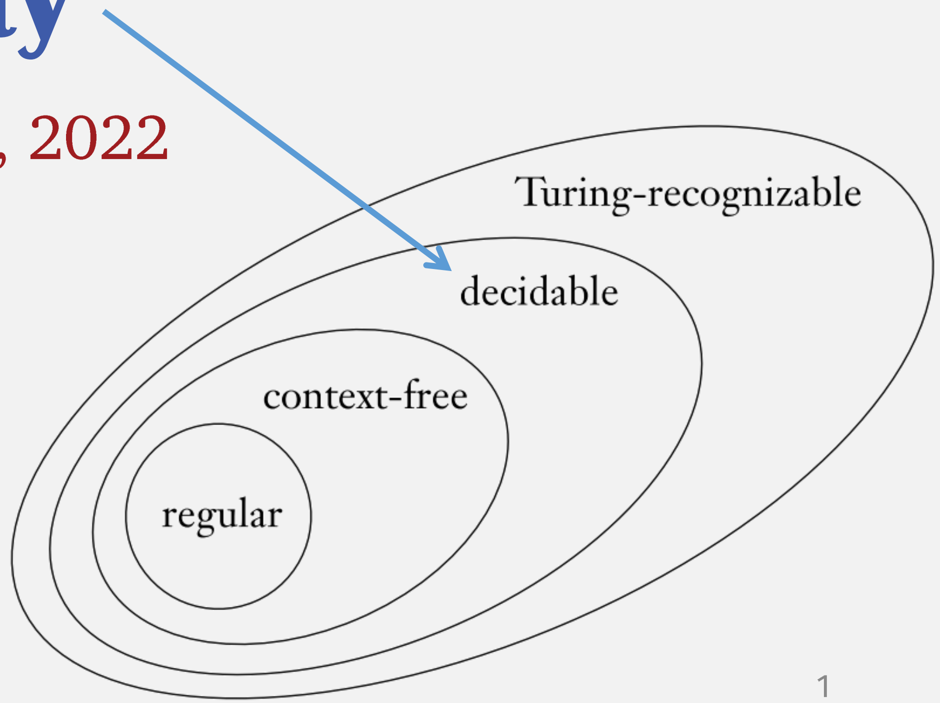
UMB CS 420

Decidability

Thursday, November 11, 2022



Announcements

- HW 7 out
  - due **Mon** 11/7 11:59pm
  - Note the extra day

Last Time: Turing Machines and Algorithms

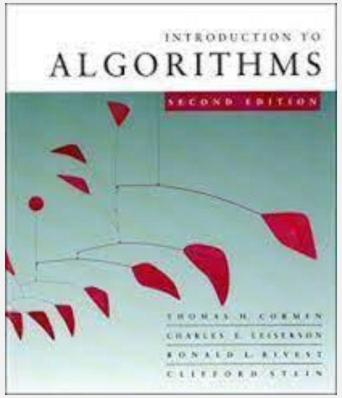
- **Turing Machines** can express any "computation"
  - I.e., a Turing Machine represents a (`Python`, `Java`) program (`function`)!

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

Next

- **Deciders = Algorithms**
  - I.e., an **algorithm** is a program that always **halts**

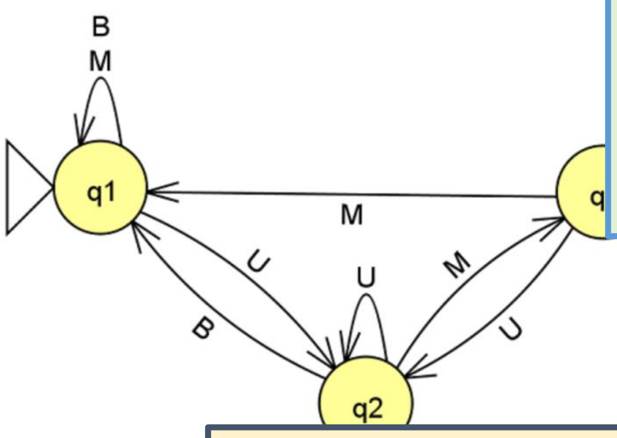
Remember:
TMs = programs



Flashback: HW 1, Problem 1

Doing this HW about computation ...
<u>is itself</u> (meta) computation!

A **Turing Machine** represents computation



Remember:
TMs = programs

1. Come up with a for

Recall that a DFA's
$M = (Q, \Sigma, \delta, q_{start}, F)$.

You may assume that the alphabet contains only the symbols from the diagram.

2. Then for each of the following, say whether the computation represents an accepting computation or not (make sure to review the definition of an accepting computation). If the answer is no, explain why not.:

a. $\hat{\delta}(q1, \texttt{UUMB})$

b. $\hat{\delta}(q1, \texttt{UMMB})$

c. $\hat{\delta}(q2, \texttt{UMBB})$

d. $\hat{\delta}(q3, \varepsilon)$

e. $\hat{\delta}(q3, \texttt{UMASSBOSTON})$

This represents computation by a DFA

Let's do this with a Turing Machine:

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 ...
   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 an accept state

The language of **`DFAaccepts`**

A **Turing Machine** represents ~~computation~~ a language ...

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

But a language is a set of strings?

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

Interlude: Encoding Things into Strings

Definition: A Turing machine's input is always a string

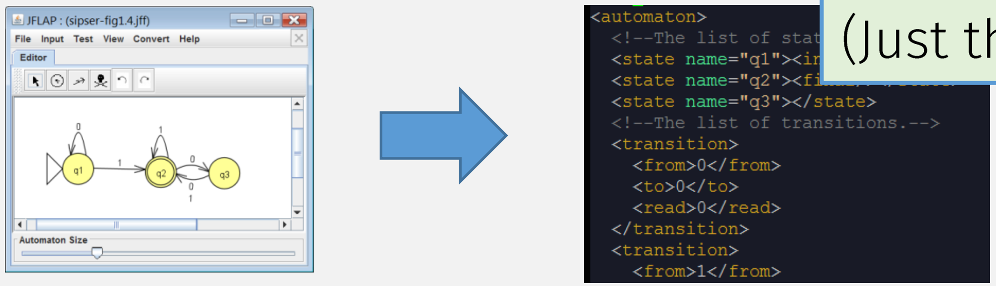
Problem: A TM's (program's) input could also be: list, graph, DFA, …?

Solution: **encode** any TM input as a string

Notation: <SOMETHING> = string **encoding** for SOMETHING

- A tuple combines multiple encodings, e.g., <*B*, *w*> (from prev slide)

Example: Possible string encoding for a DFA?



But in this class, we don't care about what the encoding is!
(Just that there is one)

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

(written as string)

Interlude: High-Level TMs and Encodings

A high-level TM description:

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

2. Assumes TM knows how to parse and extract parts of input

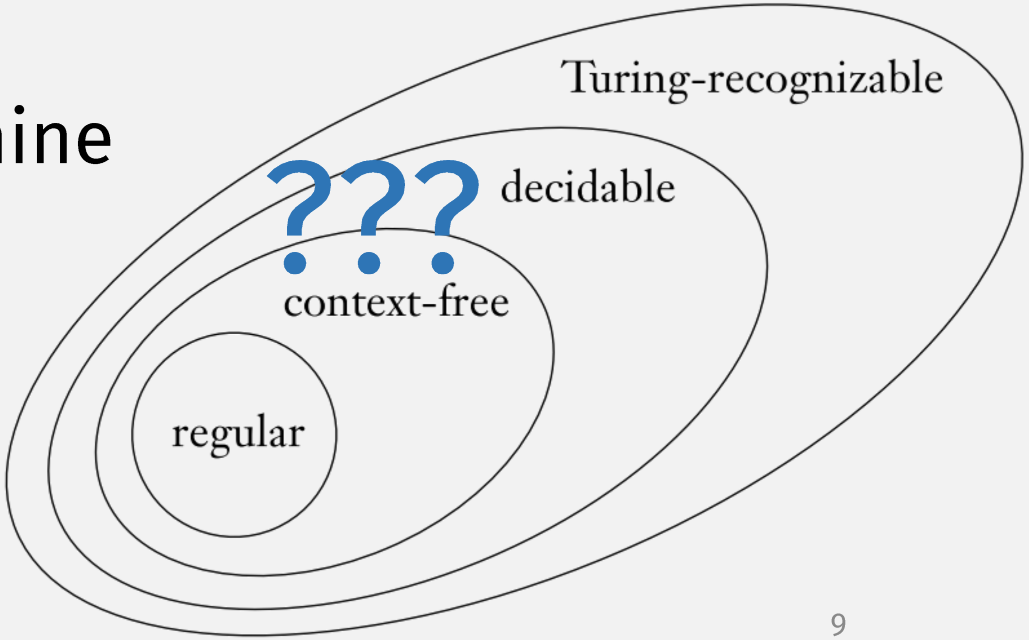
3. Assumes input is a valid encoding
   - Invalid encodings implicitly rejected

The language of **`DFAaccepts`**

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

- A_{DFA} (**`DFAaccepts`**) has a Turing machine
- But **is it a decider** or **recognizer**?
  - I.e., **is it an algorithm**?
- **To show it's an algo, need to** <u>prove</u>:

$$A_{\text{DFA}} \text{ is a decidable language}$$



How to prove that a language is decidable?

Create a Turing machine that **decides** that language!

Remember:

- A **decider** is Turing Machine that always halts
  - I.e., for any input, it either accepts or rejects it.
  - It must never go into an infinite loop

How to Design Deciders

- If TMs = Programs …

  … then **Creating** a TM = Programm**ing**

- E.g., if HW asks "Show that lang L is decidable" …
  - .. you must create a TM that decides L; to do this …
  - … think of how to write a (halting) program that does what you want

- Deciders must also include a **termination argument**:
  - Explains how every step in the TM halts
  - (Pay special attention to loops)

Thm: A_{DFA} is a decidable language

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

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 $q_{\text{current}} = q_{\text{next}}$

Remember:
TMs = programs
Creating TM = programming

Thm: A_{DFA} is a decidable language

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

Decider for A_{DFA} :

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

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

1. Simulate B on input w. ← … which can be used (properly!) in the TM description
2. If the simulation ends in an accept state, *accept*. If it ends in a nonaccepting state, *reject*."

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: **This is a decider** (i.e., **it always halts**) **because the** **input is always finite**, so the loop has finite iterations and always halts

Deciders must have a **termination argument:**

Explains how every step in the TM halts (we typically only care about loops)

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} :

Flashback: NFA→DFA

<u>Have</u>: $N = (Q, \Sigma, \delta, q_0, F)$

<u>Want to</u>: 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 construction is computation

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

Why is this guaranteed to halt?

(Could you implement this conversion **algorithm** as a **program**?)

Thm: A_{NFA} is a decidable language

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

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

Decider for A_{NFA} :

"Calling" another TM. Must give it correct argument type!

$N =$ "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*."

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

- **Step 1 always halts bc there's a finite number of states in an NFA**
- **Step 2 always halts because M is a decider**

How to Design Deciders, Part 2

- If TMs = Programs …

  … then **Creating** a TM = Programm**ing**

- E.g., if HW asks "Show that lang L is decidable" …
  - .. you must create a TM that decides L; to do this …
  - … think of how to write a (halting) program that does what you want
- Deciders must have a termination argument

Hint:

- Previous theorems are a "library" of reusable TMs
- When creating a TM, try to use this "library" to help you
  - Just like libraries are useful when programming!
- E.g., "Library" for DFAs:
  - `NFA→DFA`, `RegExpr→NFA`
  - Union operation, intersect, star, decode, reverse
  - Deciders for: A_{DFA}, A_{NFA}, A_{REX}, …

Thm: A_{REX} is a decidable language

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

Decider:

NOTE: A TM must declare "function" parameters … (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

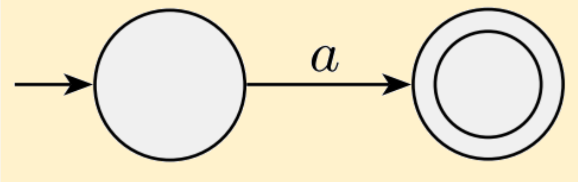
Flashback: RegExpr→NFA

Does this conversion always halt, and why?

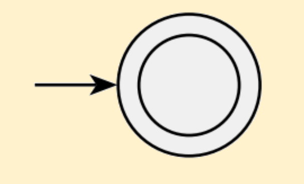
… so guaranteed to always reach base case(s)

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

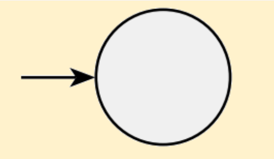
1. a for some a in the alphabet Σ,



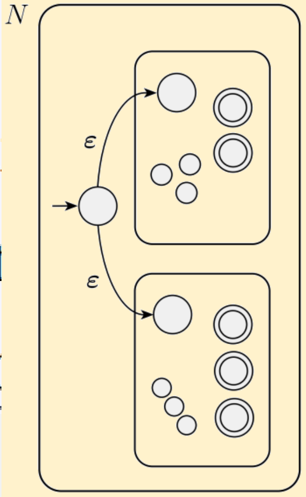
2. ε,

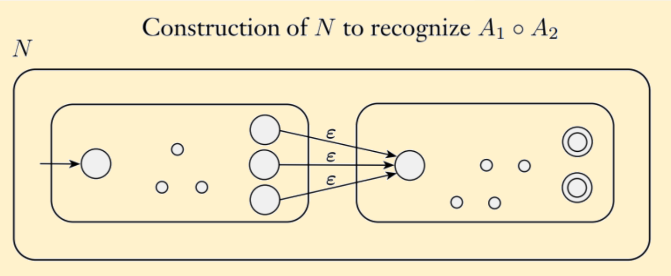


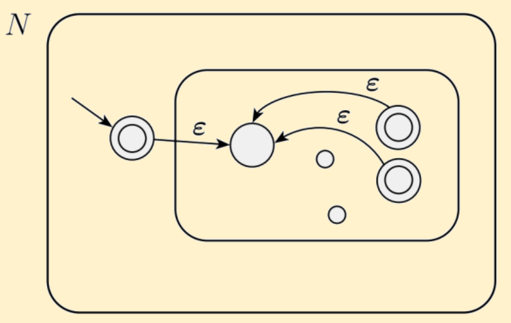
3. $\emptyset$,



4. $(R_1 \cup R_2)$, where R_1 and R_2 a… r e…
5. $(R_1 \circ R_2)$, where R_1 and R_2 a… expressions, or
6. (R_1^*), where R_1 is a regular exp…







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

Thm: A_{REX} is a decidable language

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

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

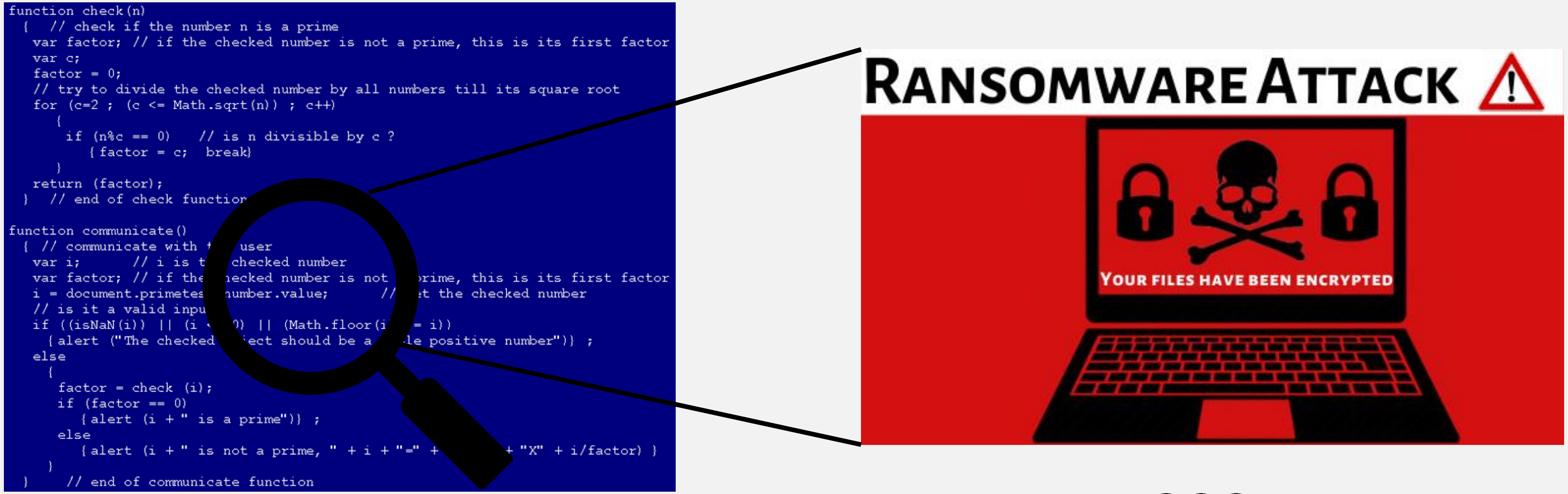
Decidable Languages for DFAs (So Far)

<u>Remember</u>:
TMs = programs
Creating TM = programming
Previous theorems = library

- $A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string}$
  - Deciding TM implements extended DFA δ
- $A_{\text{NFA}} = \{\langle B, w\rangle \mid B \text{ is an NFA that accepts input string } w\}$
  - Deciding TM uses **NFA→DFA** + DFA decider
- $A_{\text{REX}} = \{\langle R, w\rangle \mid R \text{ is a regular expression that generates string } w\}$
  - Deciding TM uses **RegExpr→NFA** + **NFA→DFA** + DFA decider

Flashback: Why Study Algorithms About Computing

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



???

Not possible in general! But …

Predicting What Some Programs Will Do …

What if we look at weaker 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 each DFA must be { }, i.e., the DFA accepts no strings

We determine what is in this language …

… by computing something about the DFA's language (by analyzing its definition)

i.e., by predicting how the DFA will behave

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

Termination argument?

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

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

Note: Machine does not "run" the DFA!

… it computes something about the DFA's language (by analyzing its definition)

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

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

A Naïve Attempt (assume alphabet **{a}**):

1. Run A with input **a**, and B with input **a**
   - **Reject** if results are different, else …
2. Run A with input **aa**, and B with input **aa**
   - **Reject** if results are different, else …
3. Run A with input **aaa**, and B with input **aaa**
   - **Reject** if results are different, else …

- …

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

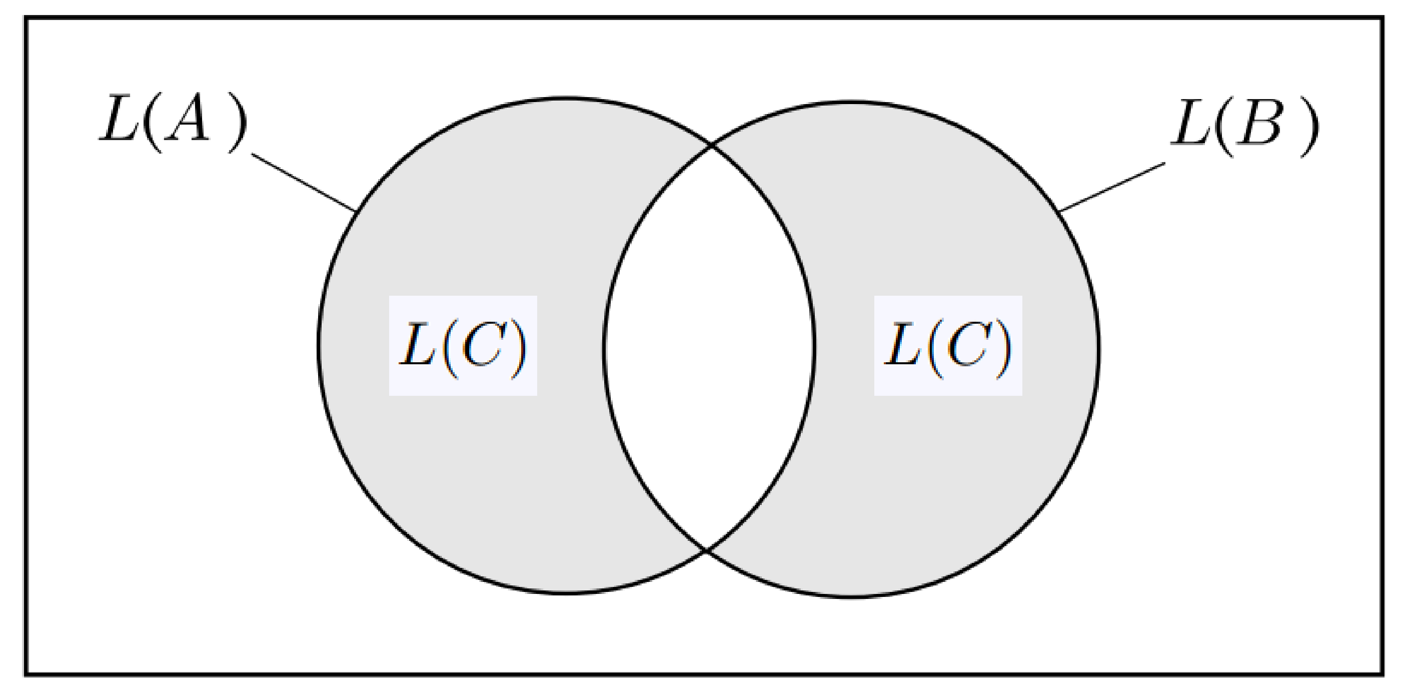
Can we compute this without running the DFAs?

Thm: EQ_{DFA} is a decidable language

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

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

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

<u>NOTE: This only works because:</u> **negation**, i.e., set complement, and **intersection is closed** for regular languages

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*."

Predicting What <u>Some</u> 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.

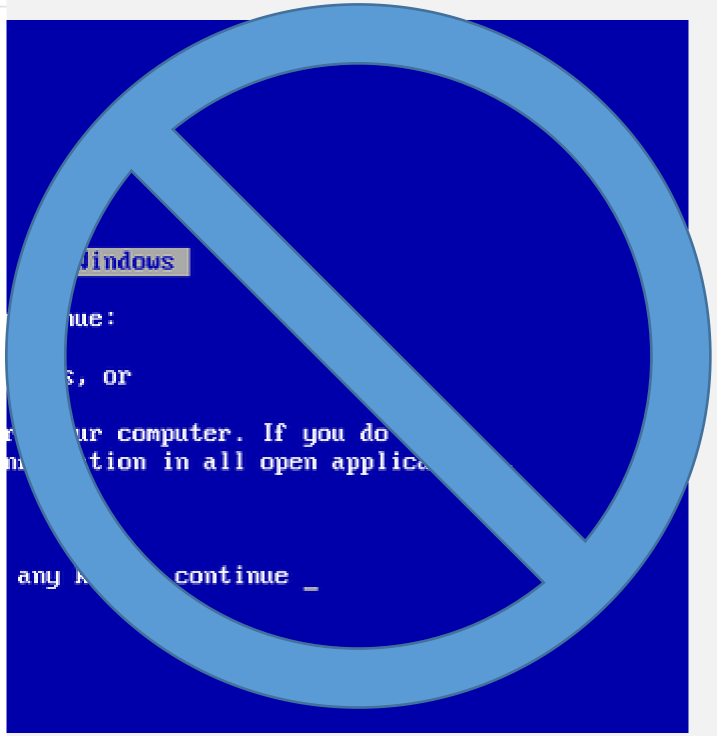
"Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we're building tools that can do actual proof about the software and how it works in order to guarantee the reliability." **Bill 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: Decidable DFA Langs (i.e., algorithms)

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

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

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

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

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

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

Next Time: Algorithms (Decider TM) for CFLs?

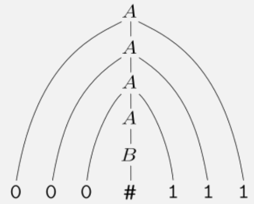
- What can we predict about CFGs or PDAs?

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 a is very practically important problem …
- … equivalent to:
  - Is there an **algorithm** to parse 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 is a rule like: $\mathbf{S} \rightarrow \mathbf{0S}$ or $\mathbf{S} \rightarrow \mathbf{S}$
  - 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?

Check-in Quiz 11/3

On gradescope