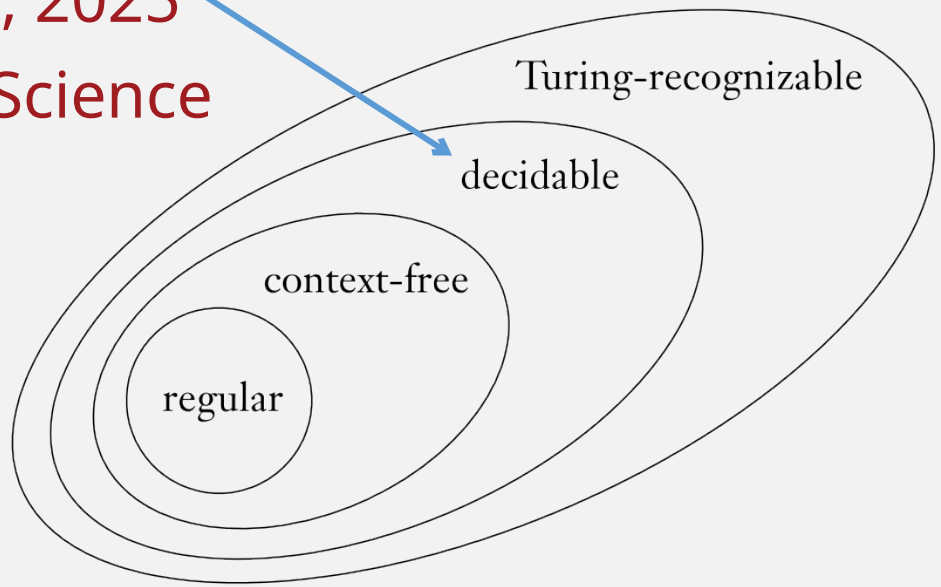
CS 420 / CS 620

Decidability

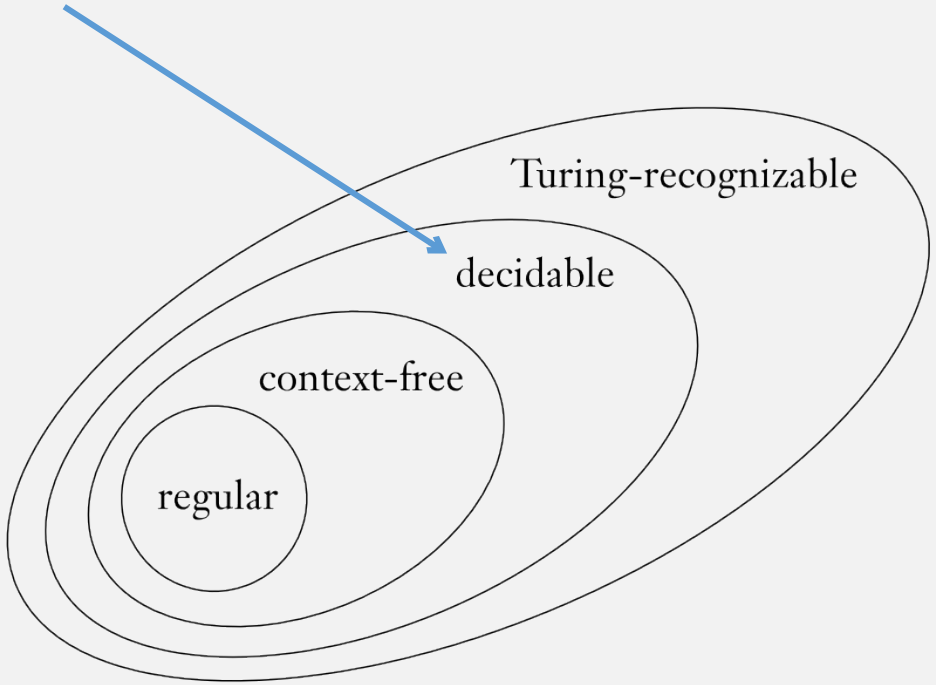
Wednesday, November 5, 2025

UMass Boston Computer Science



Announcements

- HW 9
  - Out: Mon 11/3 12pm (noon)
  - Due: Mon 11/10 12pm (noon)



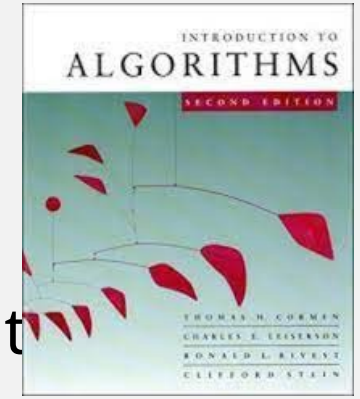
Previously: Turing Machines and Algorithms

- Turing Machines can express more "computation" (than other prev machines)
  - Analogy: a TM models a (`Python`, `Java`) program (`function`)!

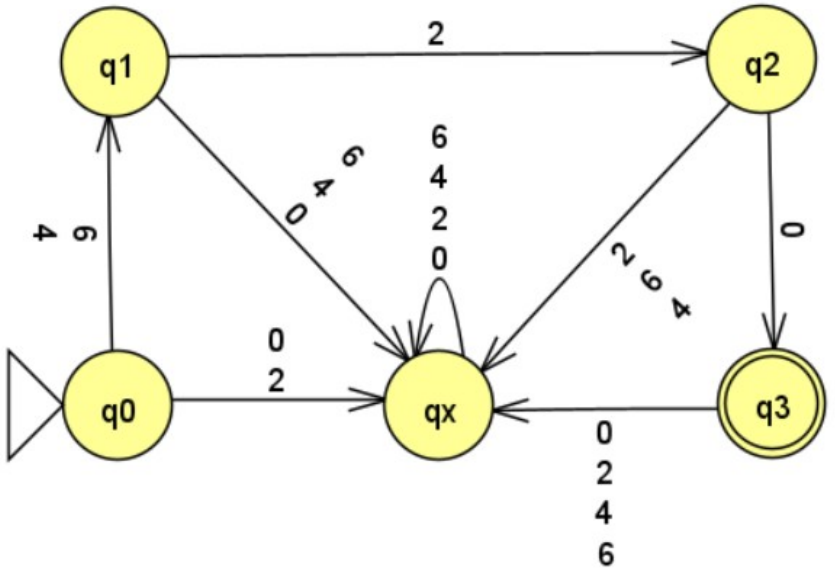
- 2 classes of Turing Machines
  - Recognizers: may loop forever
  - Deciders: always halt

Today

- Deciders = Algorithms
  - I.e., an algorithm is a program that (for any input) always halt



Flashback: HW 1, Problem 2



1. Come up with 2 strings that are accepted by the DFA. These strings are said to be "in the **language**" **recognized** by the DFA.

2. Come up with 2 strings that are not accepted (rejected) by the DFA. These strings are "not in the language" recognized by the DFA.

Figuring out this HW problem (about a DFA's computation) … is itself (meta) computation!

~~language~~

What "kind" of computation is it?

Could you write a program (function) to compute this algorithm?

A function: `DFAaccepts`(B, w) "returns" TRUE if DFA B=$(Q,\Sigma,\delta_B,q_0,F)$ accepts string w

1) `Define` "current" state $q_{current}$ = start state q_0
2) `For each` input char a_i … `in` w
   a) `Define` $q_{next} = \delta_B(q_{current}, a_i)$ "get δ_B"
   b) `Set` $q_{current} = q_{next}$
3) `Return` TRUE `if` $q_{current}$ is in set F

Your task: "compute" how a DFA computes

This is computation about compuation: whether DFA B's computation with input w

The language of DFAaccepts

$$A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$$

Def: a language is a set of strings

How is this a set of strings???

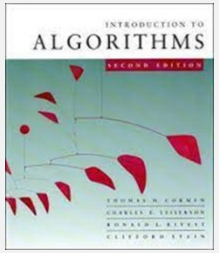
A function: DFAaccepts(B, w) "returns" TRUE if DFA B accepts string w

Interlude: Encoding Things into Strings

Definition: A language's elements / (Turing) machine's input is always a string

Problem: We sometimes want: TM's (program's) input to be "something else" …

- set, graph, DFA, …?

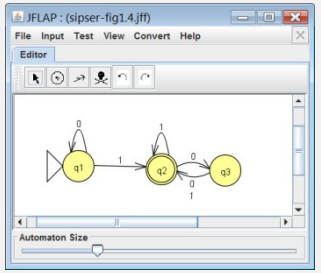


Solution: allow encoding "other kinds of input" as a string

Notation: <SOMETHING> = string encoding for SOMETHING

- A tuple combines multiple encodings, e.g., $\langle B, w \rangle$ (from prev slide)

Example: Possible string encoding for a DFA?





```
<automaton>
  <!--The list of states.-->
  <state name="q1"><initial/></state>
  <state name="q2"><final/></state>
  <state name="q3"></state>
  <!--The list of transitions.-->
  <transition>
    <from>0</from>
    <to>0</to>
    <read>0</read>
  </transition>
  <transition>
    <from>1</from>
```

Details don't matter! (In this class) Can assume some encoding is always possible

Or:

$$(Q, \Sigma, \delta, q_0, F)$$

(written as string)

Interlude: High-Level TMs and Encodings

A high-level TM description, when it uses encoded input:

1. Needs to say the type of its input
   - E.g., graph, DFA, etc.

M = "On input $\langle B, w \rangle$, where B is a DFA and w is a string:

2. Doesn't need to say how input string is encoded

e.g.,

- Assume 1: input is a valid encoding
  - Invalid encodings implicitly rejected
- Assume 2: TM knows how to parse ar

Definition of TM M can $B = (Q, \Sigma, \delta, q_0, F)$

Details don't matter! (In this class) Can assume some encoding is always possible

Implicit "getters"

DFAaccepts as a TM recognizing A_{DFA}

Remember:
TM ~ program (function)
Creating TM ~ programming

$$A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$$

Previously

A function: DFAaccepts(B,w)
"returns" TRUE if DFA B accepts string w

1) Define "current" state q_{current} = start state q_0
2) For each input char a_i ... in w
   a) Define $q_{\text{next}} = \delta(q_{\text{current}}, a_i)$
   b) Set $q_{\text{current}} = q_{\text{next}}$
3) Return TRUE if q_{current} is accept state in F

TM M_{DFA} =

"On inp ... s a DFA and w is a string:

Definition of TM M can assume:

$B = (Q, \Sigma, \delta, q_0, F)$

Implicit "getters"

1) Define "current" state q_{current} = start state q_0
2) For each input char a_i ... in w
   a) Define $q_{\text{next}} = \delta(q_{\text{current}}, a_i)$
   b) Set $q_{\text{current}} = q_{\text{next}}$
3) Accept if q_{current} is an accept state in F

The language of DFAaccepts

language

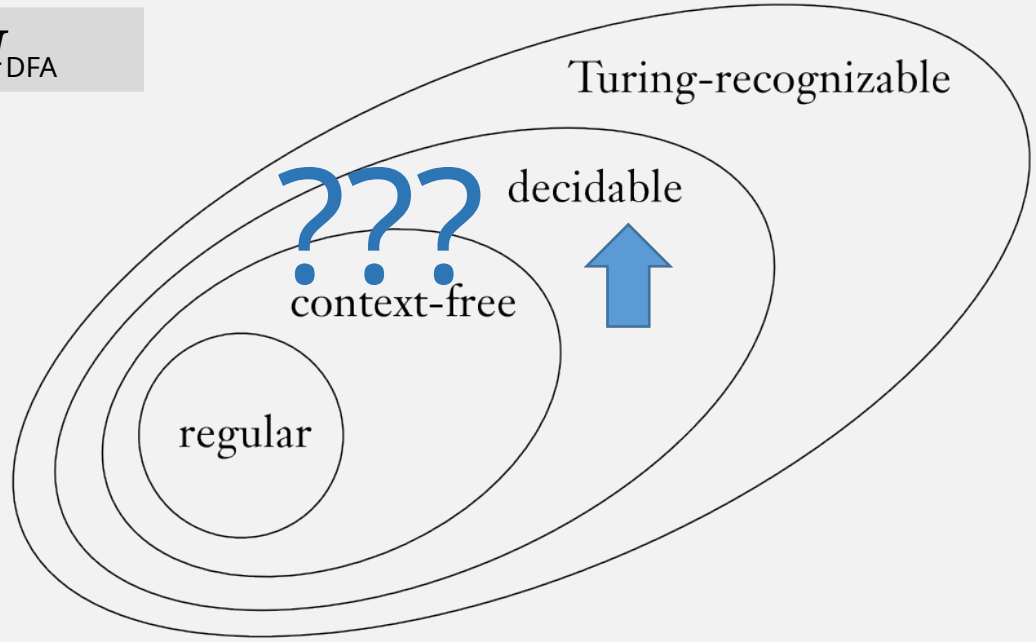
What "kind" of computation is it?

$$A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$$

TM M_{DFA}

- A_{DFA} has a Turing machine
- Is the TM a decider or recognizer?
  - I.e., is it an algorithm?
- To show it's an algo, need to prove:

A_{DFA} is a decidable language



How to prove that a language is decidable?

How to prove that a language is decidable?

Statements

1. If a decider decides a lang L,
   then L is a decidable lang

Key step

Define decider M = On input w … ,
M decides L

3. L is a decidable language

Justifications

1. Definition of decidable langs

2. See M TM def, and Examples Table

3. By statements #1 and #2

How to Design Deciders

- A Decider is a TM …
  - See previous slides on how to:
    - write a high-level TM description
    - Express encoded input strings
  - E.g., M = On input <B, w>, where B is a DFA and w is a string: …

- A Decider is a TM … that must always halt
  - Can only: accept or reject
  - Cannot: go into an infinite loop

- So a Decider definition must include: an extra termination argument:
  - Explains how every step in the TM halts
  - (Pay special attention to loops)

- Remember our analogy: TMs ~ Programs … so *Creating* a TM ~ Programm*ing*
  - To design a TM, think of how to write a program (function) that does what you want

Thm: A_{DFA} is a decidable language

$$A_{DFA} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$$

Key step

Decider for A_{DFA}

Decider input must match (encodings of) strings in the language!

M = "On input $\langle B, w\rangle$, where B is a DFA and w is a string:

1. Simulate B on input w.
2. If the simulation ends in an accept state, *accept*. If it ends in a nonaccepting state, *reject*."

"Calling" the DFA (with an input argument)

Where "Simulate" =

- Define "current" state $q_{current}$ = start state q_0
- For each input char x in w ...
  - Define $q_{next} = \delta(q_{current}, x)$
  - Set $q_{current} = q_{next}$

(meta) Compute how the DFA would compute (with input w)

Remember:
TM ~ program
Creating TM ~ programming

Thm: A_{DFA} is a decidable language

$$A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$$

Decider for A_{DFA}

NOTE: A TM must declare "function" parameters and types … (don't forget it)

$M =$ Undeclared parameters can't be used! (This TM is now invalid because B, w are undefined!)

2. If the simulation ends in an accept state, *accept*. If it ends in a nonaccepting state, *reject*."

Termination Argument

Thm: A_{DFA} is a decidable language

$$A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$$

Decider for A_{DFA}

M = "On input $\langle B, w\rangle$, where B is a DFA and w is a string:

1. Simulate B on input w.
2. If the simulation ends in an accept state, *accept*. If it ends in a nonaccepting state, *reject*."

Where "Simulate" =

- Define "current" state q_{current} = start state q_0
- For each input char x in w …
  - Define $q_{\text{next}} = \delta(q_{\text{current}}, x)$
  - Set $q_{\text{current}} = q_{\text{next}}$

Termination Argument: Step #1 always halts because: the simulation input is always finite, so the loop has finite

Deciders must have a termination argument:
Explains how every step in the TM halts (we typically only care about loops)

Termination Argument

Thm: A_{DFA} is a decidable language

$$A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$$

Decider for A_{DFA}

M = "On input $\langle B, w\rangle$, where B is a DFA and w is a string:

1. Simulate B on input w.
2. If the simulation ends in an accept state, *accept*. If it ends in a nonaccepting state, *reject*."



Termination Argument: Step #2 always halts because:
determining accept requires chec

Deciders must have a termination argument:
Explains how every step in the TM halts (we typically only care about loops)

Correctness / Examples Table

Thm: A_{DFA} is a decidable language

$$A_{DFA} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$$

Decider for A_{DFA}

M = "On input $\langle B, w\rangle$, where B is a DFA and w is a string:

1. Simulate B on input w.
2. If the simulation ends in an accept state, *accept*. If it ends in a nonaccepting state, *reject*."

(New for TMs) column(s) for "called" machines

"Actual" behavior

"Expected" behavior

Example Str	B on input w?	M?	In A_{DFA} lang?
$\langle D_1, w_1\rangle$	Accept	Accept	Yes
$\langle D_2, w_2\rangle$	Reject	Reject	No

(especially important when machine could loop)

Let:
- D_1 = DFA, accepts w_1
- D_2 = DFA, rejects w_2

Columns must match!

Thm: A_{NFA} is a decidable language

$$A_{\text{NFA}} = \{\langle B, w\rangle \mid B \text{ is an NFA that accepts input string } w\}$$

Decider for A_{NFA} :

Decider input must match (encodings of) strings in the language!

N = "On input $\langle B, w\rangle$, where B is an NFA and w is a string:

1. Convert NFA B to an equivalent DFA C, using the procedure for NFA→DFA ???
2. Run TM M on input $\langle C, w\rangle$.
3. If M accepts, *accept*; otherwise, *reject*."

Flashback: NFA→DFA

Have: $N = (Q, \Sigma, \delta, q_0, F)$

Want to: construct a DFA $M = (Q', \Sigma, \delta', q_0', F')$

1. $Q' = \mathcal{P}(Q)$.

2. For $R \in Q'$ and $a \in \Sigma$,

$$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a)$$

3. $q_0' = \{q_0\}$

4. $F' = \{R \in Q' \mid R \text{ contains an accept state of } N\}$

This conversion is computation!

So it can be computed by a (decider?) Turing Machine

Turing Machine ack: NFA→DFA

New TM Variation!
Doesn't accept or reject,
Just writes "output" to tape

TM NFA→DFA = On input <N>, where N is an NFA and $N = (Q, \Sigma, \delta, q_0, F)$

1. Write to the tape: DFA $M = (Q', \Sigma, \delta', q_0', F')$

Where: $Q' = \mathcal{P}(Q)$.

For $R \in Q'$ and $a \in \Sigma$,

$$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a)$$

$$q_0' = \{q_0\}$$

$$F' = \{R \in Q' \mid R \text{ contains an acce}$$

Why is this guaranteed to halt?

Because a DFA description has only finite parts (finite states, finite transitions, etc)

So any loop iteration over them is finite

Thm: A_{NFA} is a decidable language

$$A_{\text{NFA}} = \{\langle B, w\rangle \mid B \text{ is an NFA that accepts input string } w\}$$

Decider for A_{NFA} :

N = "On input $\langle B, w\rangle$, where B is an NFA and w is a string:

1. Convert NFA B to an equivalent DFA C, using the procedure NFA→DFA
2. Run TM M on input $\langle C, w\rangle$. (M is the A_{DFA} decider from prev slide)
3. If M accepts, *accept*; otherwise, *reject*."

Remember:
TM ~ program
Creating TM ~ programming
Previous theorems ~ library

"Calling" another TM. Must give correct arg

New capability: TM can check tape of another TM after calling it

Termination argument: This is a decider (i.e., it always halts) because:

- Step 1 always halts bc: NFA->DFA is decider (finite number of NFA states)
- Step 2 always halts because: M is a decider (prev A_{DFA} thm)

How to Design Deciders, Part 2

Hint:

- Previous theorems / constructions are a “library” of reusable TMs
- When creating a TM, use this “library” to help you!
  - Just like libraries are useful when programming!
- E.g., “Library” for DFAs:
  - NFA→DFA, RegExpr→NFA
  - UNION, CONCAT, STAR, COMPLEMENT, reverse
  - Deciders for: A_{DFA}, A_{NFA}, A_{REX}, …

Thm: A_{REX} is a decidable language

$$A_{\text{REX}} = \{\langle R, w\rangle \mid R \text{ is a regular expression that generates string } w\}$$

Decider:

NOTE: A TM must declare "function" parameters and types … (don't forget it)

P = "On input $\langle R, w\rangle$, where R is a regular expression and w is a string:

1. Convert regular expression R to an equivalent NFA A by using the procedure `RegExpr→NFA`

… which can be used (properly!) in the TM description

Remember:
TMs ~ programs
Creating TM ~ programming
Previous theorems ~ library

lashback

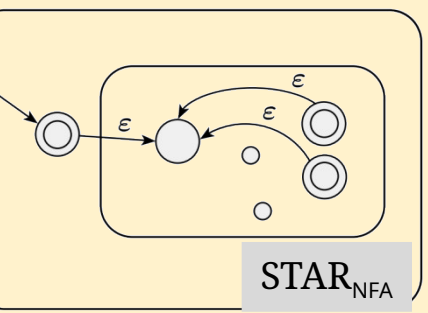
RegExpr→NFA

… so guaranteed to always reach base case(s)

Does this conversion always halt, and why?

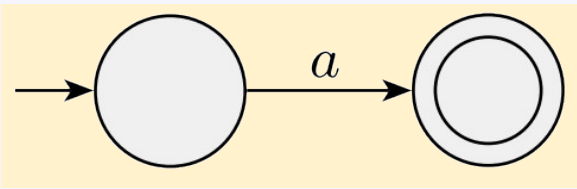
R is a ***regular expression*** if R is

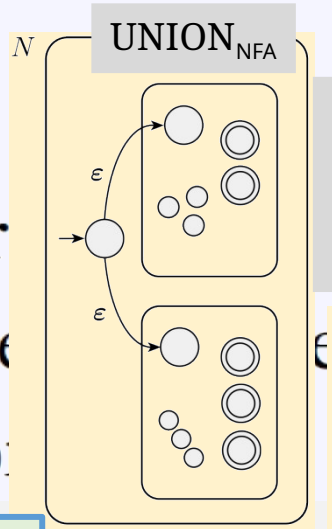
1. a for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 ar
5. $(R_1 \circ R_2)$, where R_1 and R_2 are
6. (R_1^*), where R_1 is a regular exp



STAR$_{NFA}$

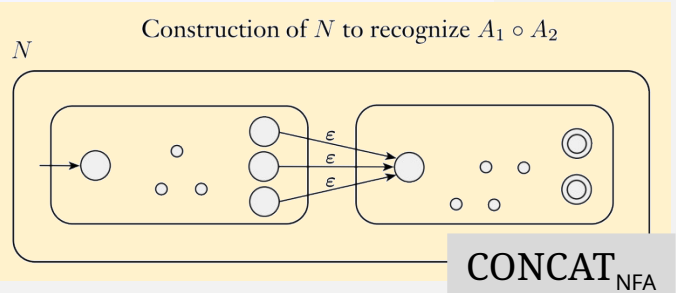
Yes, because recursive call only happens on "smaller" regular expressions …





UNION$_{NFA}$

RegExpr→NFA($R_1 \bigcup R_2$) =
UNION$_{NFA}$(RegExpr→NFA(R_1),
RegExpr→NFA(R_2))



CONCAT$_{NFA}$

Thm: A_{REX} is a decidable language

$$A_{\text{REX}} = \{\langle R, w\rangle \mid R \text{ is a regular expression that generates string } w\}$$

Decider:

$P =$ "On input $\langle R, w\rangle$, where R is a regular expression and w is a string:

1. Convert regular expression R to an equivalent NFA A by using the procedure `RegExpr→NFA`
2. Run TM N on input $\langle A, w\rangle$ (from prev slide)
3. If N accepts, *accept*; if N rejects, *reject*."

When "calling" another TM, must give proper arguments!

Termination Argument: This is a decider because:

- Step 1: always halts because: converting a reg expr to NFA is done recursively, where the reg expr gets smaller at each step, eventually reaching the base case
- Step 2: always halts because: N is a decider

Decidable Languages About DF

Remember:
TMs ~ programs
Creating TM ~ programming
Previous theorems ~ library

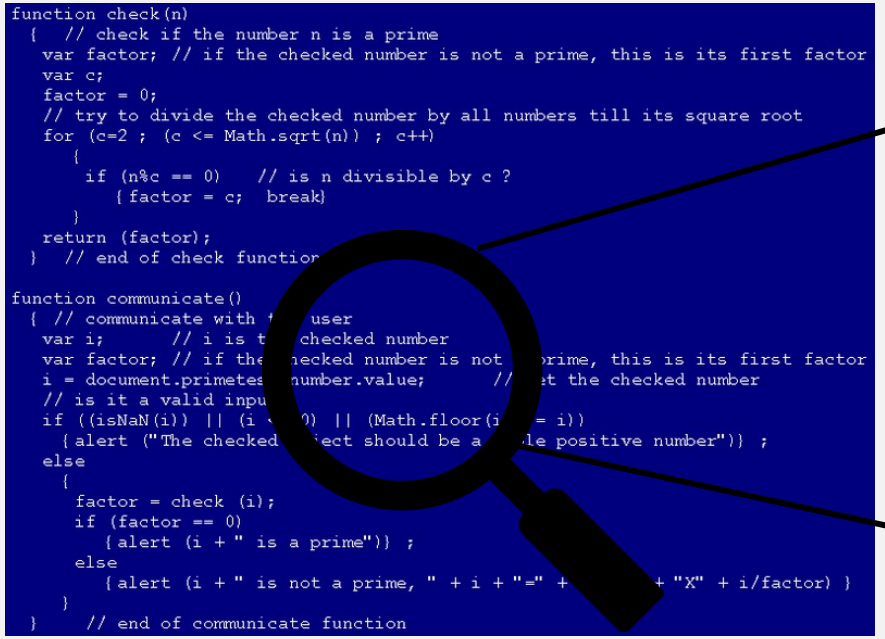
- $A_{\text{DFA}} = \{\langle B, w\rangle |\ B \text{ is a DFA that accepts input string } w\}$
  - Decider TM: implements B DFA's extended δ fn algorithm

- $A_{\text{NFA}} = \{\langle B, w\rangle |\ B \text{ is an NFA that accepts input string } w\}$
  - Decider TM: uses NFA→DFA algorithm + A_{DFA} decider

- $A_{\text{REX}} = \{\langle R, w\rangle |\ R \text{ is a regular expression that generates string } w\}$
  - Decider TM: uses RegExpr→NFA algorithm + A_{NFA} decider

Flashback: Why Study Algorithms About Computing

To predict what programs will do
(without running them!)



```
function check(n)
  {   // check if the number n is a prime
    var factor; // if the checked number is not a prime, this is its first factor
    var c;
    factor = 0;
    // try to divide the checked number by all numbers till its square root
    for (c=2 ; (c <= Math.sqrt(n)) ; c++)
       {
        if (n%c == 0)    // is n divisible by c ?
           {factor = c;  break}
       }
    return (factor);
  }  // end of check functio

function communicate()
  {  // communicate with  user
    var i;       // i is t  checked number
    var factor; // if the ecked number is not  rime, this is its first factor
    i = document.primetes umber.value;       //  t the checked number
    // is it a valid inpu
    if ((isNaN(i)) || (i  0) || (Math.floor(i  = i))
      {alert ("The checked  ject should be a  le positive number")} ;
    else
      {
       factor = check (i);
       if (factor == 0)
          {alert (i + " is a prime")} ;
       else
          {alert (i + " is not a prime, " + i + "=" +  + "X" + i/factor) }
      }
  }     // end of communicate function
```

Not possible for all programs! But ...



???

Predicting What Some Programs Will Do …

What if we: look at simpler computation models … like DFAs and regular languages!

Thm: E_{DFA} is a decidable language

$$E_{\text{DFA}} = \{\langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$$

E_{DFA} is a language … of DFA descriptions, i.e., $(Q, \Sigma, \delta, q_0, F)$ …

… where the language of <u>each</u> DFA … must be { }, i.e., DFA accepts no strings

Is there a decider that accepts/rejects DFA descriptions …

… by predicting something about the DFA's language (by analyzing its description)

<u>Key idea / question we are about to study:</u>
Compute (predict) something about the <u>runtime computation</u> of a program,
by analyzing *only* its <u>source code</u>?

<u>Analogy</u>
DFA's description : a program's source code ::
DFA's language : a program's runtime computation

Important: don't confuse the different languages here!

Thm: E_{DFA} is a decidable language

$$E_{\text{DFA}} = \{\langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$$

Decider:

T = "On input $\langle A \rangle$, where A is a DFA:

1. Mark the start state of A.
2. Repeat until no new states get marked:
3. Mark any state that has a transition coming into it from any state that is already marked.
4. If no accept state is marked, *accept*; otherwise, *reject*."

If loop marks at least 1 state on each iteration, then it eventually terminates because there are finite states; else loop terminates

(meta) Compute how the DFA would compute

I.e., this is a "reachability" algorithm …

… check if accept states are "reachable" from start state

Termination argument?

Note: TM T is doing a new computation on DFAs! (It does not "run" the DFA!)

Instead: compute something about DFA's language (runtime computation) by analyzing its description (source code)

Thm: EQ_{DFA} is a decidable language

$$EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$$

I.e., Can we compute whether two DFAs are "equivalent"?

Replacing "DFA" with "program" = A "holy grail" of computer science!



Thm: EQ_{DFA} is a decidable language

$$EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$$

~~(meta) Compute how the DFA would compute i.e., run them~~

I.e., Can we compute whether two DFAs are "equivalent"?

A Naïve Attempt (assume alphabet {a}):

1. Simulate:
   - A with input a, and
   - B with input a
   - Reject if results are different, else …
2. Simulate :
   - A with input aa, and
   - B with input aa
   - Reject if results are different, else …
- …

This might not terminate! (Hence it's not a decider)

Key idea

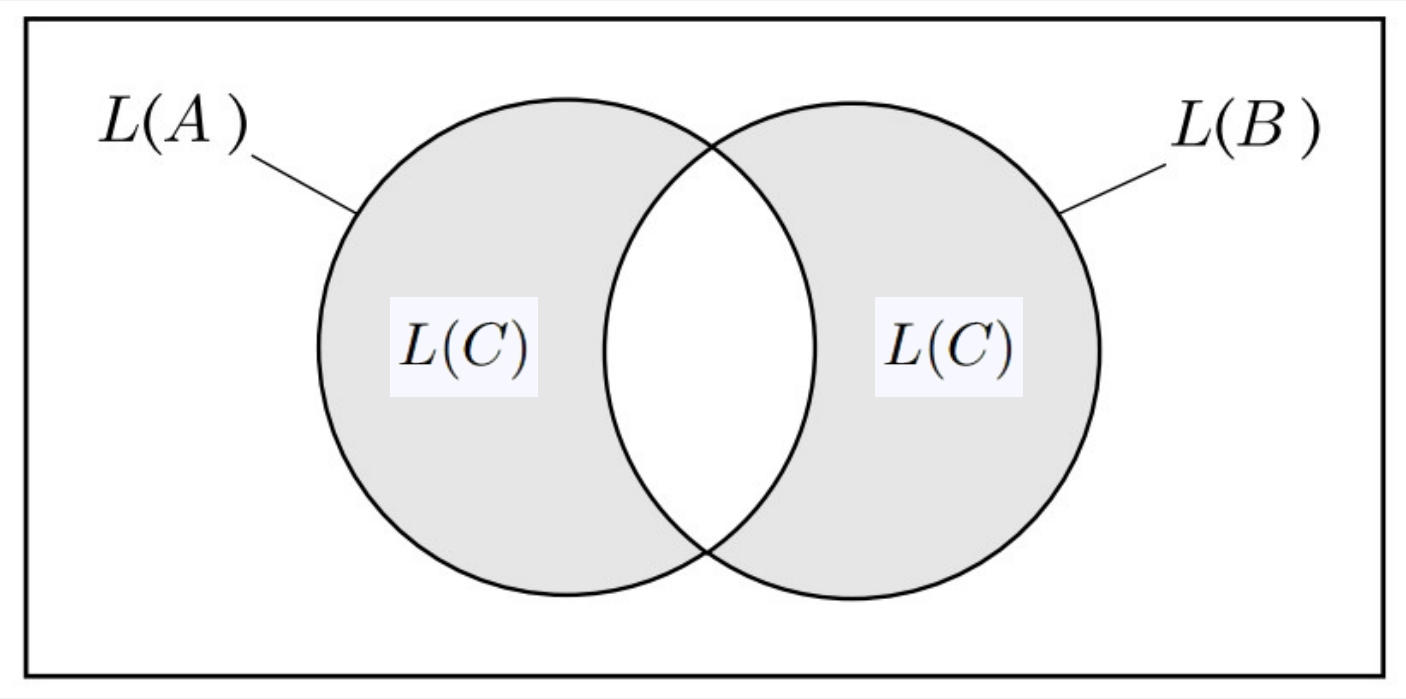
Can we compute this without running the DFAs, i.e., by only examining the DFA's "source code"?

Thm: EQ_{DFA} is a decidable language

$$EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$$

Trick: Use Symmetric Difference

Symmetric Difference



$$L(C) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(\overline{L(A)} \cap L(B)\right)$$

$$\boxed{L(C) = \emptyset \text{ iff } L(A) = L(B)}$$

Thm: EQ_{DFA} is a decidable language

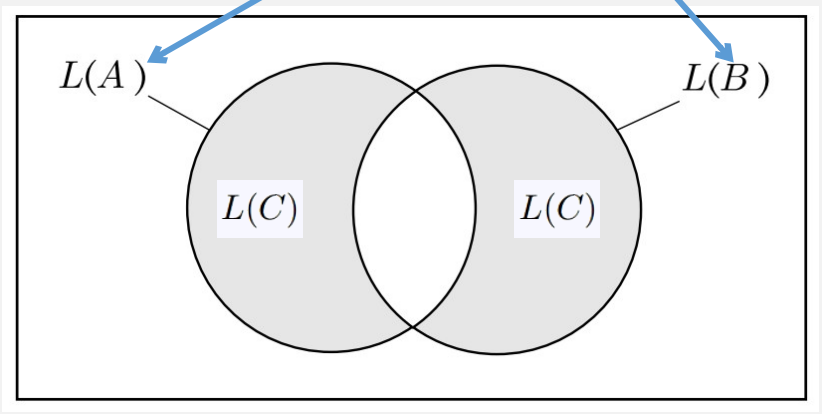
$$EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$$

(proved in prev hws!)

NOTE, This only works because:
regular langs closed under negation,
i.e., set complement, union and intersection

Construct decider using 2 parts:

1. Symmetric Difference algo: $L(C) = \left(L(A) \cap L(B)\right) \cup \left(L(A) \cap L(B)\right)$
   - Construct C = Union, intersection, negation of machines A and B
2. Decider T (from "library") for: $E_{\text{DFA}} = \{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$
   - Because $L(C) = \emptyset$ iff $L(A) = L(B)$



Thm: EQ_{DFA} is a decidable language

$$EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$$

TM input must use same string encoding as lang

Construct decider using 2 parts:

1. Symmetric Difference algo: $L(C) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(\overline{L(A)} \cap L(B)\right)$
   - Construct C = Union, intersection, negation of machines A and B
2. Decider T (from "library") for: $E_{\text{DFA}} = \{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$
   - Because $L(C) = \emptyset$ iff $L(A) = L(B)$

F = "On input $\langle A, B\rangle$, where A and B are DFAs:

1. Construct DFA C as described.
2. Run TM T deciding E_{DFA} on input $\langle C\rangle$.
3. If T accepts, *accept*. If T rejects, *reject*."

Termination argument?

om Lecture 1

Predicting What Some Programs Will Do

microsoft.com/en-us/research/project/slam/

SLAM is a project for checking that software satisfies critical behavioral properties of the interfaces it uses and to aid software engineers in designing interfaces and software that ensure reliable and correct functioning. Static Driver Verifier is a tool in the Windows Driver Development Kit that uses the SLAM verification engine.



"Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we're building tools that can do actual proof about the software and how it works in order to guarantee the reliability." **Bill Gates, April 18, 2002.** Keynote address at WinHec 2002





Static Driver Verifier Research Platform README

Overview of Static Driver Verifier Research Platform

Static Driver Verifier (SDV) is a compile-time static verification
Research Platform (SDVRP) is an extension to SDV that allows

- Support additional frameworks (or APIs) and write custo
- Experiment with the model checking step.

Model checking

From Wikipedia, the free encyclopedia

In computer science, **model checking** or **property checking** is a method for checking whether a finite-state model of a system meets a given specification (also known as correctness). This is typically

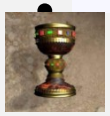
Its "language"

Summary: Algorithms About Regular Langs

- $A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$
  - Decider: Simulates DFA by implementing extended δ function
- $A_{\text{NFA}} = \{\langle B, w\rangle \mid B \text{ is an NFA that accepts input string } w\}$
  - Decider: Uses NFA→DFA decider + A_{DFA} decider
- $A_{\text{REX}} = \{\langle R, w\rangle \mid R \text{ is a regular expression that generates string } w\}$
  - Decider: Uses RegExpr→NFA decider + A_{NFA} decider
- $E_{\text{DFA}} = \{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$
  - Decider: Reachability algorithm

Lang of the DFA

- $EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$



  - Decider: Uses complement and intersection closure construction + E_{DFA} decider

Remember:
TMs ~ programs
Creating TM ~ programming
Previous theorems ~ library

Next: Algorithms (Decider TMs) for CFLs?

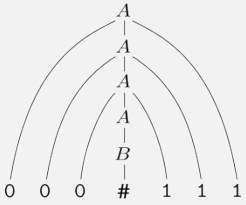
- What can we predict about CFGs or PDAs?

Thm: A_{CFG} is a decidable language

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

- This is a very practically important problem …
- … equivalent to:
  - Algorithm determining: possible to parse "program" w for a programming language with grammar G?

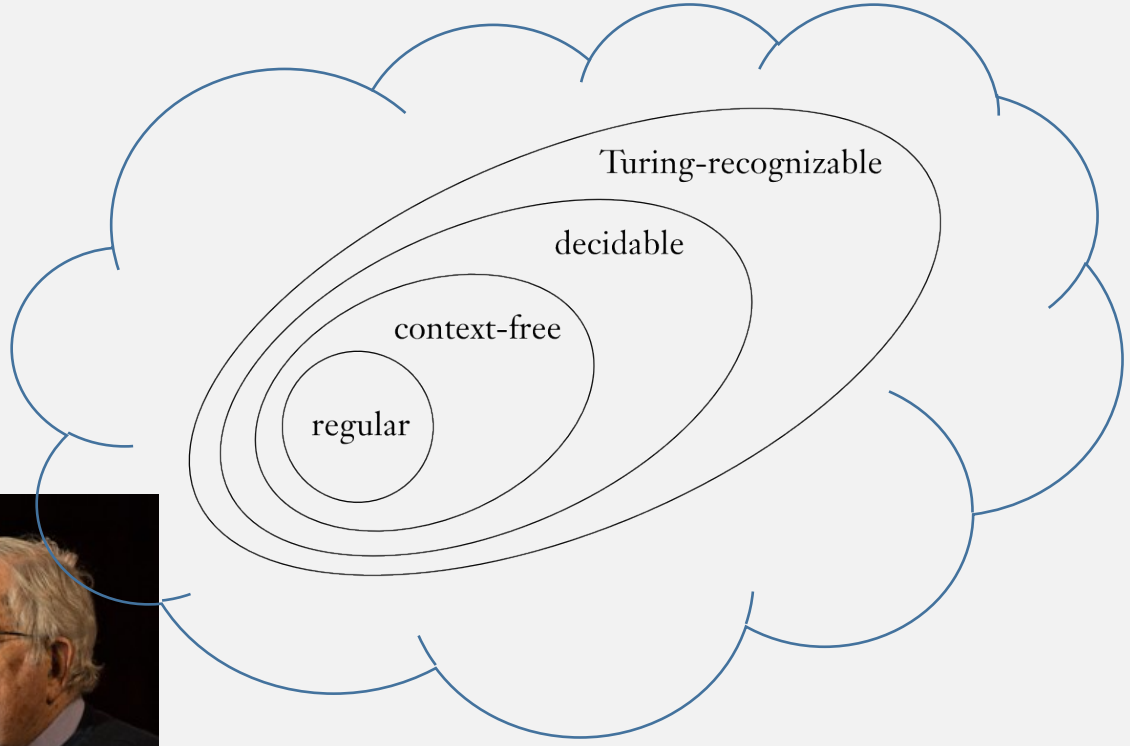
- A Decider for this problem could … ?
  - Try every possible derivation of G, and check if it's equal to w?
  - But this might never halt
    - E.g., what if there are rules like: $\mathbf{S} \to \mathtt{0}\mathbf{S}$ or $\mathbf{S} \to \mathbf{S}$
  - This TM would be a recognizer but not a decider



Idea: can the TM stop checking after some length?
- I.e., Is there upper bound on the number of derivation steps?

Chomsky Normal Form

Noam Chomsky





He came up with this hierarchy of languages

Chomsky Normal Form

A context-free grammar is in ***Chomsky normal form*** if every rule is of the form

$$A \to BC$$
$$A \to a$$

(non-start) Variables only

2 rule shapes

Terminals only

where a is any terminal and A, B, and C are any variables—except that B and C may not be the start variable. In addition, we permit the rule $S \to \varepsilon$, where S is the start variable.

Chomsky Normal Form Example

- $S \rightarrow AB$
- $B \rightarrow AB$
- $A \rightarrow$ a
- $B \rightarrow$ b

Makes the string long enough

Convert variables to terminals

- To generate string of length: 2
  - Use S rule: 1 time; Use A or B rules: 2 times
  - $S \Rightarrow AB \Rightarrow aB \Rightarrow ab$
  - Derivation total steps: 1 + 2 = 3
- To generate string of length: 3
  - Use S rule: 1 time; A rule: 1 time; A or B rules: 3 times
  - $S \Rightarrow AB \Rightarrow AAB \Rightarrow aAB \Rightarrow aaB \Rightarrow aab$
  - Derivation total steps: 1 + 1 + 3 = 5
- To generate string of length: 4
  - Use S rule: 1 time ; A rule: 2 times; A or B rules: 4 times
  - $S \Rightarrow AB \Rightarrow AAB \Rightarrow AAAB \Rightarrow aAAB \Rightarrow aaAB \Rightarrow aaaB \Rightarrow aaab$
  - Derivation total steps: 3 + 4 = 7
- …

A context-free grammar is in ***Chomsky normal form*** if every rule is of the form

$$A \rightarrow BC$$
$$A \rightarrow a$$

2 rule shapes

where a is any terminal and A, B, and C are any variables—except that B and C may not be the start variable. In addition, we permit the rule $S \rightarrow \epsilon$, where S is the start variable.

Chomsky Normal Form: Number of Steps

To generate a string of length n:

$n - 1$ steps: to generate n variables

Makes the string long enough

\+ n steps: to turn each variable into a termina

Convert string to terminals

Total: $2n - 1$ steps

(A *finite* number of steps!)

Chomsky normal form

$A \rightarrow BC$ Use n-1 times

$A \rightarrow a$ Use n times

Thm: A_{CFG} is a decidable language

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

Proof: create the decider:

$S =$ "On input $\langle G, w\rangle$, where G is a CFG and w is a string:

1. Convert G to an equivalent grammar in Chomsky normal form.
2. List all derivations with $2n - 1$ steps, where n is the length of w; except if $n = 0$, then instead list all derivations with one step.
3. If any of these derivations generate w, *accept*; if not, *reject*."

We first need to prove this is true for all CFGs!

Step 1: Conversion to Chomsky Normal Form is an algorithm
…
Step 2:
Step 3:

Termination argument?

Thm: Every CFG has a Chomsky Normal Form

Proof: Create algorithm to convert any CFG into Chomsky Normal Form

Chomsky normal form

$A \to BC$

$A \to a$

1. Add new start variable S_0 that does not appear on any RHS
   - I.e., add rule $S_0 \to S$, where S is old start var

$$S \to ASA \mid \mathtt{a}B$$
$$A \to B \mid S$$
$$B \to \mathtt{b} \mid \varepsilon$$



$$\boldsymbol{S_0 \to S}$$
$$S \to ASA \mid \mathtt{a}B$$
$$A \to B \mid S$$
$$B \to \mathtt{b} \mid \varepsilon$$

Thm: Every CFG has a Chomsky Normal Form

Chomsky normal form

$A \to BC$

$A \to a$

1. Add new start variable S_0 that does not appear on any RHS
   - I.e., add rule $S_0 \to S$, where S is old start var
2. Remove all "empty" rules of the form $A \to \varepsilon$
   - A must not be the start variable
   - Then for every rule with A on RHS, add new rule with A deleted
     - E.g., If $R \to uAv$ is a rule, add $R \to uv$
   - Must cover all combinations if A appears more than once in a RHS
     - E.g., if $R \to uAvAw$ is a rule, add 3 rules: $R \to uvAw$, $R \to uAvw$, $R \to uvw$

$S_0 \to S$

$S \to ASA \mid \mathtt{a}B \mid \mathbf{a}$

$A \to B \mid S \mid \boldsymbol{\varepsilon}$

$B \to \mathtt{b} \mid \boldsymbol{\varepsilon}$

Then, add

First, remove

$S_0 \to S$

$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \mid \boldsymbol{S}$

$A \to B \mid S \mid \boldsymbol{\varepsilon}$

$B \to \mathtt{b}$

Then add, to account for possibly empty A

Then, remove

Thm: Every CFG has a Chomsky Normal Form

Chomsky normal form

$A \to BC$

$A \to a$

1. Add new start variable S_0 that does not appear on any RHS
   - I.e., add rule $S_0 \to S$, where S is old start var
2. Remove all "empty" rules of the form $A \to \varepsilon$
   - A must not be the start variable
   - Then for every rule with A on RHS, add new rule with A deleted
     - E.g., If $R \to uAv$ is a rule, add $R \to uv$
   - Must cover all combinations if A appears more than once in a RHS
     - E.g., if $R \to uAvAw$ is a rule, add 3 rules: $R \to uvAw$, $R \to uAvw$, $R \to uvw$
3. Remove all "unit" rules of the form $A \to B$
   - Then, for every rule $B \to u$, add rule $A \to u$

$S_0 \to S$
$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \mid S$
$A \to B \mid S$
$B \to \mathtt{b}$

Remove, no add (same variable)

$S_0 \to S \mid \boldsymbol{ASA} \mid \mathbf{a}\boldsymbol{B} \mid \mathbf{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS}$
$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$
$A \to B \mid S$
$B \to \mathtt{b}$

Remove, then add S RHSs to S_0

$S_0 \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$
$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$
$A \to S \mid \mathtt{b} \mid \boldsymbol{ASA} \mid \mathbf{a}\boldsymbol{B} \mid \mathbf{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS}$
$B \to \mathtt{b}$

Remove, then add S RHSs to A

Termination argument of this algorithm?

Thm: Every CFG has a Chomsky Normal Form

Chomsky normal form

$A \to BC$

$A \to a$

1. Add new start variable S_0 that does not appear on any RHS
   - I.e., add rule $S_0 \to S$, where S is old start var
2. Remove all "empty" rules of the form $A \to \varepsilon$
   - A must not be the start variable
   - Then for every rule with A on RHS, add new rule with A deleted
     - E.g., If $R \to uAv$ is a rule, add $R \to uv$
   - Must cover all combinations if A appears more than once in a RHS
     - E.g., if $R \to uAvAw$ is a rule, add 3 rules: $R \to uvAw$, $R \to uAvw$, $R \to uvw$
3. Remove all "unit" rules of the form $A \to B$
   - Then, for every rule $B \to u$, add rule $A \to u$
4. Split up rules with RHS longer than length 2
   - E.g., $A \to wxyz$ becomes $A \to wB$, $B \to xC$, $C \to yz$
5. Replace all terminals on RHS with new rule
   - E.g., for above, add $W \to w$, $X \to x$, $Y \to y$, $Z \to z$

$S_0 \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$
$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$
$A \to \mathtt{b} \mid ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$
$B \to \mathtt{b}$



$S_0 \to AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS$
$S \to AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS$
$A \to \mathtt{b} \mid AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS$
$A_1 \to SA$
$U \to \mathtt{a}$
$B \to \mathtt{b}$

Thm: A_{CFG} is a decidable language

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

Proof: create the decider:

$S =$ "On input $\langle G, w\rangle$, where G is a CFG and w is a string:

1. Convert G to an equivalent grammar in Chomsky normal form.
2. List all derivations with $2n - 1$ steps, where n is the length of w; except if $n = 0$, then instead list all derivations with one step.
3. If any of these derivations generate w, *accept*; if not, *reject*."

We first need to prove this is true for all CFGs!

Termination argument:
Step 1: any CFG has only a finite # rules
Step 2: $2n$-1 = finite # of derivations to check
Step 3: checking finite number of derivations

Thm: E_{CFG} is a decidable language

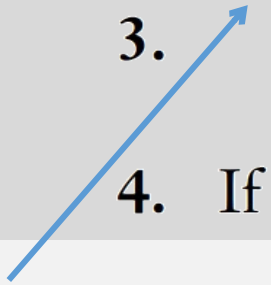
$$E_{\text{CFG}} = \{\langle G \rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}$$

Recall:

$$E_{\text{DFA}} = \{\langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$$

T = "On input $\langle A \rangle$, where A is a DFA:

1. Mark the start state of A.
2. Repeat until no new states get marked:
3. Mark any state that has a transition coming into it from any state that is already marked.
4. If no accept state is marked, *accept*; otherwise, *reject*."



"Reachability" (of accept state from start state) algorithm

Can we compute "reachability" for a CFG?

Thm: E_{CFG} is a decidable language

$$E_{\mathsf{CFG}} = \{\langle G \rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}$$

Proof: create decider that calculates reachability for grammar G

- Go backwards, start from terminals, to avoid getting stuck in looping rules

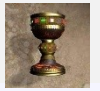
$R =$ "On input $\langle G \rangle$, where G is a CFG:

1. Mark all terminal symbols in G.
2. Repeat until no new variables get marked:
3. Mark any variable A where G has a rule $A \to U_1 U_2 \cdots U_k$ and each symbol $U_1, \ldots, U_k$ has already been marked.
4. If the start variable is not marked, *accept*; otherwise, *reject*."

Loop marks 1 new variable on each iteration or stops: it eventually terminates because there are a finite # of variables

Termination argument?

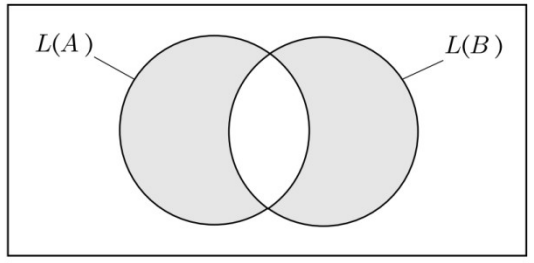
Thm: EQ_{CFG} is a decidable language?



$$EQ_{\text{CFG}} = \{\langle G, H\rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$$

Recall: $EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$

- Used Symmetric Difference



$$L(C) = \emptyset \text{ iff } L(A) = L(B)$$

  - where C = complement, union, intersection of machines A and B
- Can't do this for CFLs!
  - Intersection and complement are not closed for CFLs!!!

Intersection of CFLs is Not Closed!

Proof (by contradiction), Assume intersection is closed for CFLs

- Then intersection of these CFLs should be a CFL:

$$A = \{\mathtt{a}^m \mathtt{b}^n \mathtt{c}^n \mid m, n \geq 0\}$$

$$B = \{\mathtt{a}^n \mathtt{b}^n \mathtt{c}^m \mid m, n \geq 0\}$$

- But $A \cap B = \{\mathtt{a}^n \mathtt{b}^n \mathtt{c}^n \mid n \geq 0\}$

- … which is not a CFL! (So we have a contradiction)

Complement of a CFL is not Closed!

- Assume CFLs closed under complement, then:

$$\text{if } G_1 \text{ and } G_2 \text{ context-free}$$

$$\overline{L(G_1)} \text{ and } \overline{L(G_2)} \text{ context-free}$$

From the assumption

$$\overline{\overline{L(G_1)} \cup \overline{L(G_2)}} \text{ context-free}$$

Union of CFLs is closed

$$\overline{\overline{\overline{L(G_1)} \cup \overline{L(G_2)}}} \text{ context-free}$$

From the assumption

$$L(G_1) \cap L(G_2) \text{ context-free}$$

DeMorgan's Law!

But intersection is not closed for CFLS (prev slide)

Thm: EQ_{CFG} is a decidable language?

$$EQ_{\text{CFG}} = \{\langle G, H \rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$$

- No! 
  - There's no algorithm to decide whether two grammars are equivalent!
- It's not recognizable either! (Can't create any TM to do this!!!)
  - (details later)
- I.e., this is an impossible computation!

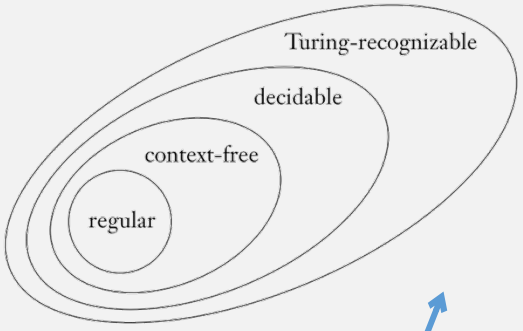
Summary Algorithms About CFLs

- $A_{\text{CFG}} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates string } w\}$
  - Decider: Convert grammar to Chomsky Normal Form
  - Then check all possible derivations up to length $2|w|$ - 1 steps

- $E_{\text{CFG}} = \{\langle G\rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}$
  - Decider: Compute “reachability” of start variable from terminals

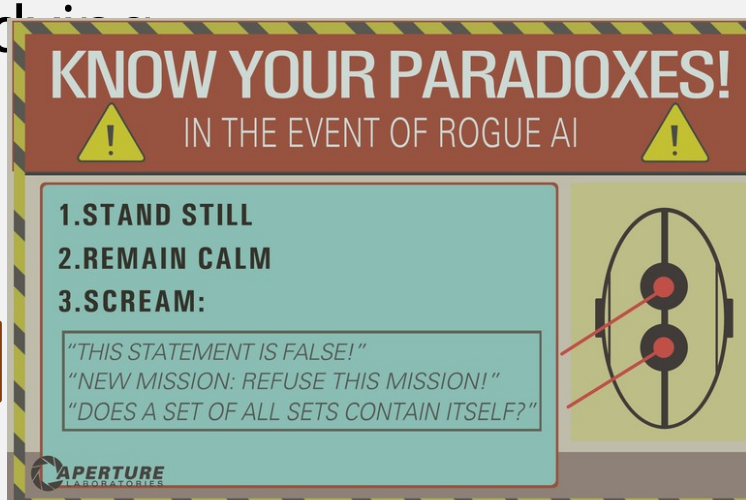
- $EQ_{\text{CFG}} = \{\langle G, H\rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$
  - We couldn’t prove that this is decidable!
  - (So you cant use this theorem when creating another decider)

The Limits of Turing Machines?



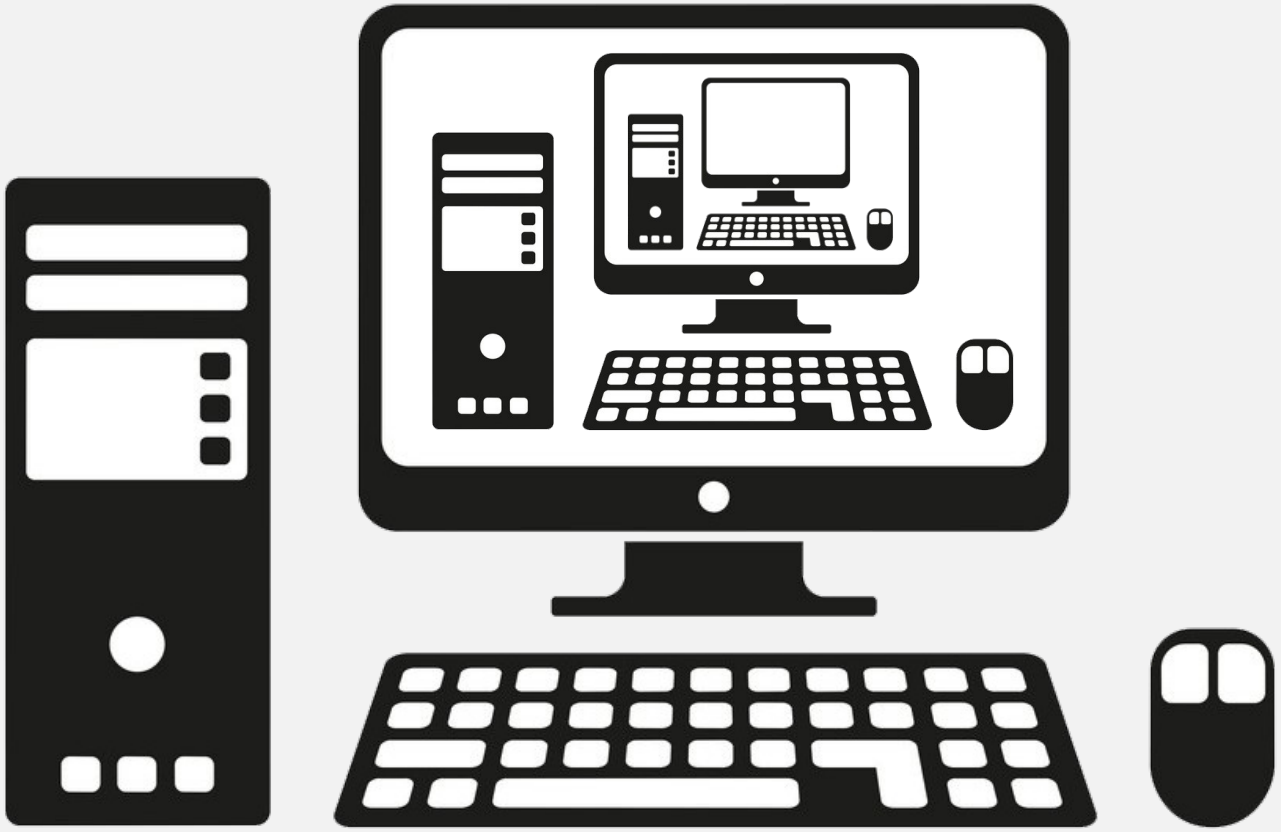
- TMs represent all possible “computations”
  - I.e., any (Python, Java, …) program you write is a TM
- But some things are not computable? I.e., some langs are out here ?
- To explore the limits of computation, we have been stud… … computation about other computation …
  - Thought: Is there a decider (algorithm) to determine whether a TM is an decider?

Hmmm, this doesn’t feel right …



Next time: Is A_{TM} decidable?

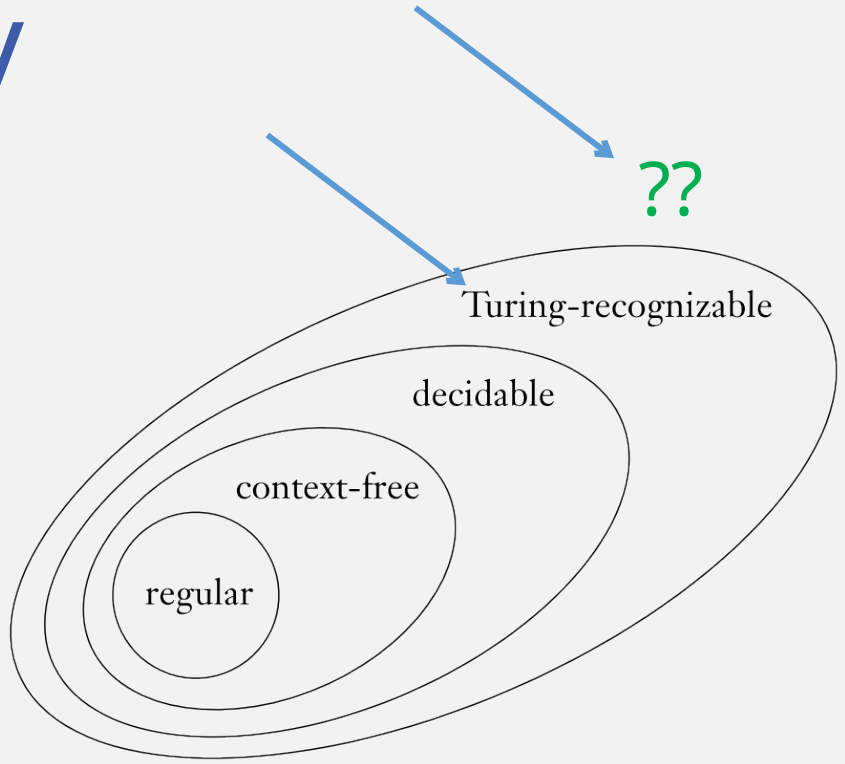
$$A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$



UMB CS 420

Undecidability

Monday, April 22, 2024



Announcements

- HW 9 out
  - due Wednesday 4/24 12pm noon
  - Problems 3 and 4 moved to HW 10

Lecture Participation Question

- Is the Universal Turing Machine (A_{TM}) a decider?

Language: of DFA description + string pairs, i.e., compute whether a DFA accepts a string

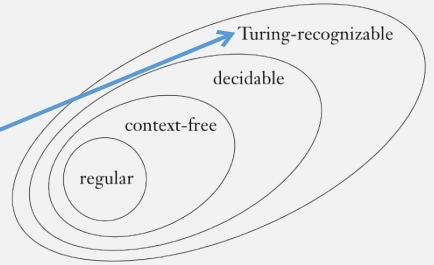
Recap: Decidability of Regular and CFLs

- $A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$ — Decidable
- $A_{\text{NFA}} = \{\langle B, w\rangle \mid B \text{ is an NFA that accepts input string } w\}$ — Decidable
- $A_{\text{REX}} = \{\langle R, w\rangle \mid R \text{ is a regular expression that generates string } w\}$ — Decidable
- $E_{\text{DFA}} = \{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$ — Decidable
- $EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$ — Decidable
- $A_{\text{CFG}} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates string } w\}$ — Decidable
- $E_{\text{CFG}} = \{\langle G\rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}$ — Decidable
- $EQ_{\text{CFG}} = \{\langle G, H\rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$ — *Undecidable?*
- $A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$ — *Undecidable?*

Compute something about DFA language, from its description

compute whether a TM accepts a string

Thm: A_{TM} is Turing-recognizable



$$A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

U = "On input $\langle M, w\rangle$, where M is a TM and w is a string:

1. Simulate M on input w.
2. If M ever enters its accept state, *accept*; if M ever enters its reject state, *reject*."

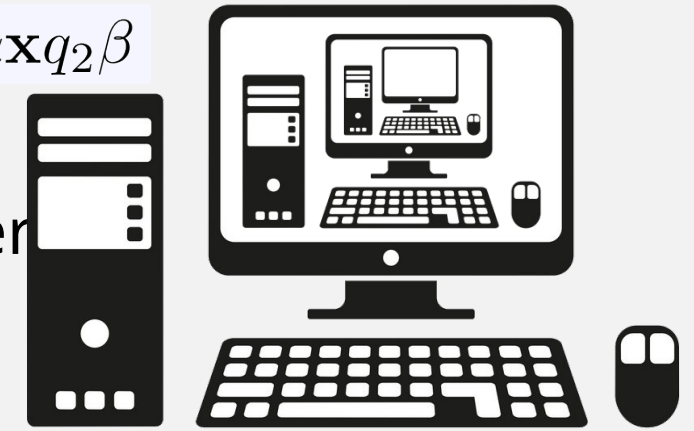
Can go into infinite loop, causing U to loop

U = Implements TM computation step $\alpha q_1 \mathbf{a}\beta \vdash \alpha \mathbf{x} q_2 \beta$

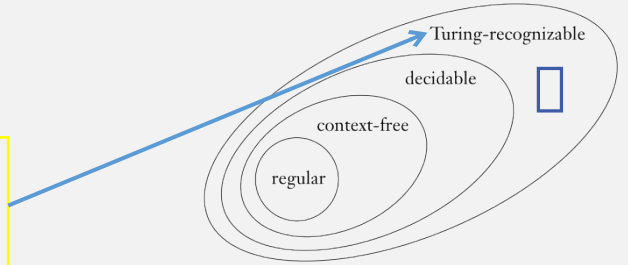
- i.e., "The Universal Turing Machine"
- "Program" simulating other programs (interpreter
- Problem (Step 1): U loops when M loops

Termination argument?

So it's not a decider. Is it recognizer?



Thm: A_{TM} is Turing-recognizable



$$A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

U = "On input $\langle M, w\rangle$, where M is a TM and w is a string:

1. Simulate M on input w.
2. If M ever enters its accept state, *accept*; if M ever enters its reject state, *reject*."

"Actual" behavior

"Expected" behavior

Example Str	M on input w?	U?	In A_{TM} lang?
$\langle M_1, \texttt{01\#01}\rangle$	Accept	Accept	Yes
$\langle M_1, \texttt{00\#11}\rangle$	Reject	Reject	No
$\langle M_{\text{loop}}, *\rangle$	Loop!	Loop!	No

Let:
- M_1 = "w#w" lang decider
- M_{loop} = looping TM

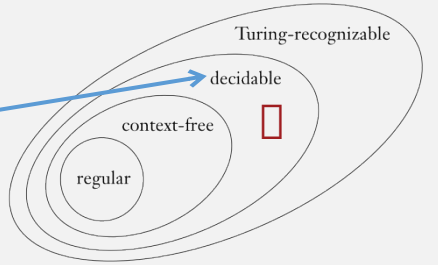
Columns must match!

Is this right?

Yes!

Thm: A_{TM} is undecidable

How to prove … not in here?



$$A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

- ???

lashback

Prove: Spider-Man does not exist???



In general, proving something not true is different (and harder) than proving it true

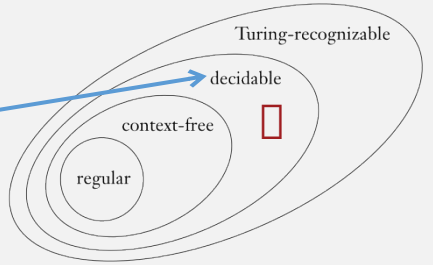
In some cases, it's possible, but typically requires new proof techniques!

Example (Regular Languages)
Prove a language is regular:
- Create a DFA

Prove a language is not regular:
- Proof by contradiction using Pumping Lemma

Thm: A_{TM} is undecidable

Not in here?



$$A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$



today

Example (decidable languages)

Prove a language is decidable:

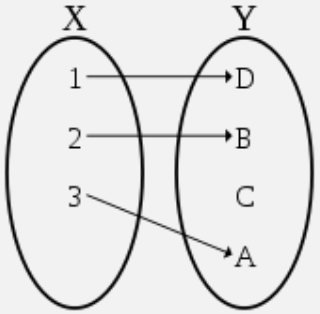
- Create a decider TM (with termination argument)

Prove a language is not decidable:

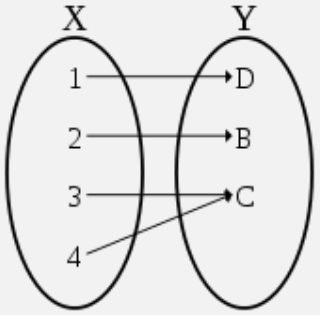
- ????

Kinds of Functions (a fn maps DOMAIN → RANGE)

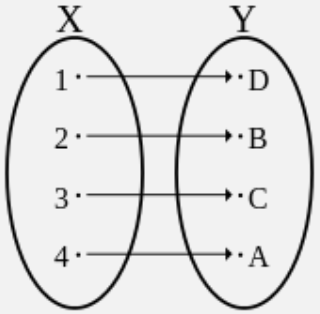
- Injective, a.k.a., “one-to-one”
  - Every element in DOMAIN has a unique mapping
  - How to remember:
    - Entire DOMAIN is mapped “in” to the RANGE



- Surjective, a.k.a., “onto”
  - Every element in RANGE is mapped to
  - How to remember:
    - “Sur” = “over” (eg, survey); DOMAIN is mapped “over” the RANGE



- Bijective, a.k.a., “correspondence” or “one-to-one correspondence”
  - Is both injective and surjective
  - Unique pairing of every element in DOMAIN and RANGE



Countability

- A set is “countable” if it is:
  - Finite
  - Or, there exists a bijection between the set and the natural numbers
    - In this case, the set has the same size as the set of natural numbers
    - This is called “countably infinite”

Exercise: Which set is larger?

- The set of:
  - Natural numbers, or
  - Even numbers?
- They are the same size! Both are countably infinite
  - Proof: Bijection:

n	$f(n) = 2n$
1	2
2	4
3	6
$\vdots$	$\vdots$

Natural numbers | Even numbers

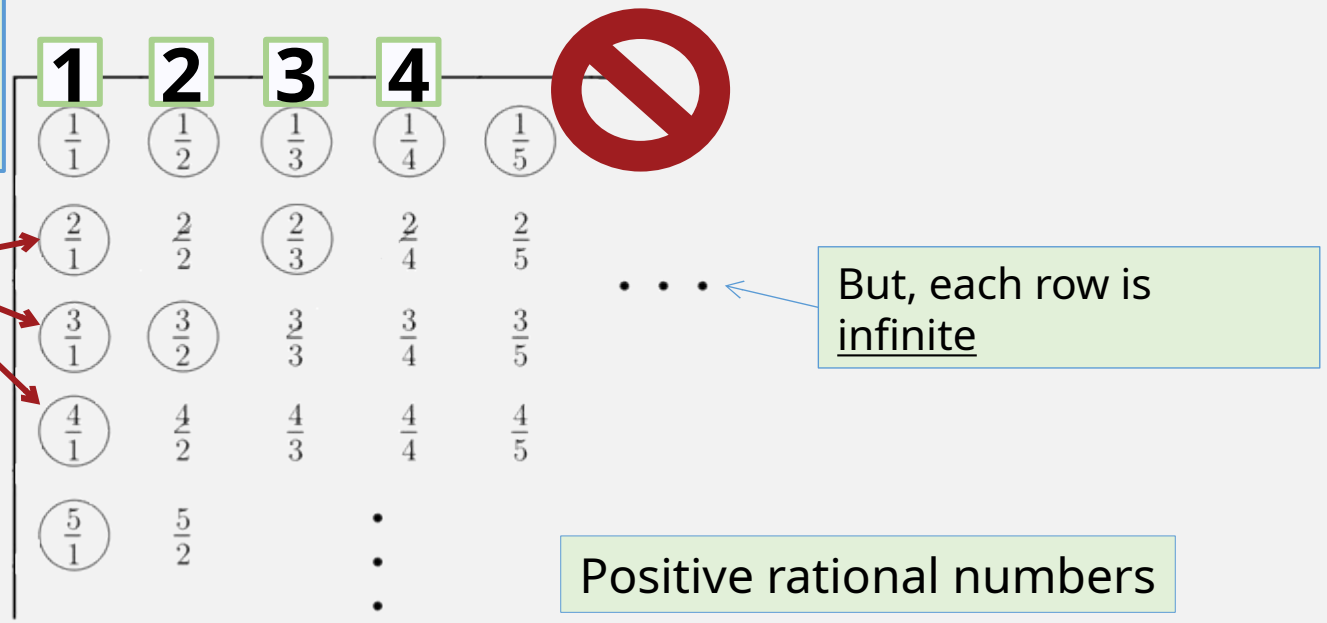
Every natural number maps to a unique even number, and vice versa

Exercise: Which set is larger?

- The set of:
  - Natural numbers $\mathcal{N}$, or
  - Positive rational numbers? $\mathcal{Q} = \{\frac{m}{n} | \, m, n \in \mathcal{N}\}$
- They are the same size! Both are countably infinite

A possible mapping of Natural numbers to Positive rationals?

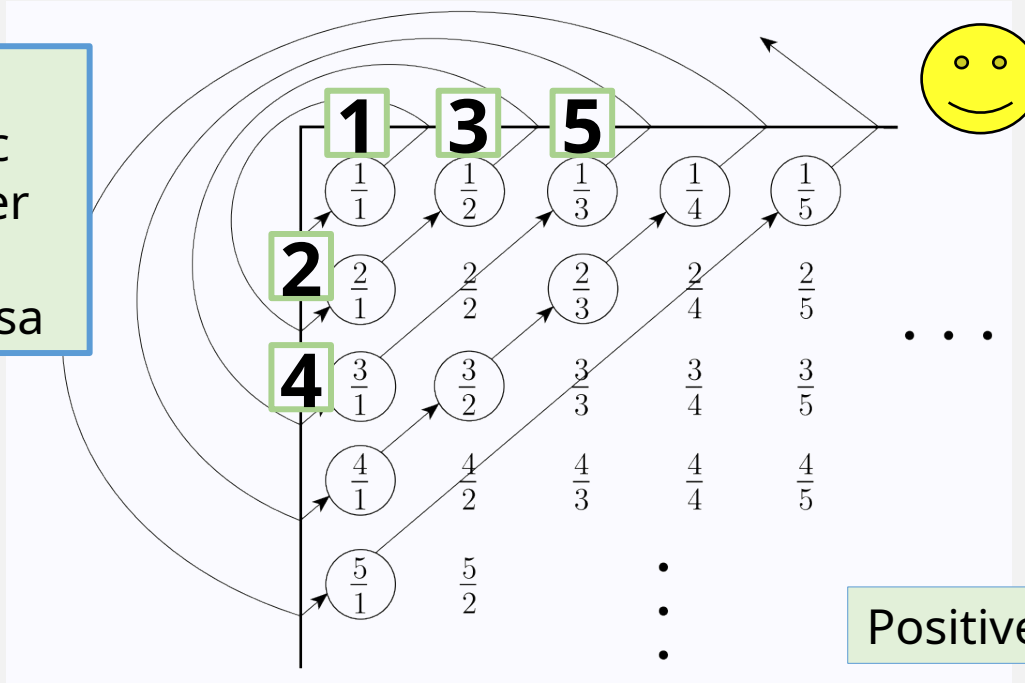
So these don't get mapped to: (not a bijection)



Exercise: Which set is larger?

- The set of:
  - Natural numbers $\mathcal{N}$, or
  - Positive rational numbers? $\mathcal{Q} = \{\frac{m}{n} \mid m, n \in \mathcal{N}\}$
- They are the same size! Both are countably infinite

Another mapping:
This is a bijection bc every natural number maps to a unique fraction, and vice versa



Exercise: Which set is larger?

- The set of:
  - Natural numbers $\mathcal{N}$, or
  - Real numbers? $\mathcal{R}$
- There are more real numbers. It is uncountably infinite.

Proof, by contradiction:

- Assume: a bijection between natural and real numbers exists.
  - So: every natural num maps to a unique real, and vice versa
- But we show that in any given mapping, … e.g.:
  - Some real number is not mapped to …
  - E.g., a number that has different digits at each position:

$$x = 0.4641\ldots$$

  - This number cannot be in the mapping …
  - … So we have a contradiction!

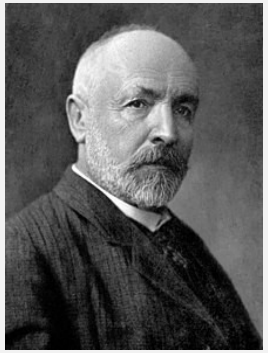
This proof technique is called diagonalization

different

n	$f(n)$
1	3.14159…
2	55.55555…
3	0.12345…
4	0.50000…
⋮	⋮

A hypothetical mapping

Georg Cantor





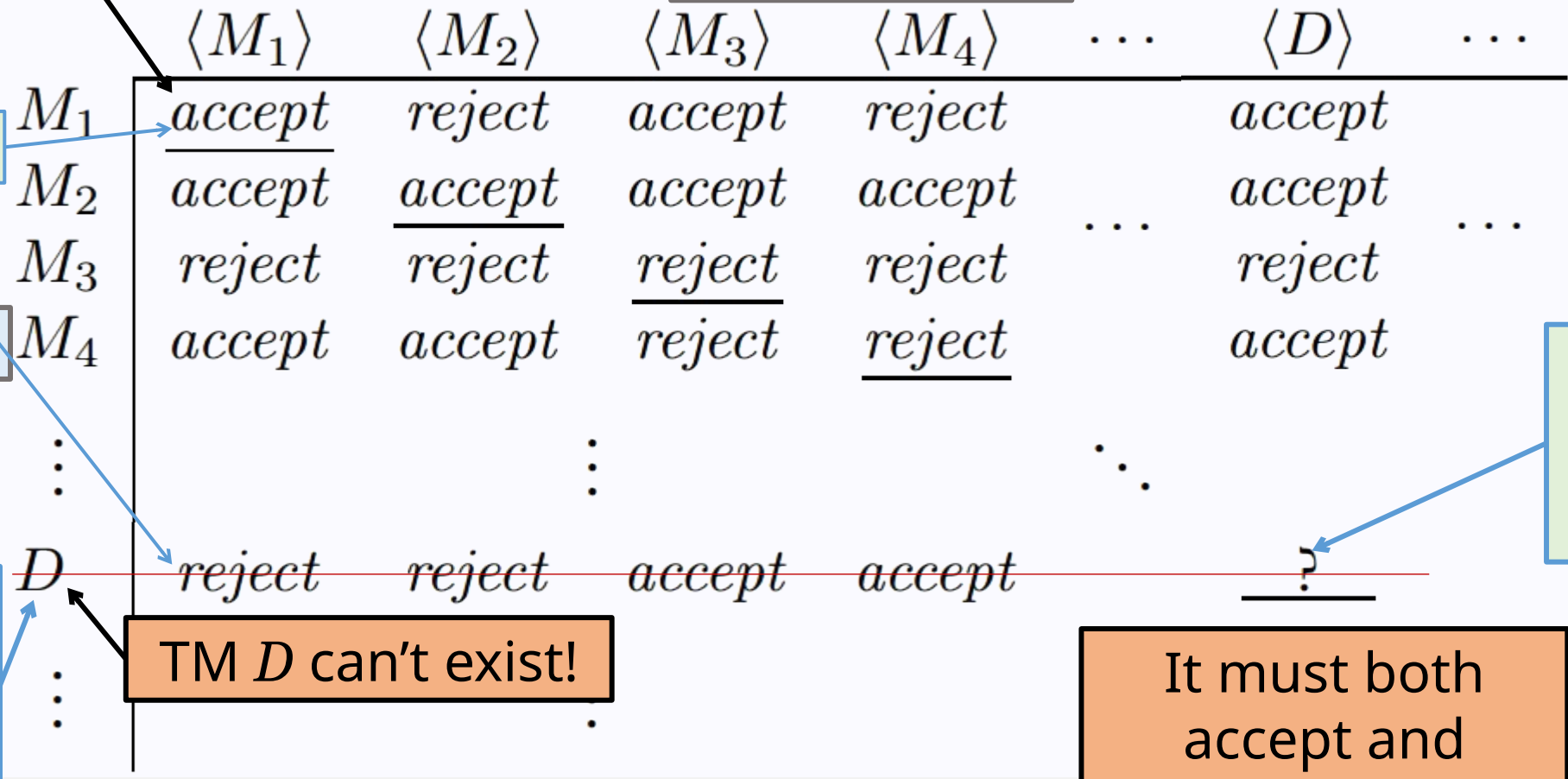
A formative day for Georg Cantor.

- Invented set theory
- Came up with countable infinity (1873)
- And uncountability:
  - Also: how to show uncountability with “diagonalization” technique

Diagonalization with Turing Machines

Diagonal: Result of Giving a TM its own Encoding as Input

All TM Encodings



opposites

All TMs

What should happen here?

Try to construct "opposite" TM D

TM D can't exist!

It must both accept and reject!

3 Easy Steps!

Thm: A_{TM} is undecidable

$$A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

Proof by contradiction:

1. Assume A_{TM} is decidable. So there exists a decider H for it:

$$H(\langle M, w\rangle) = \begin{cases} accept & \text{if } M \text{ accepts } w \\ reject & \text{if } M \text{ does not accept } w \end{cases}$$

2. Use H in another TM … the impossible "opposite" machine:

D = "On input $\langle M\rangle$, where M is a TM:

1. Run H on input $\langle M, \langle M\rangle\rangle$.
2. Output the opposite of what H outputs. That is, if H accepts, *reject*; and if H rejects, *accept*."

From previous slide (does opposite of what input TM would do if given itself)

H computes: M's result with itself as input

Do the opposite

3 Easy Steps!

Thm: A_{TM} is undecidable

$$A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

Proof by contradiction:

This cannot be true

1. Assume A_{TM} is decidable. So there exists a decider H for it:

$$H(\langle M, w\rangle) = \begin{cases} accept & \text{if } M \text{ accepts } w \\ reject & \text{if } M \text{ does not accept } w \end{cases}$$

2. Use H in another TM … the impossible "opposite" machine:

D = "On input $\langle M\rangle$, where M is a TM:

1. Run H on input $\langle M, \langle M\rangle\rangle$.
2. Output the opposite of what H outputs. That is, if H accepts, *reject*; and if H rejects, *accept*."

3. But D does not exist! Contradiction! So the assumption is false.

Easier Undecidability Proofs

- We proved $A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$ ecidable …
- … by contradiction:
  - Use hypothetical A_{TM} decider to create an impossible decider "D"!

reduce "D problem" to A_{TM}

- Step # 1: coming up with "D" --- hard!
  - Need to invent diagonalization

	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	$\cdots$	$\langle D\rangle$
M_1	*accept*	*reject*	*accept*	*reject*		*accept*
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$	*accept*
M_3	*reject*	*reject*	*reject*	*reject*		*reject*
M_4	*accept*	*accept*	*reject*	*reject*		*accept*
$\vdots$			$\vdots$		$\ddots$	
D	*reject*	*reject*	*accept*	*accept*		?

- Step # 2: reduce "D" problem to A_{TM} --- easier!

- From now on: undecidability proofs only need step # 2!
  - And we now have two "impossible" problems to choose from

The Halting Problem

$$HALT_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$$

Thm: $HALT_{\text{TM}}$ is undecidable

Proof, by contradiction:

- Assume: $HALT_{\text{TM}}$ has *decider R*: use it to create decider for A_{TM}.

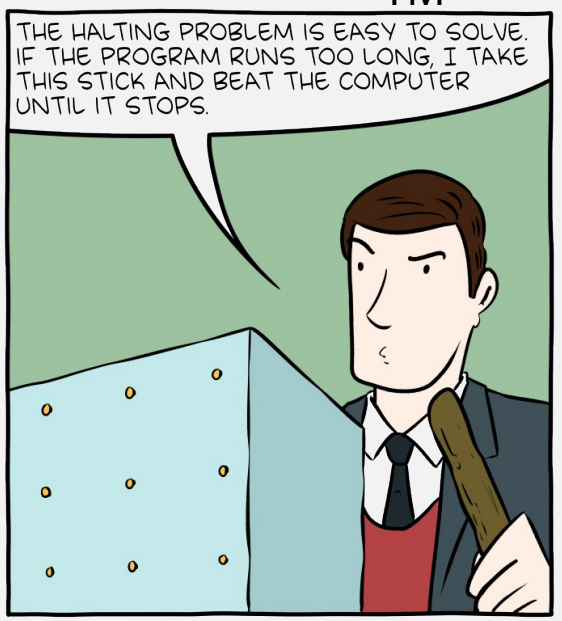
$$A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

reduce (from known undecidable) A_{TM} to $HALT_{\text{TM}}$

contradiction

- …

- But A_{TM} is undecidable and has no decider!



What if Alan Turing had been an engineer?

The Halting Problem

$$HALT_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$$

Thm: $HALT_{\text{TM}}$ is undecidable

Proof, by contradiction: Using our hypothetical $HALT_{\text{TM}}$ decider R

- Assume: $HALT_{\text{TM}}$ has *decider* R: use it to create decider for A_{TM}:

$$A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

$S =$ "On input $\langle M, w\rangle$, an encoding of a TM M and a string w:

1. Run TM R on input $\langle M, w\rangle$.
2. If R rejects, *reject*.
3. If R accepts, simulate
4. If M has accepted, *accept*; if M has rejected,

This means M loops on input w

This step always halts

Termination argument:
Step 1: R is a decider so always halts
Step 3: M always halts because R said

The Halting Problem

Undecidability Proof Technique #1:
Reduce (directly) from A_{TM}
(by creating A_{TM} decider)

$$HALT_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$$

Thm: $HALT_{\text{TM}}$ is undecidable

Proof, by contradiction:

- Assume: $HALT_{\text{TM}}$ has *decider* R: use it to create decider for A_{TM}:

$$A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

S = "On input $\langle M, w\rangle$, an encoding of a TM M and a string w:

1. Run TM R on input $\langle M, w\rangle$.
2. If R rejects, *reject*.
3. If R accepts, simulate M on w until it halts.
4. If M has accepted, *accept*; if M has rejected, *reject*."

Now we have three "impossible" deciders to choose from

- But A_{TM} is undecidable! I.e., this decider does not exist!
  - So $HALT_{\text{TM}}$ is also undecidable!

Interlude: Reducing from $HALT_{TM}$

A practical thought experiment …
… about compiler optimizations

Your compiler changes your program!

```
If TRUE then A else B  ⇒  A

1 + 2 + 3  ⇒  6
```

Compiler Optimizations

Optmization - docs

- -O0
  - No optmization, faster compilation time, better for debugging builds.
- -O2
- -O3
  - Higher level of optmization. Slower compile-time, better for production builds.
- -OFast
  - Enables higher level of optmization than (-O3). It enables lots of flags as can be seen src (-ffloat-store, -ffsast-math, -ffinite-math-only, -O3 …)
- -finline-functions
- -m64
- -funroll-loops
- -fvectorize
- -fprofile-generate

Types of optimization [edit]

Techniques used in optimization can be broken up among various *scopes* which can affect anything from a single statement to the entire program. Generally speaking, locally scoped techniques are easier to implement than global ones but result in smaller gains. Some examples of scopes include:

Peephole optimizations

These are usually performed late in the compilation process after machine code has been generated. This form of optimization examines a few adjacent instructions (like "looking through a peephole" at the code) to see whether they can be replaced by a single instruction or a shorter sequence of instructions.[2] For instance, a multiplication of a value by 2 might be more efficiently executed by left-shifting the value or by adding the value to itself (this example is also an instance of strength reduction).

Local optimizations

These only consider information local to a basic block.[3] Since basic blocks have no control flow, these optimizations need very little analysis, saving time and reducing storage requirements, but this also means that no information is preserved across jumps.

Global optimizations

These are also called "intraprocedural methods" and act on whole functions.[3] This gives them more information to work with, but often makes expensive computations necessary. Worst case assumptions have to be made when function calls occur or global variables are accessed because little information about them is available.

Loop optimizations

These act on the statements which make up a loop, such as a *for* loop, for example loop-invariant code motion. Loop optimizations can have a significant impact because many programs spend a large percentage of their time inside loops.[4]

Prescient store optimizations

These allow store operations to occur earlier than would otherwise be permitted in the context of threads and locks. The process needs some way of knowing ahead of time what value will be stored by the assignment that it should have followed. The purpose of this relaxation is to allow compiler optimization to perform certain kinds of code rearrangement that preserve the semantics of properly synchronized programs.[5]

Interprocedural, whole-program or link-time optimization

These analyze all of a program's source code. The greater quantity of information extracted means that optimizations can be more effective compared to when they only have access to local information, i.e. within a single function. This kind of optimization can also allow new techniques to be performed. For instance, function inlining, where a call to a function is replaced by a copy of the function body.

Machine code optimization and object code optimizer

These analyze the executable task image of the program after all of an executable machine code has been linked. Some of the techniques that can be applied in a more limited scope, such as macro compression which saves space by collapsing common sequences of instructions, are more effective when the entire executable task image is available for analysis.[6]

The Optimal Optimizing Compiler

"Full Employment" Theorem

Thm: The Optimal (C++) Optimizing Compiler does not exist

Proof, by contradiction:

Assume: *OPT* is the Perfect Optimizing Compiler

Use it to create $HALT_{TM}$ decider (accepts <*M*,*w*> if *M* halts with *w*, else rejects):

S = On input <*M*, *w*>, whe

- If *OPT*(*M*) == `for(;;)`
  a) Then Reject
  b) Else Accept

In computer science and mathematics, a **full employment theorem** is a term used, often humorously, to refer to a theorem which states that no algorithm can optimally perform a particular task done by some class of professionals. The name arises because such a theorem ensures that there is endless scope to keep discovering new techniques to improve the way at least some specific task is done.

For example, the *full employment theorem for compiler writers* states that there is no such thing as a provably perfect size-optimizing compiler, as such a proof for the compiler would have to detect non-terminating computations and reduce them to a one-instruction infinite loop. Thus, the existence of a provably perfect size-optimizing compiler would imply a solution to the halting problem, which cannot exist. This also implies that there may always be a better compiler since the proof that one has the best compiler cannot exist. Therefore, compiler writers will always be able to speculate that they have something to improve.

Summary: The Limits of Algorithms

- $A_{\text{DFA}} = \{\langle B, w\rangle |\ B \text{ is a DFA that accepts input string } w\}$ — Decidable
- $A_{\text{CFG}} = \{\langle G, w\rangle |\ G \text{ is a CFG that generates string } w\}$ — Decidable
- $A_{\text{TM}} = \{\langle M, w\rangle |\ M \text{ is a TM and } M \text{ accepts } w\}$ — Undecidable
- $HALT_{\text{TM}} = \{\langle M, w\rangle |\ M \text{ is a TM and } M \text{ halts on input } w\}$ — Undecidable

Similar languages

It's straightforward to use hypothetical $HALT_{\text{TM}}$ decider to create A_{TM} decider

Summary: The Limits of Algorithms

- $A_{\text{DFA}} = \{\langle B, w\rangle | \ B \text{ is a DFA that accepts input string } w\}$ — Decidable
- $A_{\text{CFG}} = \{\langle G, w\rangle | \ G \text{ is a CFG that generates string } w\}$ — Decidable
- $A_{\text{TM}} = \{\langle M, w\rangle | \ M \text{ is a TM and } M \text{ accepts } w\}$ — Undecidable
- $HALT_{\text{TM}} = \{\langle M, w\rangle | \ M \text{ is a TM and } M \text{ halts on input } w\}$ — Undecidable
- $E_{\text{DFA}} = \{\langle A\rangle | \ A \text{ is a DFA and } L(A) = \emptyset\}$ — Decidable
- $E_{\text{CFG}} = \{\langle G\rangle | \ G \text{ is a CFG and } L(G) = \emptyset\}$ — Decidable
- $E_{\text{TM}} = \{\langle M\rangle | \ M \text{ is a TM and } L(M) = \emptyset\}$ — Undecidable

Not as similar languages

next

How can we use a hypothetical E_{TM} decider to create A_{TM} or $HALT_{\text{TM}}$ decider?

Undecidability Proof Technique #2

Reducibility: Modifying the TM

$$E_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$$

Thm: E_{TM} is undecidable

Proof, by contradiction:

- Assume E_{TM} has *decider* R; use it to create *decider* for A_{TM}:

S = "On input $\langle M, w \rangle$, an encoding of a TM M and a string w:

- Run R on input $\langle M_1 \rangle$
- If R accepts, *reject* (because it means $\langle M \rangle$ doesn't accept anything)
- if R rejects, then ... ($\langle M \rangle$ accepts something, but is it w?...)

First, construct M_1

Note: M_1 is only used as arg to R; we never run it!

w

- Idea: Wrap $\langle M \rangle$ in a new TM that can *only* accept w:

M_1 = "On input x:

1. If $x \neq w$, *reject*.
2. If $x = w$, run M on input w and *accept* if M does."

Input not w, always reject

Input is w, maybe accept

M_1 accepts w if M does

Reducibility: Modifying the TM

$$E_{\mathsf{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$$

Thm: E_{TM} is undecidable

Proof, by contradiction:

This decider for A_{TM} cannot exist!

- Assume E_{TM} has *decider* R; use it to create *decider* for A_{TM}:

$S =$ "On input $\langle M, w \rangle$, an encoding of a TM M and a string w:

First, construct M_1

- Run R on input $\langle M_1 \rangle$
- If R accepts, *reject* (because it means $\langle M \rangle$ doesn't accept anything)
- if R rejects, then *accept* ($\langle M \rangle$ accepts something, but is it w???)

w

w

- Idea: Wrap $\langle M \rangle$ in a new TM that can *only* accept w:

$M_1 =$ "On input x:

1. If $x \neq w$, *reject*.
2. If $x = w$, run M on input w and *accept* if M does."

Summary: The Limits of Algorithms

- $A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$ — Decidable
- $A_{\text{CFG}} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates string } w\}$ — Decidable
- $A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$ — Undecidable
- $E_{\text{DFA}} = \{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$ — Decidable
- $E_{\text{CFG}} = \{\langle G\rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}$ — Decidable
- $E_{\text{TM}} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$ — Undecidable
- $EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$ — Decidable
- $EQ_{\text{CFG}} = \{\langle G, H\rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$ — Undecidable
- $EQ_{\text{TM}} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$ — Undecidable

needs

next

Undecidability Proof Technique #3

Reduce to something else: EQ_{TM} is undecidable

$$EQ_{\text{TM}} = \{\langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$$

Proof, by contradiction:

- Assume: EQ_{TM} has *decider* R; use it to create *decider* for A_{TM}:

$$E_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$$

S = "On input $\langle M \rangle$, where M is a TM:

1. Run R on input $\langle M, M_1 \rangle$, where M_1 is a TM that rejects all inputs.
2. If R accepts, *accept*; if R rejects, *reject*."

Reduce to something else: EQ_{TM} is undecidable

$$EQ_{\text{TM}} = \{\langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$$

Proof, by contradiction:

- Assume: EQ_{TM} has *decider* R; use it to create *decider* for E_{TM}:

$$= \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$$

S = "On input $\langle M \rangle$, where M is a TM:

1. Run R on input $\langle M, M_1 \rangle$, where M_1 is a TM that rejects all inputs.
2. If R accepts, *accept*; if R rejects, *reject*."

- But E_{TM} is undecidable!

Summary: Undecidability Proof Techniques

- Proof Technique #1:
  - Use hypothetical decider to implement impossible A_{TM} decider
  - Example Proof:

$$A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

Reduce

$$HALT_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$$

- Proof Technique #2:
  - Use hypothetical decider to implement impossible A_{TM} decider
  - But first modify the input M
  - Example Proof:

Can also combine these techniques

$$E_{\text{TM}} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$$

Reduce

- Proof Technique #3:
  - Use hypothetical decider to implement non-A_{TM} impossible decider
  - Example Proof:

$$EQ_{\text{TM}} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$$

Summary: Decidability and Undecidability

- $A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$ — Decidable
- $A_{\text{CFG}} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates string } w\}$ — Decidable
- $A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$ — Undecidable
- $E_{\text{DFA}} = \{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$ — Decidable
- $E_{\text{CFG}} = \{\langle G\rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}$ — Decidable
- $E_{\text{TM}} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$ — Undecidable
- $EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$ — Decidable
- $EQ_{\text{CFG}} = \{\langle G, H\rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$ — Undecidable
- $EQ_{\text{TM}} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$ — Undecidable

Also Undecidable …

next

- $REGULAR_{TM}$ = {<M> | M is a TM and $L(M)$ is a regular language}

Undecidability Proof Technique #2:
Modify input TM M

Thm: $REGULAR_{\text{TM}}$ is undecidable

$$REGULAR_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is a regular language}\}$$

Proof, by contradiction:

- Assume: $REGULAR_{\text{TM}}$ has *decider* R; use it to create *decider* for

S = "On input $\langle M, w \rangle$, an encoding of a TM M and a string w:

First, construct M_2 (??)

- Run R on input $\langle M_2 \rangle$
- If R accepts, *accept*; if R rejects, *reject*
-

Want: $L(M_2)$ =

- regular, if M accepts w
- nonregular, if M does not accept w

Thm: $REGULAR_{\mathsf{TM}}$ is undecidable (continued)

$$REGULAR_{\mathsf{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is a regular language}\}$$

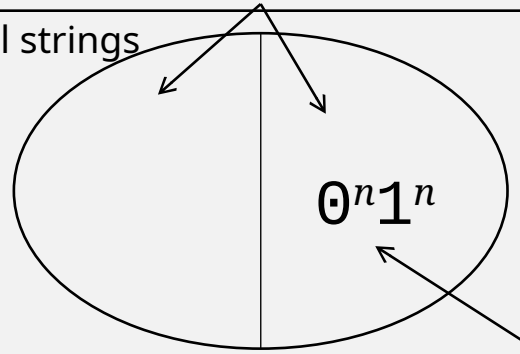
M_2 = "On input x:

1. If x has the form 0^n1^n, *accept*.
2. If x does not have this form, run M on input w and *accept* if M accepts w."

Always accept strings 0^n1^n
$L(M_2)$ = nonregular, so far

If M accepts w, accept everything else, so $L(M_2) = \Sigma^* =$ regular

if M does not accept w, M_2 accepts all strings (regular lang)



Want: $L(M_2)$ =

- regular, if M accepts w
- nonregular, if M does not accept w

if M accepts w, M_2 accepts this nonregular lang

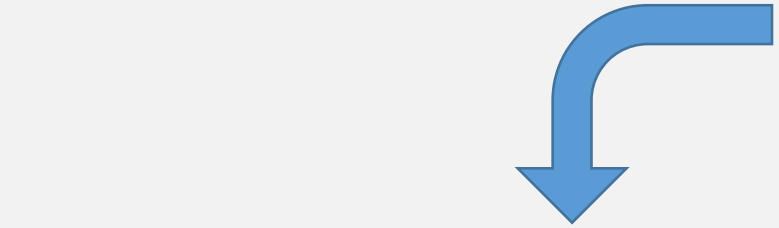
Also Undecidable …

Seems like no algorithm can compute anything about
the language of a Turing Machine,
i.e., about the runtime behavior of programs
…

- $REGULAR_{TM}$ = {<M> | M is a TM and $L(M)$ is a regular language}
- $CONTEXTFREE_{TM}$ = {<M> | M is a TM and $L(M)$ is a CFL}
- $DECIDABLE_{TM}$ = {<M> | M is a TM and $L(M)$ is a decidable language}
- $FINITE_{TM}$ = {<M> | M is a TM and $L(M)$ is a finite language}

An Algorithm About Program Behavior?

```
main()
{
    printf("hello, world\n");
}
```



Write a program that,
given another program as its argument,
returns TRUE if that argument prints
“Hello, World!”



TRUE

Fermat's Last Theorem
(unknown for ~350 years,
solved in 1990s)

```
main()
{
   If x^n + y^n = z^n, for any integer n > 2
      printf("hello, world\n");
}
```

Write a program that,
given another program as its argument,
returns TRUE if that argument prints
"Hello, World!"

?????

Also Undecidable …

Seems like no algorithm can compute anything about the language of a Turing Machine, i.e., about the runtime behavior of programs …

- $REGULAR_{TM}$ = {<M> | M is a TM and $L(M)$ is a regular language}
- $CONTEXTFREE_{TM}$ = {<M> | M is a TM and $L(M)$ is a CFL}
- $DECIDABLE_{TM}$ = {<M> | M is a TM and $L(M)$ is a decidable language}
- $FINITE_{TM}$ = {<M> | M is a TM and $L(M)$ is a finite language}
- …
- $ANYTHING_{TM}$ = {<M> | M is a TM and "… **anything** …" about $L(M)$}

Rice's Theorem

Rice's Theorem: $ANYTHING_{TM}$ is Undecidable

$ANYTHING_{TM}$ = {<M> | M is a TM and … **anything** … about $L(M)$}

- "… Anything …", more precisely:

  For any M_1, M_2,

  - if $L(M_1) = L(M_2)$
  - then $M_1 \in ANYTHING_{TM} \Leftrightarrow M_2 \in ANYTHING_{TM}$

- Also, "… Anything …"must be "non-trivial":
  - $ANYTHING_{TM}$!= {}
  - $ANYTHING_{TM}$!= set of all TMs

Rice's Theorem: $ANYTHING_{TM}$ is Undecidable

$ANYTHING_{TM}$ = {<M> | M is a TM and … **anything** … about $L(M)$}

Proof by contradiction

- Assume some language satisfying $ANYTHING_{TM}$ has a decider R.
  - Since $ANYTHING_{TM}$ is non-trivial, then there exists $M_{ANY} \in ANYTHING_{TM}$
  - Where R accepts M_{ANY}
- Use R to create decider for A_{TM}:

On input <M, w>:

- Create M_w:

  M_w = on input x:
  - Run M on w
    - If M rejects w: reject x
    - If M accepts w:
      Run M_{ANY} on x and accept if it accepts, else reject

  If M accepts w: $M_w = M_{ANY}$
  If M doesn't accept w: M_w accepts nothing

- Run R on M_w
  - If it accepts, then $M_w = M_{ANY}$, so M accepts w, so accept
  - Else reject

These two cases must be different, (so R can distinguish when M accepts w)

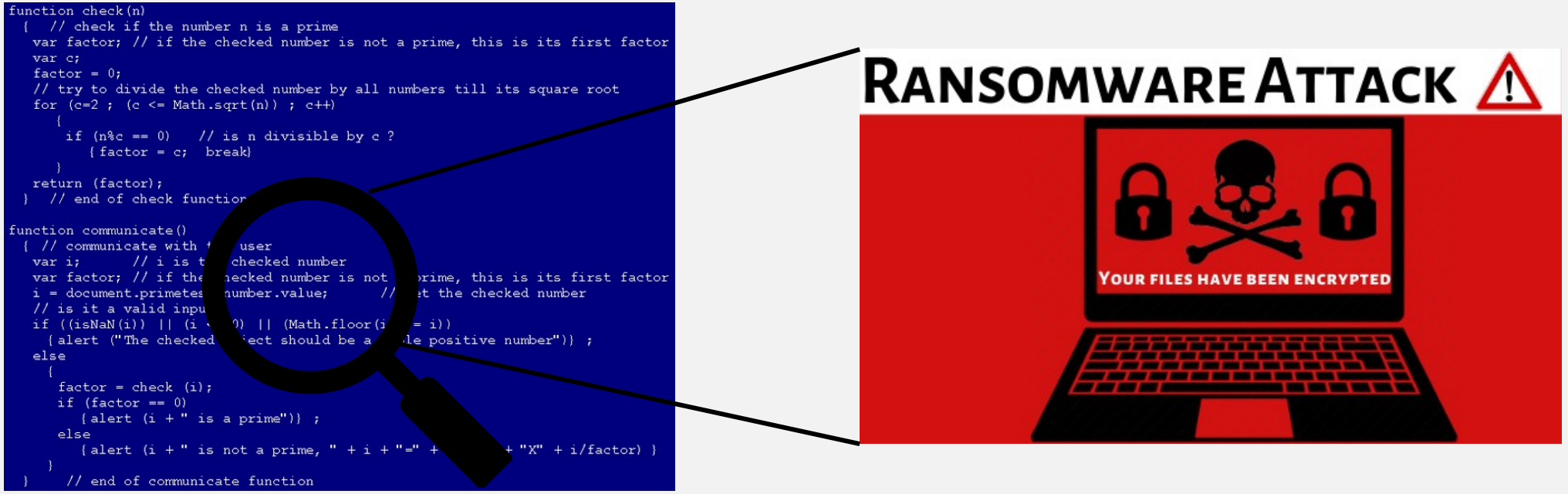
Wait! What if the TM that accepts nothing is in $ANYTHING_{TM}$!

Proof still works! Just use the complement of $ANYTHING_{TM}$ instead!

Rice's Theorem Implication

$\{\langle M \rangle \mid M \text{ is a TM that installs malware}\}$

Undecidable!
(by Rice's Theorem)



Submit in-class participation question

On gradescope

Easier Undecidability Proofs

- We proved $A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$ undecidable …

  … by contradiction:

- By showing its decider can help create <u>impossible</u> decider "D"!

- <u>Hard</u>: Coming up with "D" (needed to invent diagonalization)

- But then we <u>more easily</u> reduced A_{TM} to "D"

- <u>Easier</u>: reduce problems to A_{TM} !

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$	$\langle D \rangle$
M_1	*accept*	*reject*	*accept*	*reject*		*accept*
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$	*accept*
M_3	*reject*	*reject*	*reject*	*reject*		*reject*
M_4	*accept*	*accept*	*reject*	*reject*		*accept*
$\vdots$			$\vdots$		$\ddots$	
D	*reject*	*reject*	*accept*	*accept*		?

I.e., “Algorithm to determine if a TM is an decider”?

The Halting Problem

$$HALT_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$$

Thm: $HALT_{\text{TM}}$ is undecidable

Proof, by contradiction:

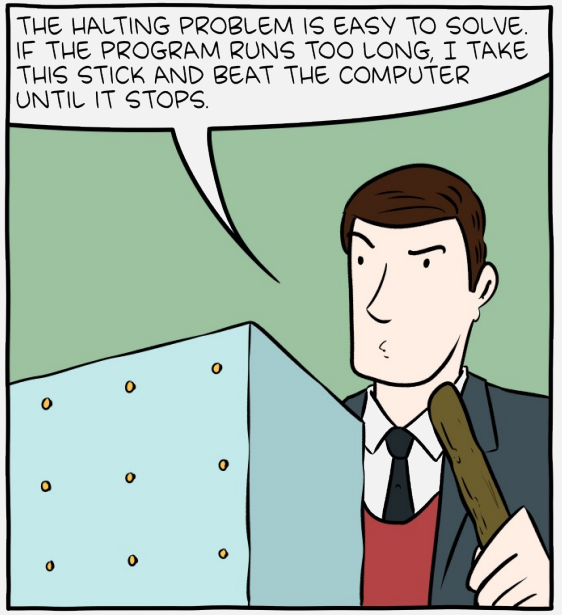
- Assume $HALT_{\text{TM}}$ has decider R; use it to create decider for A_{TM}

  ⋮

- …

contradiction

- But A_{TM} is undecidable and has no decider!



What if Alan Turing had been an engineer?

The Halting Problem

$$HALT_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$$

Thm: $HALT_{\text{TM}}$ is undecidable

Proof, by contradiction: Using our hypothetical decider R

- Assume $HALT_{\text{TM}}$ has decider R; use it to create decider for A_{TM}

:

$S =$ "On input $\langle M, w\rangle$, an encoding of a TM M and a string w:

1. Run TM R on input $\langle M, w\rangle$.
2. If R rejects, *reject*.
3. If R accepts, simulate
4. If M has accepted, *accept*; if M has rejected,

This means M loops on input w

This step always halts

Termination argument:
Step 1: R is a decider so always halts
Step 3: M always halts bc R said so

The Halting Problem

$$HALT_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$$

Thm: $HALT_{\text{TM}}$ is undecidable

Proof, by contradiction:

- Assume $HALT_{\text{TM}}$ has decider R; use it to create decider for A_{TM}

S = "On input $\langle M, w\rangle$, an encoding of a TM M and a string w:

1. Run TM R on input $\langle M, w\rangle$.
2. If R rejects, *reject*.
3. If R accepts, simulate M on w until it halts.
4. If M has accepted, *accept*; if M has rejected, *reject*."

- But A_{TM} is undecidable!
  - I.e., the decider we just created does not exist! So $HALT_{\text{TM}}$ is undecidable

Easier Undecidability Proofs

In general, to prove the undecidability of a language,
use proof by contradiction:

1. Assume the language is decidable (and thus has a decider)

2. Show that its decider can be used to create another decider …

   … for a known undecidable language …

3. … which cannot have a decider! That's a Contradiction!

Summary: The Limits of Algorithms

- $A_{\text{DFA}} = \{\langle B, w\rangle | \ B \text{ is a DFA that accepts input string } w\}$ — Decidable
- $A_{\text{CFG}} = \{\langle G, w\rangle | \ G \text{ is a CFG that generates string } w\}$ — Decidable
- $A_{\text{TM}} = \{\langle M, w\rangle | \ M \text{ is a TM and } M \text{ accepts } w\}$ — Undecidable
- $E_{\text{DFA}} = \{\langle A\rangle | \ A \text{ is a DFA and } L(A) = \emptyset\}$ — Decidable
- $E_{\text{CFG}} = \{\langle G\rangle | \ G \text{ is a CFG and } L(G) = \emptyset\}$ — Decidable
- $E_{\text{TM}} = \{\langle M\rangle | \ M \text{ is a TM and } L(M) = \emptyset\}$ — Undecidable

next

Undecidability Proofs

- We proved $A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$ decidable …
- … by contradiction:
  - Use hypothetical A_{TM} decider to create an impossible decider "D"!

	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	$\cdots$	$\langle D\rangle$
M_1	*accept*	*reject*	*accept*	*reject*		*accept*
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$	*accept*
M_3	*reject*	*reject*	*reject*	*reject*		*reject*
M_4	*accept*	*accept*	*reject*	*reject*		*accept*
$\vdots$			$\vdots$		$\ddots$	
D	*reject*	*reject*	*accept*	*accept*		?

- Step # 1: coming up with "D" --- hard!
  - Need to invent diagonalization
- Step # 2: "reduce" A_{TM} to the "D" problem --- easier!
- From now on: undecidability proofs only need to do step # 2!
  - And we now have two "impossible" problems to choose from

The Halting Problem

$$HALT_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$$

Thm: $HALT_{\text{TM}}$ is undecidable

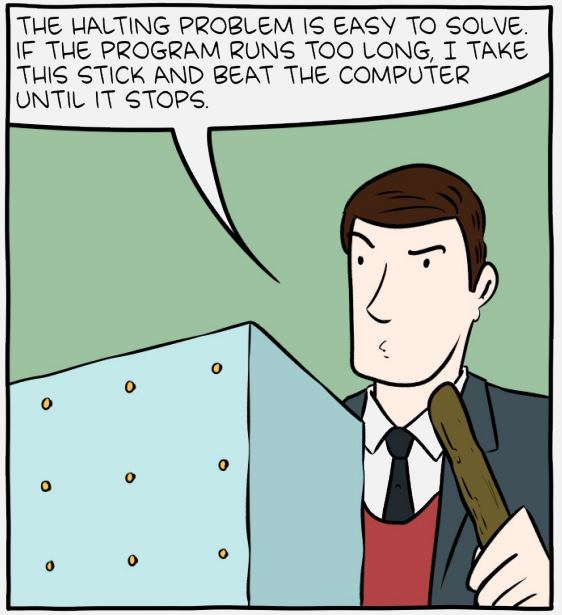
Proof, by contradiction:

- Assume: $HALT_{\text{TM}}$ has *decider* R: use it to create decider for A_{TM}:

$$A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

- …

contradiction



What if Alan Turing had been an engineer?

- But A_{TM} is undecidable and has no decider!

Last Time: The Halting Problem

$$HALT_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$$

Thm: $HALT_{\text{TM}}$ is undecidable

Proof, by contradiction: Using our hypothetical $HALT_{\text{TM}}$ decider R

- Assume: $HALT_{\text{TM}}$ has *decider* R: use it to create decider for A_{TM}:

$$A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

$S =$ "On input $\langle M, w\rangle$, an encoding of a TM M and a string w:

1. Run TM R on input $\langle M, w\rangle$.
2. If R rejects, *reject*.
3. If R accepts, simulate
4. If M has accepted, *accept*; if M has rejected,

This means M loops on input w

This step always halts

Termination argument:
Step 1: R is a decider so always halts
Step 3: M always halts because R said

Last Time: The Halting Problem

Undecidability Proof Technique #1:
Reduce from A_{TM}

$$HALT_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$$

Thm: $HALT_{\text{TM}}$ is undecidable

Proof, by contradiction:

- Assume: $HALT_{\text{TM}}$ has *decider* R: use it to create decider for A_{TM}:

$$A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

$S =$ "On input $\langle M, w\rangle$, an encoding of a TM M and a string w:

1. Run TM R on input $\langle M, w\rangle$.
2. If R rejects, *reject*.
3. If R accepts, simulate M on w until it halts.
4. If M has accepted, *accept*; if M has rejected, *reject*."

- But A_{TM} is undecidable! I.e., this decider does not exist!
  - So $HALT_{\text{TM}}$ is also undecidable!

Reducibility: Modifying the TM

$$E_{\mathsf{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$$

Thm: E_{TM} is undecidable

Proof, by contradiction:

- Assume E_{TM} has *decider* R; use to create A_{TM} *decider*:

$S =$ "On input $\langle M, w \rangle$, an encoding of a TM M and a string w:

- Run R on input $\langle M_1 \rangle$
- If R accepts, *reject* (because it means $\langle M \rangle$ doesn't accept w)
- if R rejects, then *accept* ($\langle M \rangle$ accepts w)

First, construct M_1

Note: M_1 is only used as arg to R; we never run it!

- Idea: Wrap $\langle M \rangle$ in a new TM that can *only* accept w:

$M_1 =$ "On input x:

1. If $x \neq w$, *reject*.
2. If $x = w$, run M on input w and *accept* if M does."

Reducibility: Modifying the TM

$$E_{\mathsf{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$$

Thm: E_{TM} is undecidable

Proof, by contradiction:

This decider R cannot exist!

- Assume E_{TM} has *decider* R; use to create A_{TM} *decider*:

$S =$ "On input $\langle M, w \rangle$, an encoding of a TM M and a string w:

- Run R on input $\langle M_1 \rangle$
  - If R accepts, *reject* (because it means $\langle M \rangle$ doesn't accept w)
  - if R rejects, then *accept* ($\langle M \rangle$ accepts w)

First, construct M_1

- Idea: Wrap $\langle M \rangle$ in a new TM that can *only* accept w:

$M_1 =$ "On input x:

1. If $x \neq w$, *reject*.
2. If $x = w$, run M on input w and *accept* if M does."

One more, modify M: $REGULAR_{\mathsf{TM}}$ is undecidable

$$REGULAR_{\mathsf{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is a regular language}\}$$

Proof, by contradiction:

- Assume $REGULAR_{\mathsf{TM}}$ has *decider* R; use to create A_{TM} *decider*:

$S =$ "On input $\langle M, w \rangle$, an encoding of a TM M and a string w:

- First, construct M_2 (??)
- Run R on input $\langle M_2 \rangle$
- If R accepts, *accept*; if R rejects, *reject*

Want: $L(M_2)$ =

- regular, if M accepts w
- nonregular, if M does not accept w

Thm: $REGULAR_{\text{TM}}$ is undecidable (continued)

$$REGULAR_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is a regular language}\}$$

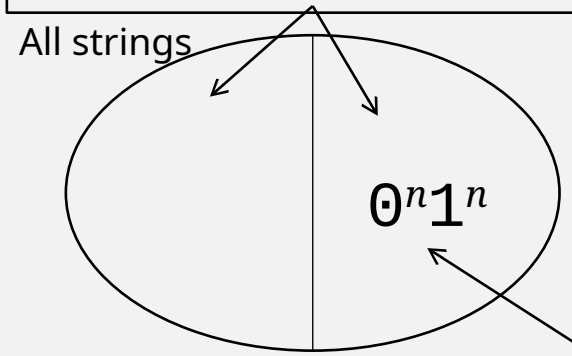
M_2 = "On input x:

1. If x has the form 0^n1^n, *accept*.
2. If x does not have this form, run M on input w and *accept* if M accepts w."

Always accept strings 0^n1^n
$L(M_2)$ = nonregular, so far

If M accepts w, accept everything else,
so $L(M_2) = \Sigma^* =$ regular

if M does not accept w, M_2 accepts all strings (regular lang)



Want: $L(M_2)$ =

- regular, if M accepts w
- nonregular, if M does not accept w

if M accepts w, M_2 accepts this non-regular lang

Summary: The Limits of Algorithms

- $A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$ — Decidable
- $A_{\text{CFG}} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates string } w\}$ — Decidable
- $A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$ — Undecidable
- $E_{\text{DFA}} = \{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$ — Decidable
- $E_{\text{CFG}} = \{\langle G\rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}$ — Decidable
- $E_{\text{TM}} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$ — Undecidable
- $EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$ — Decidable
- $EQ_{\text{CFG}} = \{\langle G, H\rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$ — Undecidable
- $EQ_{\text{TM}} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$ — Undecidable

needs

next

Reduce to something else: EQ_{TM} is undecidable

$$EQ_{\mathsf{TM}} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$$

Proof, by contradiction:

$$E_{\mathsf{TM}} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$$

- Assume EQ_{TM} has *decider* R; use to create ~~A_{TM}~~ *decider*:

$S =$ "On input $\langle M\rangle$, where M is a TM:

1. Run R on input $\langle M, M_1\rangle$, where M_1 is a TM that rejects all inputs.
2. If R accepts, *accept*; if R rejects, *reject*."

Reduce to something else: EQ_{TM} is undecidable

$$EQ_{\mathsf{TM}} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$$

Proof, by contradiction:

$$E_{\mathsf{TM}} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$$

- Assume EQ_{TM} has *decider* R; use to create ~~A_{TM}~~ *decider*:

S = "~~On input $\langle M\rangle$, where M is a TM:~~

1. Run R on input $\langle M, M_1\rangle$, where M_1 is a TM that rejects all inputs.
2. If R accepts, *accept*; if R rejects, *reject*."

- But E_{TM} is undecidable!

Summary

- $A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$ — Decidable
- $A_{\text{CFG}} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates string } w\}$ — Decidable
- $A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$ — Undecidable
- $E_{\text{DFA}} = \{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$ — Decidable
- $E_{\text{CFG}} = \{\langle G\rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}$ — Decidable
- $E_{\text{TM}} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$ — Undecidable
- $EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A)$ — Decidable
- $EQ_{\text{CFG}} = \{\langle G, H\rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$ — Undecidable
- $EQ_{\text{TM}} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$ — Undecidable

There's no algorithm to compute anything about Turing Machines, i.e., about general programs!

Also Undecidable …

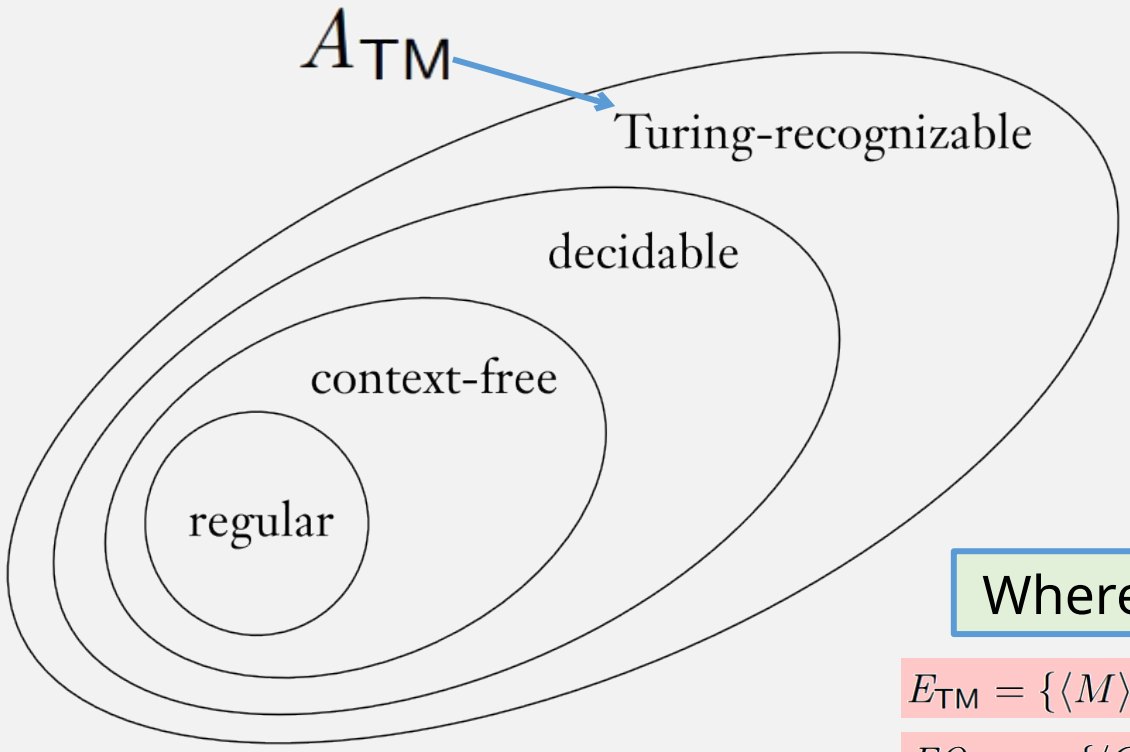
today

- $REGULAR_{TM}$ = {<M> | M is a TM and $L(M)$ is a regular language}
- $CONTEXTFREE_{TM}$ = {<M> | M is a TM and $L(M)$ is a CFL}
- $DECIDABLE_{TM}$ = {<M> | M is a TM and $L(M)$ is a decidable language}
- $FINITE_{TM}$ = {<M> | M is a TM and $L(M)$ is a finite language}
- …
- $ANYTHING_{TM}$ = {<M> | M is a TM and "**anything**" about $L(M)$}

Rice's Theorem

Turing Unrecognizable?

Is there anything out here?



Where do these go?

$E_{\text{TM}} = \{\langle M \rangle | \ M \text{ is a TM and } L(M) = \emptyset\}$

$EQ_{\text{CFG}} = \{\langle G, H \rangle | \ G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$

$EQ_{\text{TM}} = \{\langle M_1, M_2 \rangle | \ M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$

Thm: Some langs are not Turing-recognizable

Proof: requires 2 lemmas

- Lemma 1: The **set of all languages** is *uncountable*
  - Proof: Show there is a bijection with another uncountable set …
    - … The set of all infinite binary sequences
- Lemma 2: The **set of all TMs** is *countable*
- Therefore, some language is not recognized by a TM

Mapping a Language to a Binary Sequence

All Possible Strings

Some Language (subset of above)

Its (unique) Binary Sequence

$$\Sigma^* = \{ \varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \cdots \}$$

$$A = \{ 0, 00, 01, 000, 001, \cdots \}$$

$$\chi_A = 0\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ \cdots$$

Each digit represents one possible string:
- 1 if lang has that string,
- 0 otherwise

Thm: Some langs are not Turing-recognizable

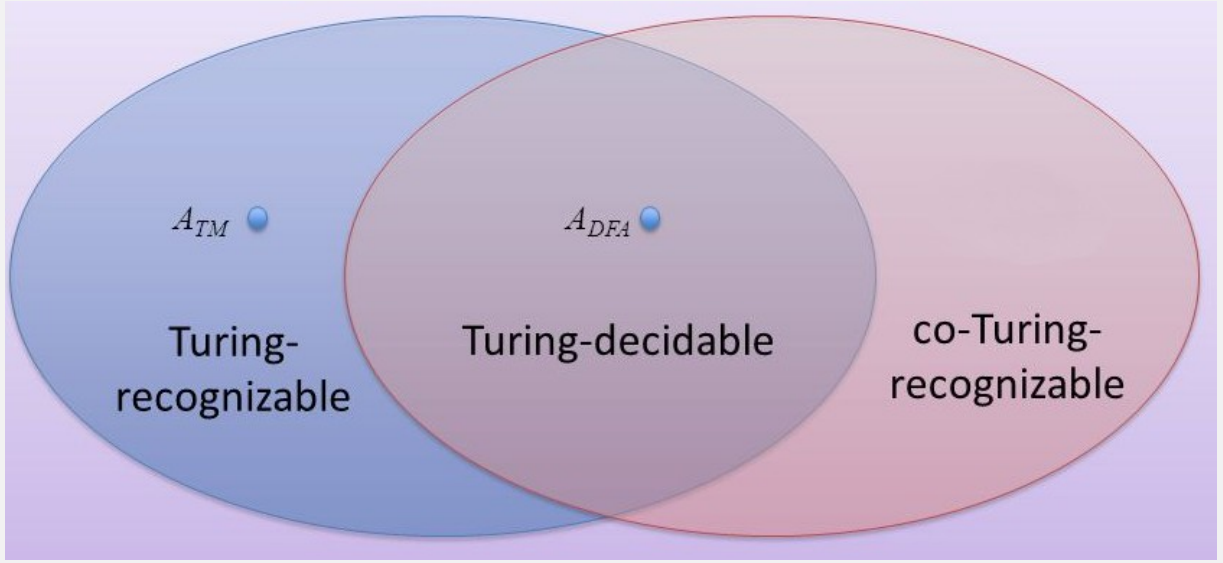
Proof: requires 2 lemmas

- Lemma 1: The **set of all languages** is *uncountable*
  - Proof: Show there is a bijection with another uncountable set …
    - … The set of all infinite binary sequences
    - ➢ Now just prove set of infinite binary sequences is uncountable (diagonalization)
- Lemma 2: The **set of all TMs** is *countable*
  - Because every TM M can be encoded as a string $\langle M \rangle$
  - And set of all strings is countable
- Therefore, some language is not recognized by a TM ■

Co-Turing-Recognizability

- A language is **co-Turing-recognizable** if …
- … it is the complement of a Turing-recognizable language.

Thm: Decidable ⇔ Recognizable & co-Recognizable



Thm: Decidable ⇔ Recognizable & co-Recognizable

⇒ If a language is decidable, then it is recognizable and co-recognizable

- Decidable => Recognizable:
  - A decider is a recognizer, bc decidable langs are a subset of recognizable langs
- Decidable => Co-Recognizable:
  - To create co-decider from a decider … switch reject/accept of all inputs
  - A co-decider is a co-recognizer, for same reason as above

⇐ If a language is recognizable and co-recognizable, then it is decidable

Thm: Decidable ⇔ Recognizable & co-Recognizable

⇒ If a language is decidable, then it is recognizable and co-recognizable

- Decidable => Recognizable:
  - A decider is a recognizer, bc decidable langs are a subset of recognizable langs
- Decidable => Co-Recognizable:
  - To create co-decider from a decider … switch reject/accept of all inputs
  - A co-decider is a co-recognizer, for same reason as above

⇐ If a language is recognizable and co-recognizable, then it is decidable

- Let M_1 = recognizer for the language,
- and M_2 = recognizer for its complement
- Decider M:
  - Run 1 step on M_1,
  - Run 1 step on M_2,
  - Repeat, until one machine accepts. If it's M_1, accept. If it's M_2, reject

Termination Arg: Either M_1 or M_2 must accept and halt, so M halts and is a

A Turing-unrecognizable language

- We've proved:

A_{TM} is Turing-recognizable

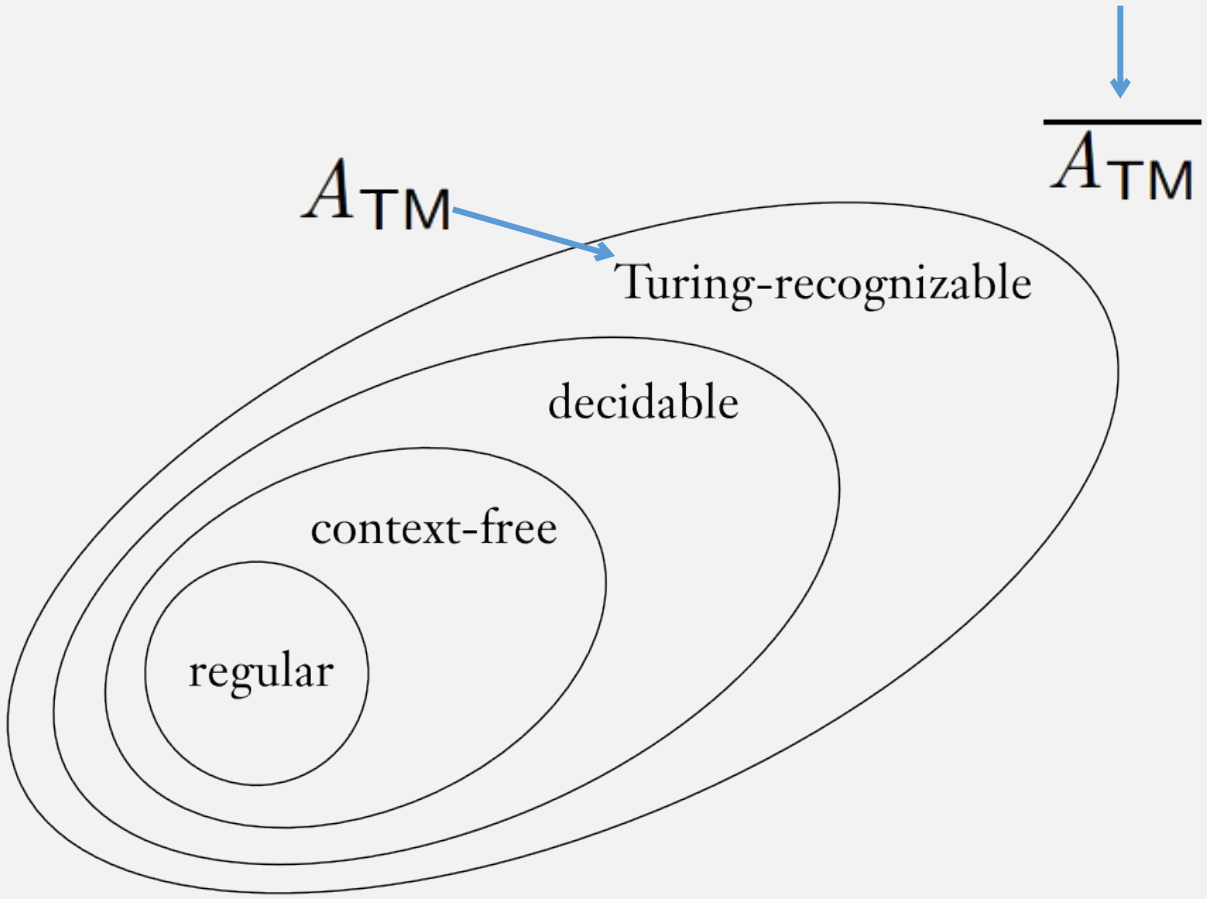
A_{TM} is undecidable

- So:

$\overline{A_{\mathsf{TM}}}$ is not Turing-recognizable

- Because: recognizable & co-recognizable implies decidable

Is there anything out here?



Check-in Quiz 4/12

On gradescope