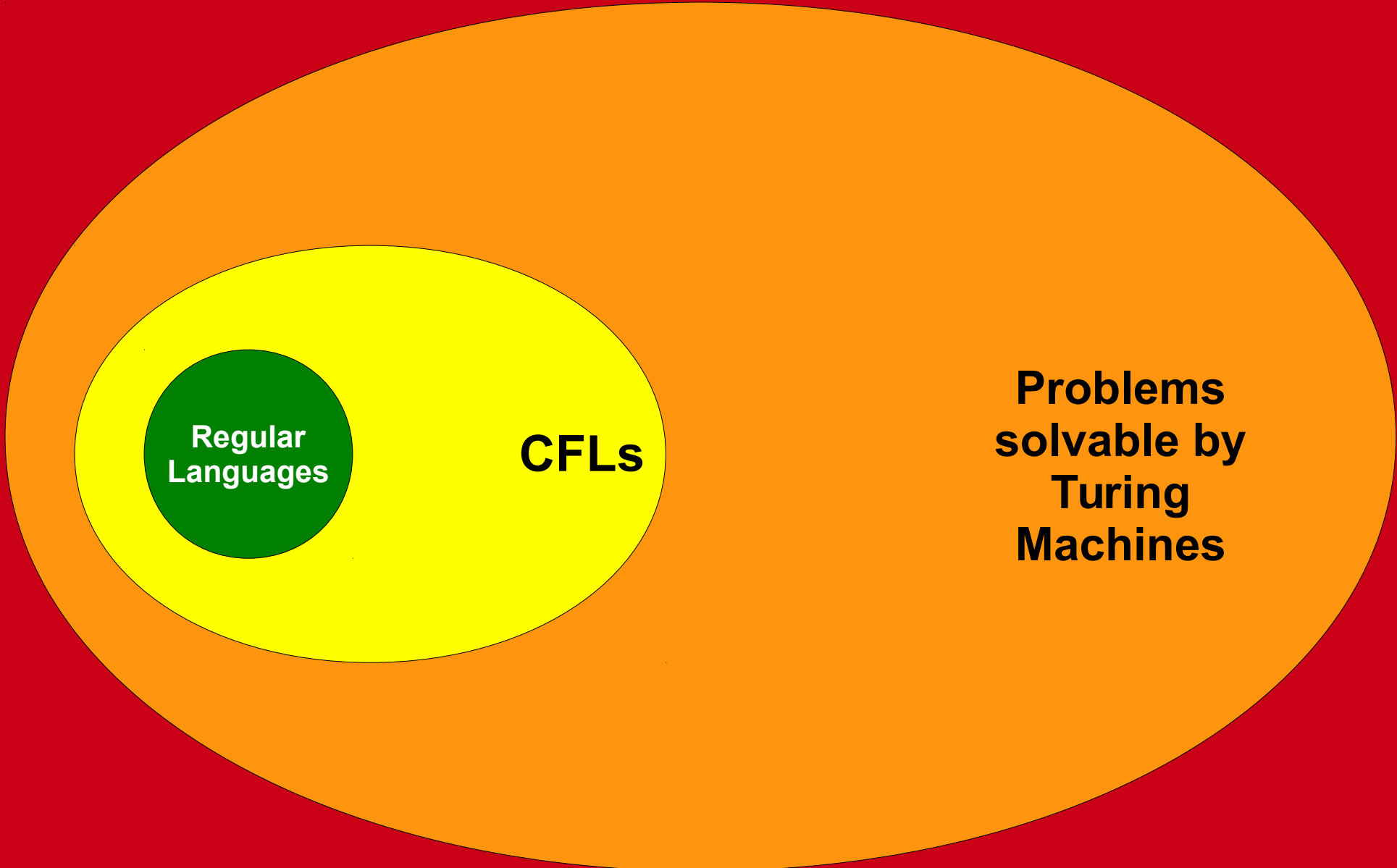
All Languages

**Regular
Languages**

CFLs

**Problems
solvable by
Turing
Machines**


All Languages



Strings, Languages, Encodings, and Problems


What problems can we solve with a computer?

What kind of
computer?



What **problems** can we solve with a computer?

What is a
"problem?"



Languages and Problems

- We've been using formal languages as a way of modeling computational problems.
- However, the problems we encounter in The Real World don't look at all like language problems.
- Is this all theoretical nonsense? Or is there a reason for this?

“In theory, there's no difference between theory and practice. In practice, there is.”

Decision Problems

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

- Example: checking arithmetic.

Given x , y , and z , is $x+y=z$?

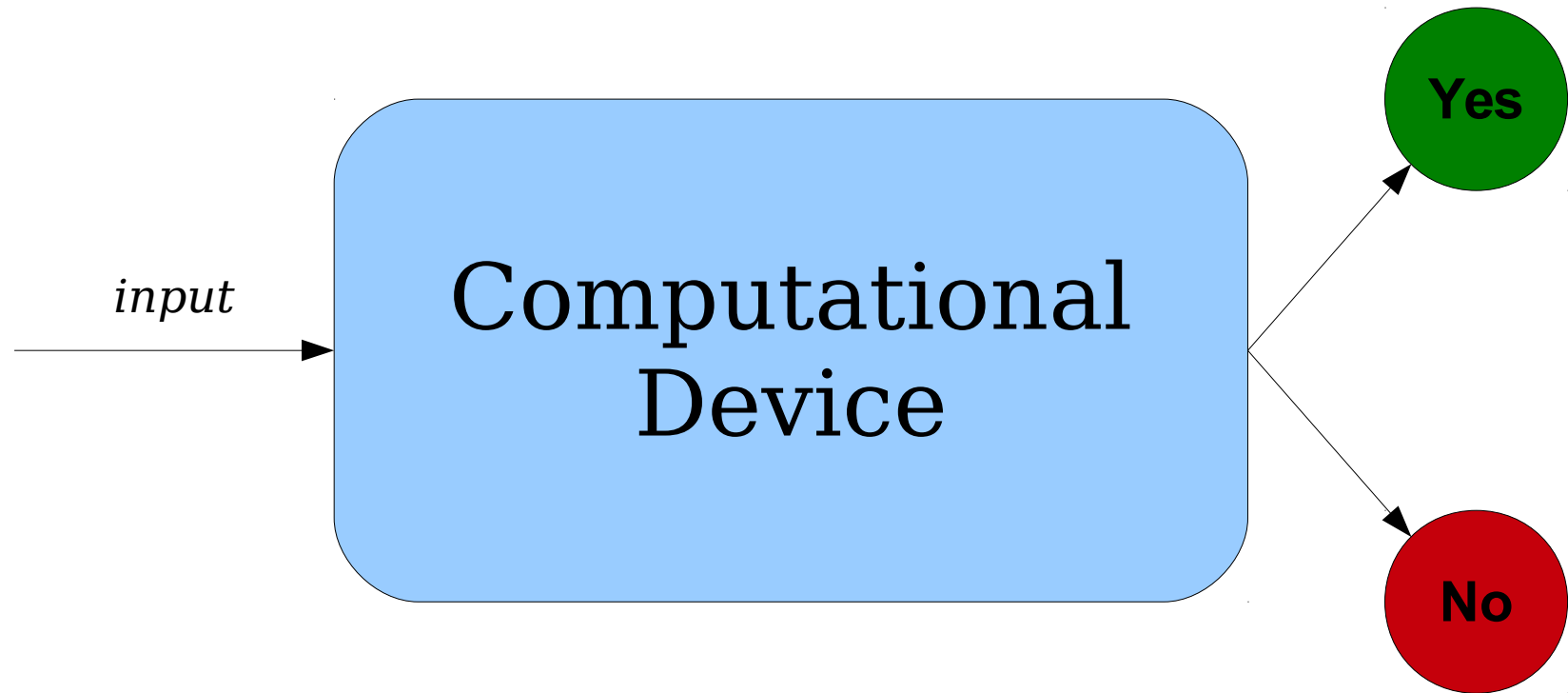
- Example: detecting relationships.

Given a family tree T and people x and y ,
is x a grandparent of y ?

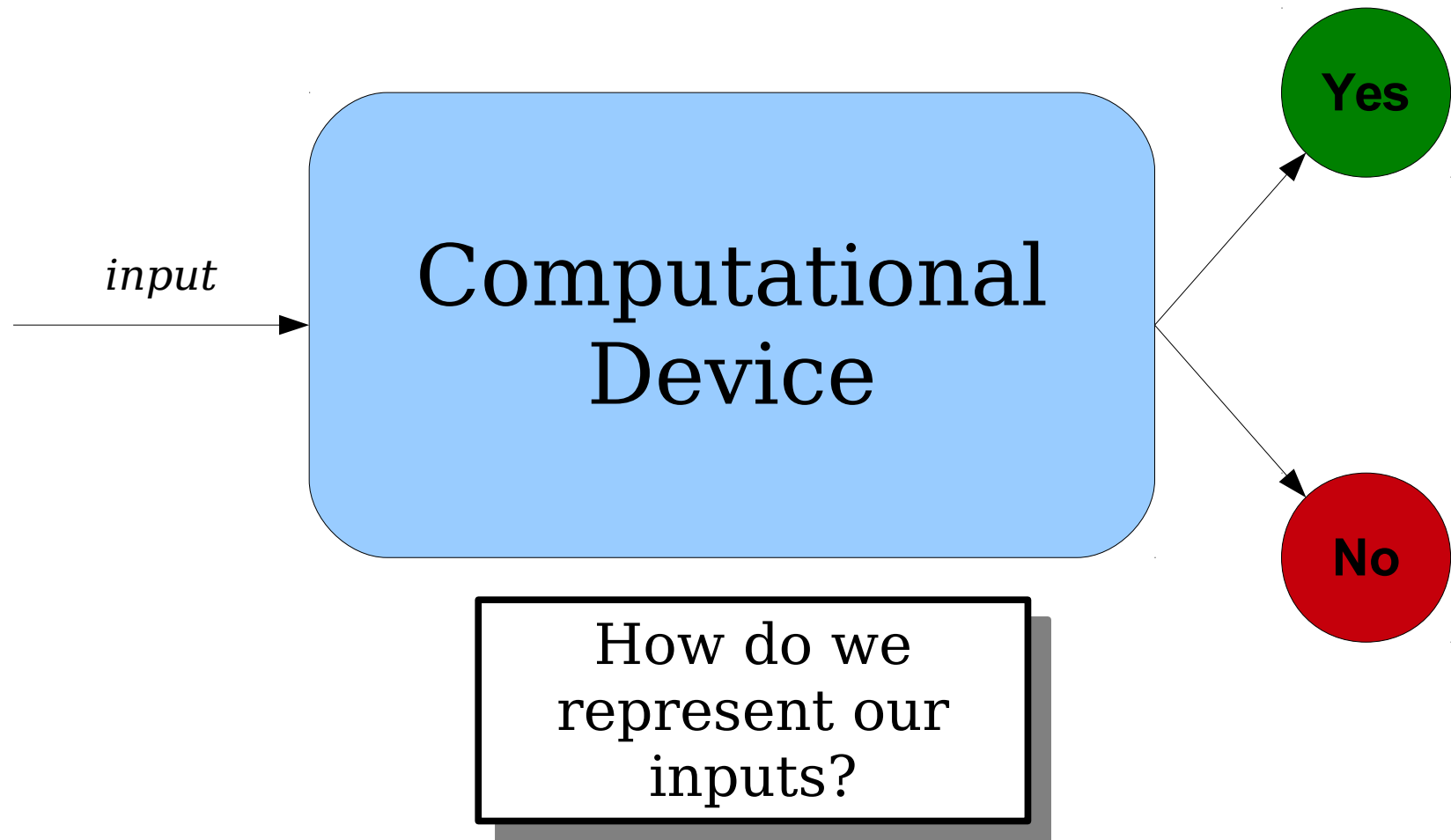
- Example: avoiding traffic.

Given a transportation grid G annotated with traffic information, a start location s , a destination d , and a time limit t , is there a way to get from s to d within time t ?

Solving Decision Problems



Solving Decision Problems



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.
- All data on my computer can be thought of as (suitably-encoded) strings of 0s and 1s.



Strings and Objects

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



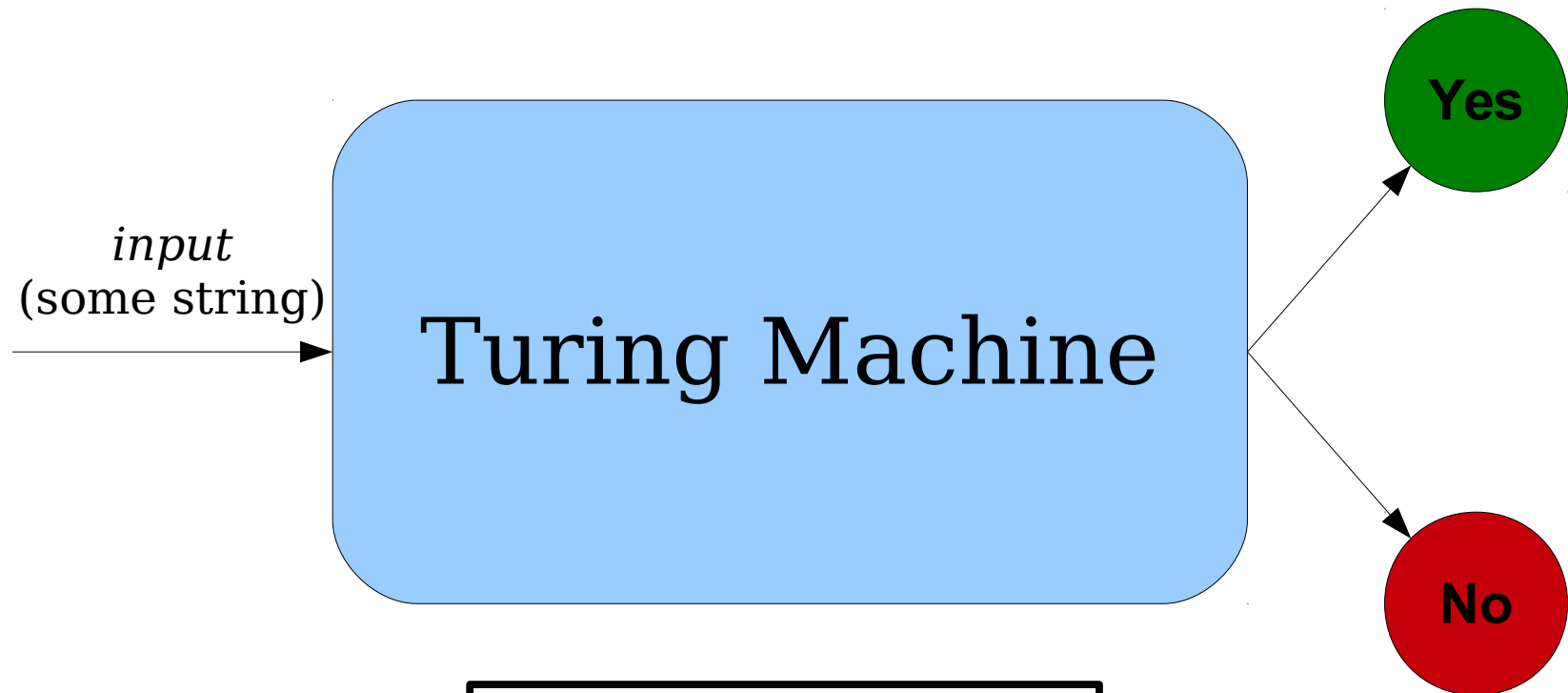
Strings and Objects

- Let *Obj* be some discrete, finite object (a string, a video, an image, a text file, etc.)
- Let Σ be some alphabet.
- We'll represent an **encoding of *Obj*** using the characters in Σ by writing **$\langle Obj \rangle$** . Think of $\langle Obj \rangle$ like a file on disk – it encodes complex data as a series of characters.
- A few remarks about encodings:
 - We don't care *how* we encode the object, just that we can.
 - The particular choice of alphabet isn't important. Given any alphabet, we can always find a way of encoding things.
 - We'll assume we can perform “reasonable” operations on encoded objects.

Strings and Objects

- Given a group of objects $Obj_1, Obj_2, \dots, Obj_n$, we can create a single string encoding all these objects.
 - Think of it like a .zip file, but without the compression.
- We'll denote the encoding of all of these objects as a single string by **$\langle Obj_1, \dots, Obj_n \rangle$** .
- This lets us feed multiple inputs into our computational device at the same time.

Solving Decision Problems



How do we specify
the behavior we
want?

Specifying a Decision Problem

- Consider this decision problem:

Given $x, y, z \in \mathbb{N}$, determine whether $x+y=z$.

- With our computational model, we'll feed some string into a TM, and it then might come back with an answer (yes or no).
- Some strings are accepted, some are rejected, and some cause the machine to loop infinitely.

Specifying a Decision Problem

- Consider this decision problem:

Given $x, y, z \in \mathbb{N}$, determine whether $x+y=z$.

- If we give the input as $\langle x, y, z \rangle$, the set of strings the TM should say YES to is

$\{ \langle x, y, z \rangle \mid x, y, z \in \mathbb{N} \text{ and } x + y = z \}$

- Notice that this is a language - it's a set of strings!

Problems and Languages

- ***Key intuition:*** Every language corresponds to some decision problem.
- Example:
 - $\{ \langle x, y \rangle \mid x, y \in \mathbb{N} \text{ and } x \equiv_3 y \}$ is a language.
 - It corresponds to the following decision problem:

Given $x, y \in \mathbb{N}$, do x and y leave the same remainder when divided by 3?

Problems and Languages

- ***Key intuition:*** Every language corresponds to some decision problem.
- Example:
 - $\{ \langle D \rangle \mid D \text{ is a DFA that accepts } \varepsilon \}$ is a language.
 - It corresponds to the following decision problem:
Given a DFA D , does D accept ε ?

Problems and Languages

- ***Key intuition:*** Every language corresponds to some decision problem.
- Example:
 - $\{ \langle G \rangle \mid G \text{ is a planar graph} \}$ is a language.
 - It corresponds to the following decision problem:

Given a graph G , is G planar?

Problems and Languages

- **Key intuition:** Every decision problem on finite, discrete objects corresponds to some language.
- If Π is a problem and the inputs to Π are objects x_1, x_2, \dots, x_n , we can form the language
 $\{ \langle x_1, x_2, \dots, x_n \rangle \mid \text{the answer to problem } \Pi, \text{ given inputs } x_1, \dots, x_n, \text{ is "yes."} \}$

What All This Means

- Our goal is to speak of *computers solving problems*.
- We will model this by looking at *TMs recognizing languages*.
- For *decision problems* that we're interested in solving, this precisely captures what we're interested in capturing.