Unsolvable Problems

Part One

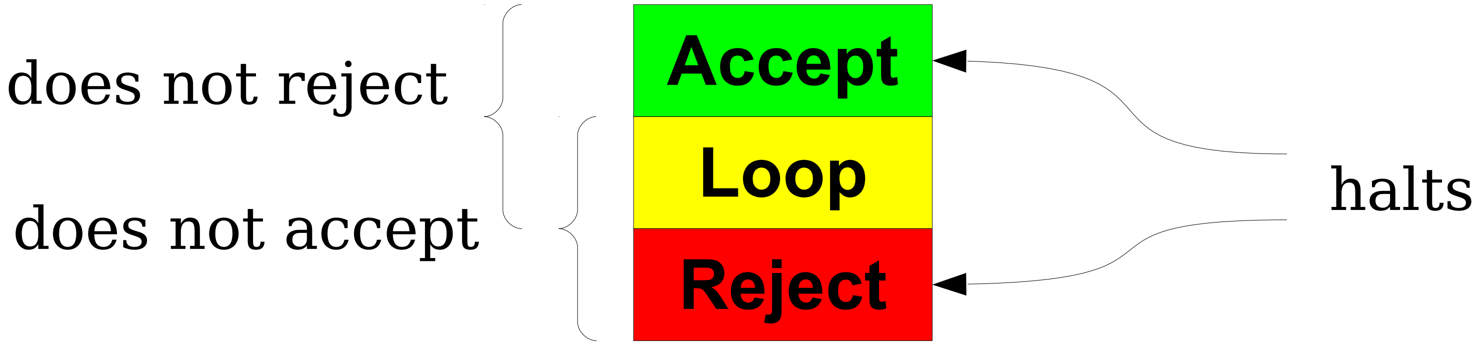
A (Not So) Brief Recap of Last Time

What problems can we solve with a computer?

What does it
mean to solve
a problem?

Very Important Terminology

- Let M be a Turing machine.
- M ***accepts*** a string w if it enters the accept state when run on w.
- M ***rejects*** a string w if it enters the reject state when run on w.
- M ***loops infinitely*** (or just ***loops***) on a string w if when run on w it enters neither the accept or 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:

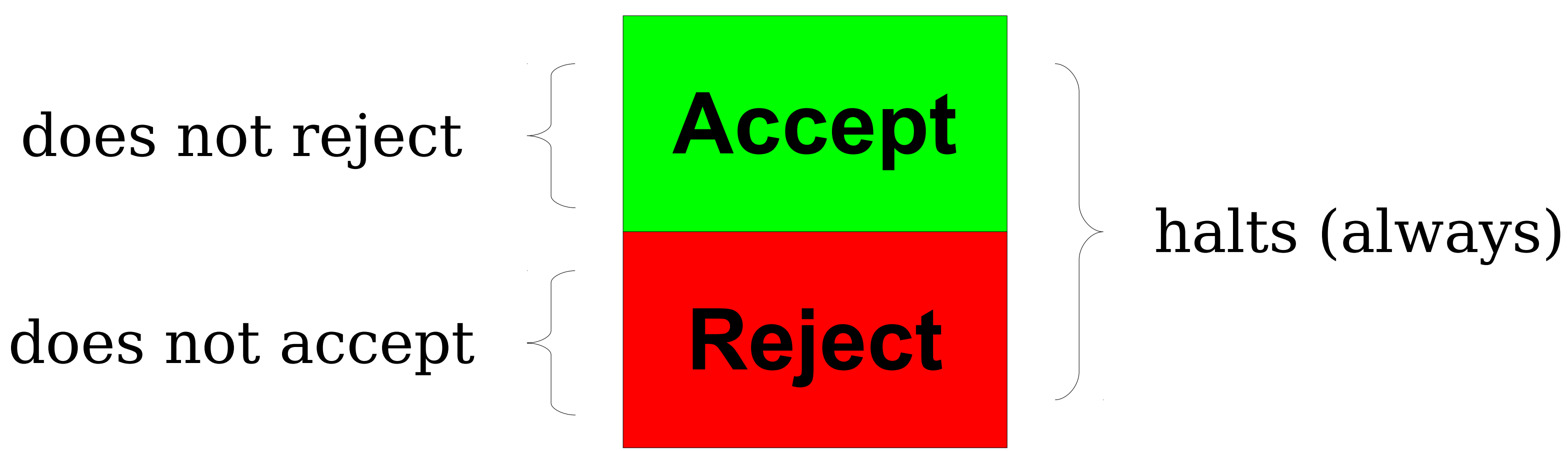
$$\mathscr{L}(\boldsymbol{M}) = \{ \boldsymbol{w} \in \boldsymbol{\Sigma}^* \mid \boldsymbol{M} \textbf{ accepts } \boldsymbol{w} \}$$

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

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

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.

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

The Universal Turing Machine

- ***Theorem**:* There is a Turing machine $\mathbf{U_{TM}}$ called the ***universal Turing machine*** that, when run on ⟨*M*, *w*⟩, where *M* is a Turing machine and *w* is a string, simulates *M* running on *w*.
- Conceptually:

U_{TM} = "On input ⟨*M*, *w*⟩, where *M* is a TM and $w \in \Sigma^*$:

Set up the initial configuration of *M* running on *w*.

```
while (true) {
```

If *M* accepted *w*, then U_{TM} accepts ⟨*M*, *w*⟩.

If *M* rejected *w*, then U_{TM} rejects ⟨*M*, *w*⟩.

Otherwise, simulate one more step of *M* on *w*.

```
}
```
"

The Language of U_{TM}

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

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

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

- For simplicity, define $\mathbf{A_{TM}} = \mathscr{L}(\mathbf{U_{TM}})$.

Self-Referential Programs

- ***Claim:*** Going forward, assume that any program can be augmented to include a method called `mySource()` that returns a string representation of its source code.
- General idea:
  - Write the initial program with `mySource()` as a placeholder.
  - Use the Quine technique we just saw to convert the program into something self-referential.
  - Now, `mySource()` magically works as intended.

The Recursion Theorem

- There is a deep result in computability theory called ***Kleene's second recursion theorem*** that, informally, states the following:

It is possible to construct TMs that perform arbitrary computations on their own descriptions.

- Intuitively, this generalizes our Quine constructions to work with arbitrary TMs.
- Want the formal statement of the theorem? Take CS154!

A Recipe for Disaster

- 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 Recipe for Disaster

- 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?

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 happens if…

… this program accepts its input?

It rejects the input!

… this program doesn't accept its input?

It accepts the input!

Outline for Today

- What exactly did we just do?
- How would we prove it?
- Why does any of this matter?
- What other problems are unsolvable?
- And what does “unsolvable” even mean?

First, The Proof

Theorem: $\mathrm{A}_{\mathrm{TM}} \notin \mathbf{R}$.

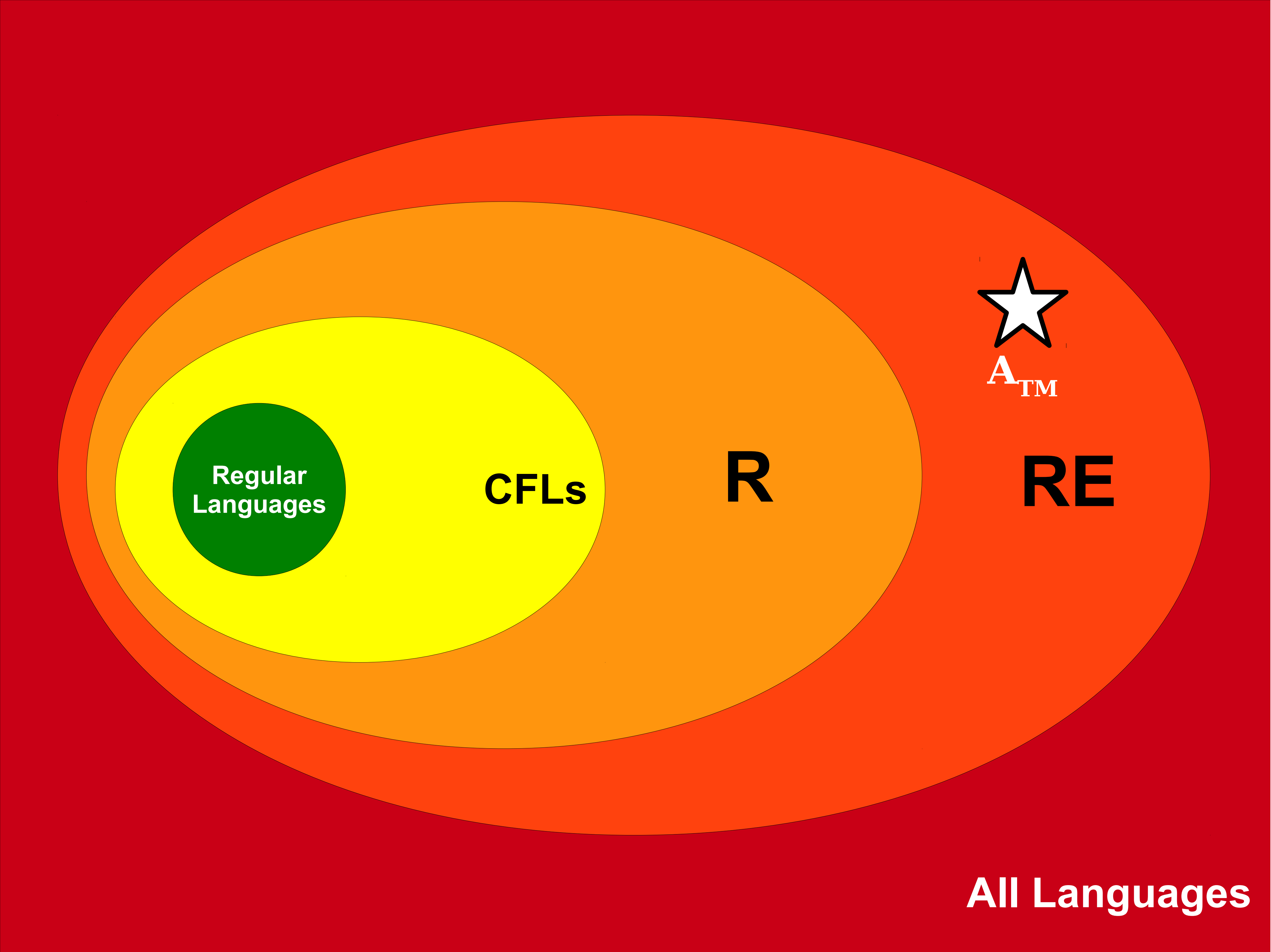
Proof: By contradiction; assume that $\mathrm{A}_{\mathrm{TM}} \in \mathbf{R}$. Then there is some decider D for A_{TM}. If this machine is given any TM/string pair, it will then determine whether the TM accepts the string and report back the answer.

Given this, we could then construct the following TM:

M = "On input w:
  Have M obtain its own description, $\langle M \rangle$.
  Run D on $\langle M, w \rangle$ and see what it says.
  If D says that M will accept w, reject.
  If D says that M will not accept w, accept."

Choose any string w and trace through the execution of the machine, focusing on the answer given back by machine D. If D says that M will accept w, notice that M then proceeds to reject w, contradicting what D says. Otherwise, if D says that M will not accept w, notice that M then proceeds to accept w, contradicting what D says.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, $\mathrm{A}_{\mathrm{TM}} \notin \mathbf{R}$. ■



A_{TM}
Regular Languages
CFLs
R
RE
All Languages

What Does This Mean?

- In one fell swoop, we've proven that
  - A_{TM} is ***undecidable***; there is no 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 only relies on the *observable behavior* of a proposed decider for A_{TM} and not on its internal workings. This immediately rules out all possible implementations!

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

- At a more fundamental level, the existence of undecidable problems tells us the following:

There is a difference between what is true and what we can show is true.

- Given an TM and any string w, either the TM accepts the string or it doesn't – *but there is no algorithm we can follow that will tell us which it is!*

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

- What exactly does it mean for A_{TM} to be undecidable?
- ***Intuition: The only general way to find out what a program will do is to run it.***
- As you'll see, this means that it's provably impossible for computers to be able to answer questions about what a program will do.

R ≠ RE

- The fact that **R ≠ RE** has enormous philosophical ramifications.
- A problem is in class **R** if there is an *algorithm* for solving it – there's some computational procedure that will give you the answer.
- A problem is in class **RE** if there is a *semialgorithm* for it. If the answer is “yes,” the machine can tell this to you, but if the answer is “no,” you may never learn this.
- 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 Impossibility 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

$$\boldsymbol{HALT = \{ \langle M, w \rangle \mid M \text{ is a TM that halts on } w \}}$$

- How hard is this problem to solve?
- How do we know?

$HALT \in \mathbf{RE}$

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

```
int main() {
    TM M = getInputTM();
    string w = getInputString();

    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;
    }
}
```

$HALT \notin \mathbf{R}$

- ***Claim:*** $HALT \notin \mathbf{R}$.
- 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 willHalt(string program, string input) {
   /* … some implementation … */
}

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

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

What happens if…

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

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

Theorem: $HALT \notin \mathbf{R}$.

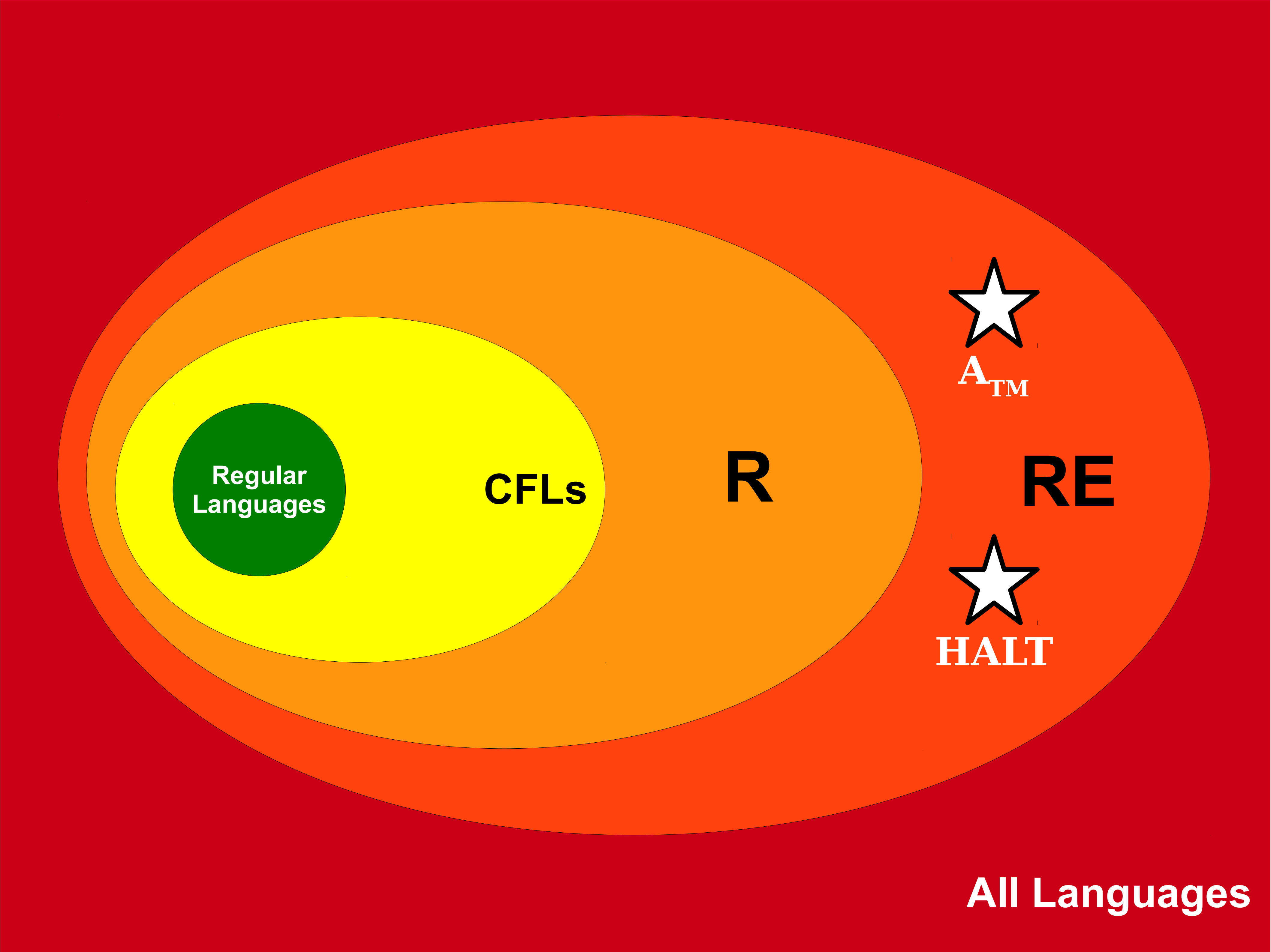
Proof: By contradiction; assume that $HALT \in \mathbf{R}$. Then there is some decider D for $HALT$. If this machine is given any TM/string pair, it will then determine whether the TM halts on the string and report back the answer.

Given this, we could then construct the following TM:

M = "On input w:
Have M obtain its own description, $\langle M \rangle$.
Run D on $\langle M, w \rangle$ and see what it says.
If D says that M halt on w, go into an infinite loop.
If D says that M loop on w, accept."

Choose any string w and trace through the execution of the machine, focusing on the answer given back by machine D. If D says that M will halt on w, notice that M then proceeds to loop on w, contradicting what D says. Otherwise, if D says that M will loop on w, notice that M then proceeds to accept w, so M halts on w, contradicting what D says.

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


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

So What?

- These problems might not seem all that exciting, so who cares if we can't solve them?
- Turns out, this same line of reasoning can be used to show that some very important problems are impossible to solve.

Secure Voting

- Suppose that you want to make a voting machine for use in an election between two parties.
- Let Σ = {`r`, `d`}. A string in *w* corresponds to a series of votes for the candidates.
- Example: `rrdddrd` means “two people voted for `r`, then three people voted for `d`, then one more person voted for `r`, then one more person voted for `d`.”

Secure Voting

- A voting machine is a program that accepts a string of **r**'s and **d**'s, then reports whether person **r** won the election.
- Formally: a TM M is a voting machine if $\mathscr{L}(M) = \{ w \in \{\mathtt{r}, \mathtt{d}\}^* \mid w$ has more $\mathtt{r}$'s than $\mathtt{d}$'s $\}$
- ***Question:*** Given a TM that claims to be a voting machine, could we check whether it actually is a fair voting machine?

Secure Voting

- The ***secure voting problem*** is the following:

Given a TM *M*, is the language of *M* { *w* ∈ {`r`, `d`}* | *w* has more `r`'s than `d`'s }?

- **Claim:** This problem is not decidable – there is no algorithm that can check an arbitrary TM to verify that it's a secure voting machine!

Secure Voting

- Suppose that the secure voting problem is decidable. Then we could write a function

```
bool isSecureVotingMachine(string program)
```

  that would accept as input a program and return whether or not it's a secure voting machine.
- As you might expect, this lets us do Cruel and Unusual Things…

```
bool isSecureVotingMachine(string program) {
   /* … some implementation … */
}

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

   bool actualAnswer =
       countRs(input) > countDs(input);

   if (isSecureVotingMachine(me)) {
      return !actualAnswer;
   } else {
      return actualAnswer;
   }
}
```

What happens if…

… this program is a secure voting machine?

It's not a secure machine!

… this program is not a secure voting machine?

It is a secure voting machine!

This previous example is not contrived!

This is a problem we really would like to be able to solve!

Yet it's provably impossible!

Time-Out for Announcements!

Second Midterm Exam

- Second midterm exam is this Thursday, May 21 from 7PM – 10PM.
- Rooms divvied up by last (family) name:
  - Aba – Sow: Go to **Hewlett 200**
  - Spe – Zoc: Go to **Hewlett 201**
- Closed-book, closed-computer, open one double-sided 8.5” × 11” sheet of notes.
- Cumulative, focusing on PS4 – PS6.

Practice Midterm Exam

- We will be holding a practice midterm exam ***tonight*** from 7PM – 10PM in room 320-105.
- Structure and format of practice exam is similar to that of the main exam.
- TAs will be on-hand to answer questions; we'll release solutions as well.
- Can't make it? Don't worry! We'll post the exam on the course website.

More Practice Problems

- Solutions to Extra Practice Problems 5 are available for pickup right now.
- We've released a sixth and final set of extra practice problems you can use to prepare for the midterm.
- Solutions will go out on Wednesday.

Problem Set Seven

- Problem Set Six was due at the start of class.
  - Due tomorrow by 12:50PM with one late day and on Wednesday at 12:50PM with two.
  - Solutions will go out on Wednesday.
- Problem Set Seven goes out now. It's due on Wednesday of next week.
  - Play around with Turing machines, **R**, **RE**, and the limits of computation!

Turing Machine Tool

- This quarter, we're piloting a new tool you can use to design, edit, test, and submit Turing machines.
- We'll send out an email with details about this later today or early tomorrow.
- Please email the staff list with any feedback – we want this tool to be as useful as possible!

WiCS Casual Dinner

- WiCS is holding their second biquarterly Casual CS Dinner on Wednesday from 6:00PM – 8:00PM in the Women's Community Center.
- This is a wonderful event and I highly recommend it!
- RSVP requested; use ***this link***.

Checking In – Seriously

Back to CS103!

Beyond **R**

What exactly is the class **RE**?

An Intuition for **RE**

- Intuitively, a language L is in **RE** if a TM can search for positive proof that a string w belongs to L.
- Such a machine could work as follows:
  - Find a possible proof.
  - Check the proof.
  - If correct, accept!
  - If not, try the next proof.

The `MU` Puzzle

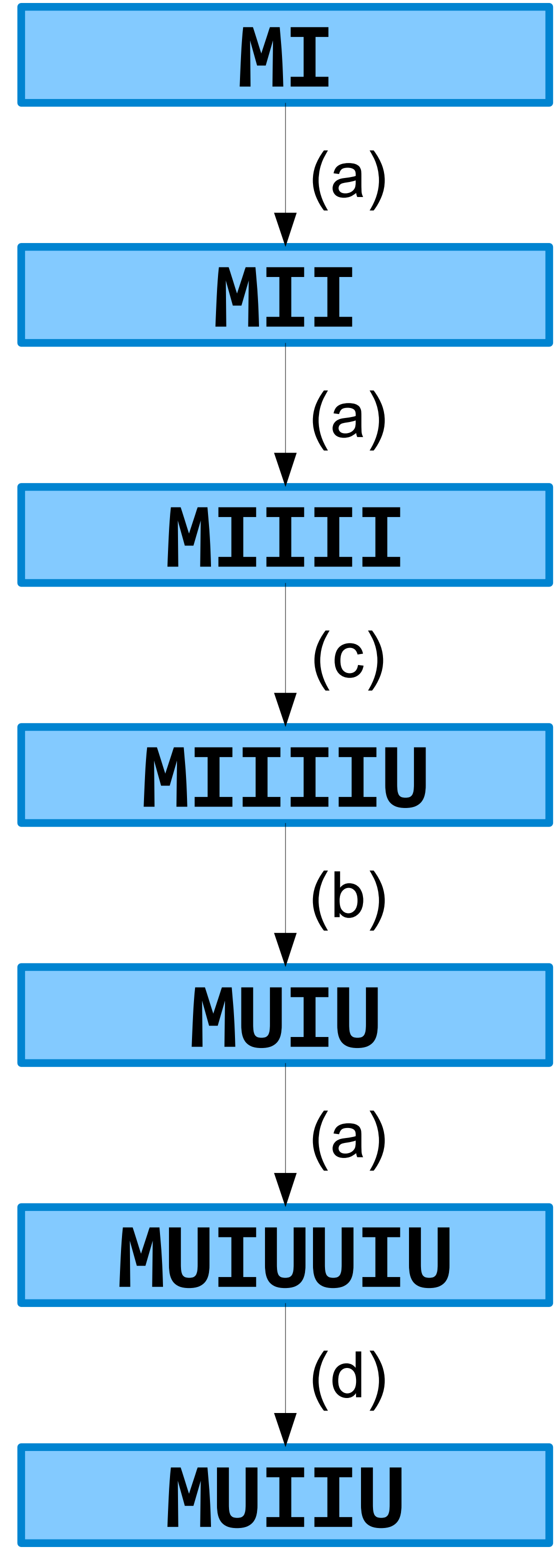
- Begin with the string `MI`.
- Repeatedly apply one of the following operations:
  - Double the contents of the string after the `M`: for example, `MIIU` becomes `MIIUIIU`, or `MI` becomes `MII`.
  - Replace `III` with `U`: `MIIII` becomes `MUI` or `MIU`.
  - Append `U` to the string if it ends in `I`: `MI` becomes `MIU`.
  - Remove any `UU`: `MUUU` becomes `MU`.
- ***Question***: How do you transform `MI` to `MU`?

(a) Double the string after an `M`.

(b) Replace `III` with `U`.

(c) Append `U`, if the string ends in `I`.

(d) Delete `UU` from the string.



An Intuition for **RE**

- Let's consider the *generalized MU puzzle*:

Given a string *w*, can you transform it into `MU` using the four rules?

- ***Claim:*** We can build a computer program that, given any string *w*, will report "yes" if *w* can be converted into `MU`.

```
int main() {
    string w = getInput();
    queue<string> configs;
    configs.enqueue(w);

    while (!configs.isEmpty()) {
        string curr = configs.dequeue();
        if (curr == “MU”) return true;

        if (curr starts with 'M') {
            curr.enqueue(doubleContentsAfterM(curr));
        }
        for (each copy of III in curr) {
            curr.enqueue(replace the III with U);
        }
        if (curr ends with 'I') {
            curr.enqueue(curr + “U”);
        }
        for (each copy of UU in curr) {
            curr.enqueue(delete that UU);
        }
    }
    return false;
}
```

An Intuition for **RE**

- Many problems in **RE** can be solved by *searching* for a solution:
  - Try all possible combinations of moves in a puzzle.
  - Try all possible strings to see if any of them have some property.
- In other words, the TM needs to both *search* for answers and *verify* whether those answers work.
- This leads to a new perspective on the **RE** languages.

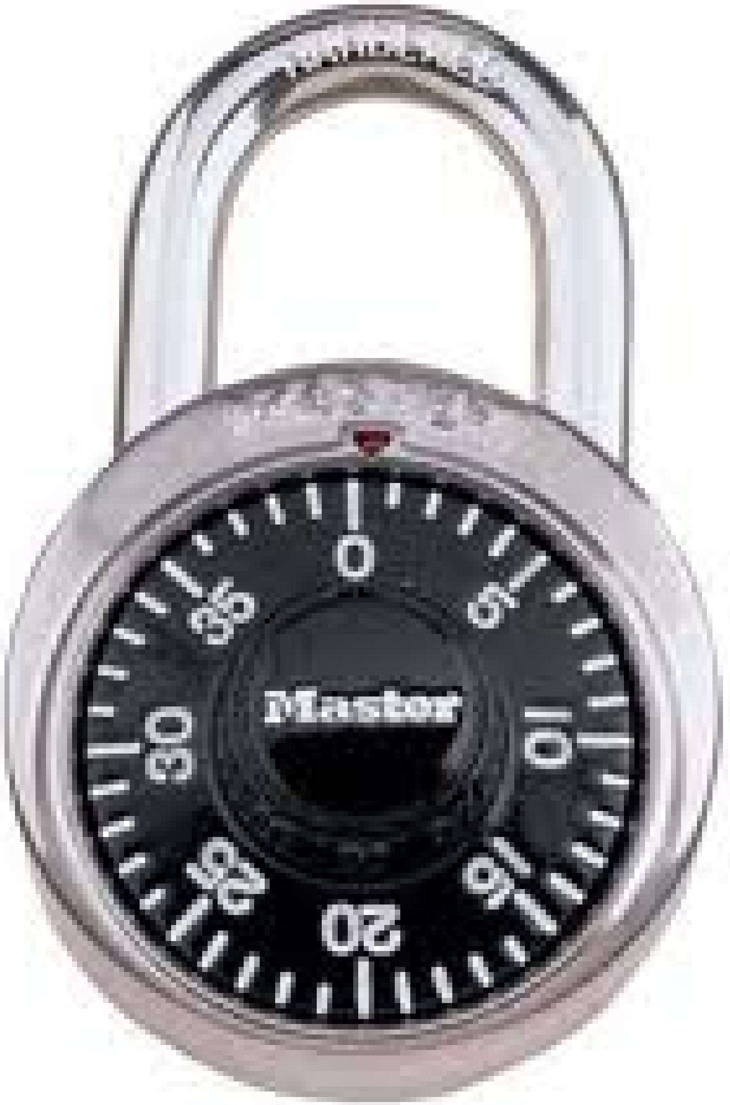
Verifiers

- A ***verifier*** for a language L is a TM V with the following properties:
  - V is a decider (that is, V halts on all inputs.)
  - For any string $w \in \Sigma^*$, the following is true:

$$w \in L \leftrightarrow \exists c \in \Sigma^*. \ V \text{ accepts } \langle w, c \rangle$$

- Intuitively, what does this mean?

Intuiting Verifiers



Question:
Can this lock be opened?

Verifiers

- A ***verifier*** for a language L is a TM V with the following properties:
  - V is a decider (that is, V halts on all inputs.)
  - For any string $w \in \Sigma^*$, the following is true:

$$\boldsymbol{w \in L \quad \leftrightarrow \quad \exists c \in \Sigma^*.\ V \text{ accepts } \langle w, c \rangle}$$

- Some notes about V:
  - If V accepts $\langle w, c \rangle$, then we're guaranteed $w \in L$.
  - If V does not accept $\langle w, c \rangle$, then either
    - $w \in L$, but you gave the wrong c, or
    - $w \notin L$, so no possible c will work.

Verifiers

- A ***verifier*** for a language L is a TM V with the following properties:
  - V is a decider (that is, V halts on all inputs.)
  - For any string $w \in \Sigma^*$, the following is true:

$$\boldsymbol{w \in L \quad \leftrightarrow \quad \exists c \in \Sigma^*.\ V \textbf{ accepts } \langle w, c \rangle}$$

- Some notes about V:
  - If $w \in L$, a string c for which V accepts $\langle w, c \rangle$ is called a ***certificate*** for w.
  - V is required to halt, so given any potential certificate c for w, you can check whether the certificate is correct.

Verifiers

- A ***verifier*** for a language L is a TM V with the following properties:
  - V is a decider (that is, V halts on all inputs.)
  - For any string $w \in \Sigma^*$, the following is true:

$$\boldsymbol{w \in L \quad \leftrightarrow \quad \exists c \in \Sigma^*.\ V \textbf{ accepts } \langle w, c \rangle}$$

- Some notes about V:
  - Notice that $\mathscr{L}(V) \neq L$. Instead:

$$\boldsymbol{\mathscr{L}(V) = \{\ \langle w, c \rangle \mid w \in L \textbf{ and } c \textbf{ is a certificate for } w\ \}}$$

  - The job of V is just to check certificates, not to decide membership in L.

Some Verifiers

- Let L be the following language:

  $L = \{ \langle G, w \rangle \mid G \text{ is a CFG that generates } w \}$

- Let's see how to build a verifier for L.
- A certificate for a grammar G string w should convince us that G accepts w. What kind of information would help us with that?
- One option: Let the certificate be a possible derivation of w from the start symbol.
- Our verifier then just needs to check whether the derivation is valid.

Some Verifiers

- Let L be the following language:

$$L = \{ \langle G, w \rangle \mid G \text{ is a CFG that generates } w \}$$

- Here is one possible verifier for L:

> V = "On input $\langle G, w, c \rangle$, where G is a CFG:
>
> Check whether c is a valid derivation of w from the start symbol of G.
>
> If so, accept. If not, reject."

- If the certificate is a correct derivation, we know for a fact that G can generate w.
- If not, we can't tell whether we got a bad certificate or whether G doesn't generate w.

Some Verifiers

- Let L be the following language:

$$L = \{ \langle n \rangle \mid n \in \mathbb{N} \text{ and the hailstone sequence terminates for } n \}$$

- Let's see how to build a verifier for L.
- A certificate for $\langle n \rangle$ should convince us that the hailstone sequence terminates for n. A bad certificate shouldn't leave us running forever.
- A thought: if the hailstone sequence terminates for n, then it has to terminate in some number of steps.
- Let the certificate be that number of steps.

Some Verifiers

- Let L be the following language:

$$L = \{ \langle n \rangle \mid n \in \mathbb{N} \text{ and the hailstone sequence terminates for } n \}$$

> V = "On input $\langle n, k \rangle$, where $n, k \in \mathbb{N}$.
> Check that $n \neq 0$.
> Run the hailstone sequence, starting at n, for at most k steps.
> If after k steps we reach 1, accept.
> Otherwise, reject."

- Do you see why $\langle n \rangle \in L$ iff there is some k such that V accepts $\langle n, k \rangle$?

What languages are verifiable?