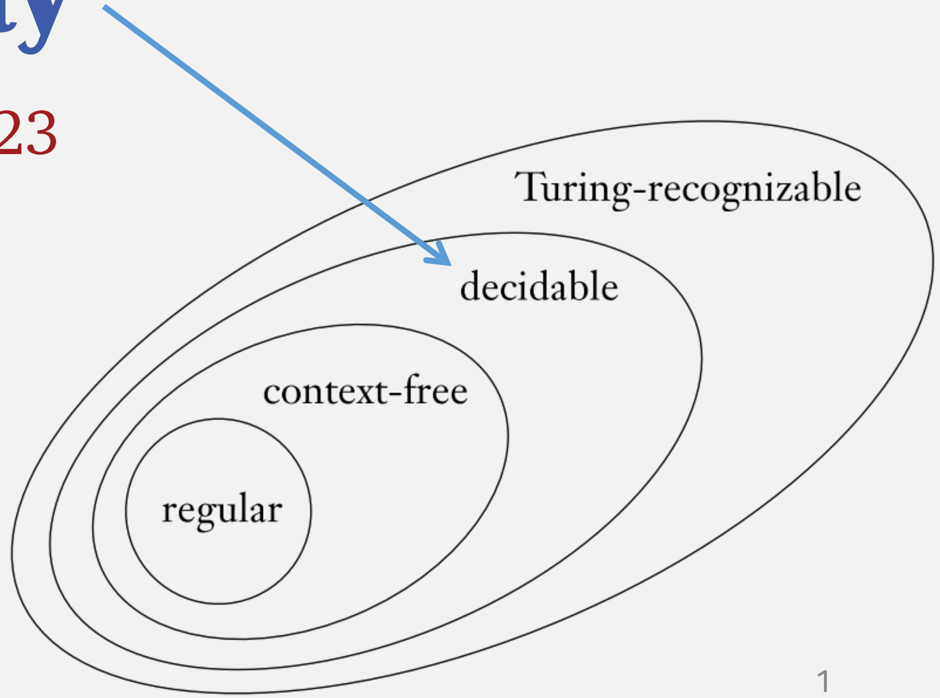
UMB CS 420

Decidability

Monday, April 3, 2023



Announcements

- HW 7 extended
  - ~~Due Sun 4/2 11:59pm~~
  - Due Tue 4/4 11:59pm

- HW 8 out Wed 4/5
  - Due Tue 4/11 11:59pm

Quiz Preview

- A decider TM definition requires specifying which of the following parts?

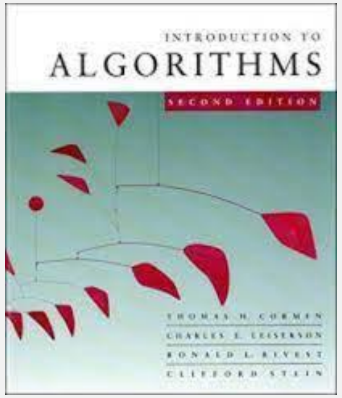
Last Time: Turing Machines and Algorithms

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

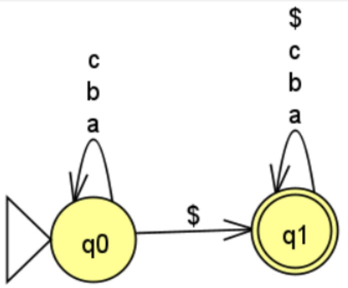
- 2 classes of Turing Machines
  - **Recognizers** may loop forever
  - **Deciders** always halt

Today

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



Flashback: HW 1, Problem 1



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.
3. Is the empty string, ε, in the language of the DFA?
4. Come up wit

   Recall that a
   $M = (Q, \Sigma$

   You may assu
5. 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. $\hat{\delta}(q0, \texttt{a\$b})$

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

   c. $\hat{\delta}(q0, \texttt{abc})$

   d. $\hat{\delta}(q0, \texttt{cd\$})$

Remember:
TMs = program (functions)

You had to figure out a DFA's computation

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

What kind of ~~computation~~ language is it?

Could you write a <u>program</u> (<u>function</u>) to do it?

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 an accept state

This is just checks for an **accepting computation** $\hat{\delta}(q_0, w) \in F$!!

The language of **`DFAaccepts`**

The set of strings that a **Turing Machine** accepts is a **language** ...

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

Is this language 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: We sometimes want TM's (program's) input to be something else …

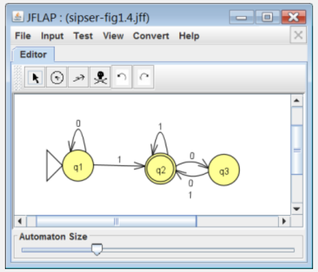
- **set, graph, DFA, …?**

Solution: allow **encoding** other kinds of 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?




```
<automaton>
  <!--The list of states.-->
  <state name="q1"><initial/></st
  <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>
```


It doesn't matter!
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. 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 <u>how</u> input string is encoded

3. Assumes TM <u>knows</u> how to parse and extract parts of input

   Description of M can refer to B's $(Q, \Sigma, \delta, q_0, F)$

4. Assumes input is a <u>valid</u> encoding
   - Invalid encodings implicitly rejected

`DFAaccepts` as a TM recognizing A_{DFA}

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

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

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 an accept state

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

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

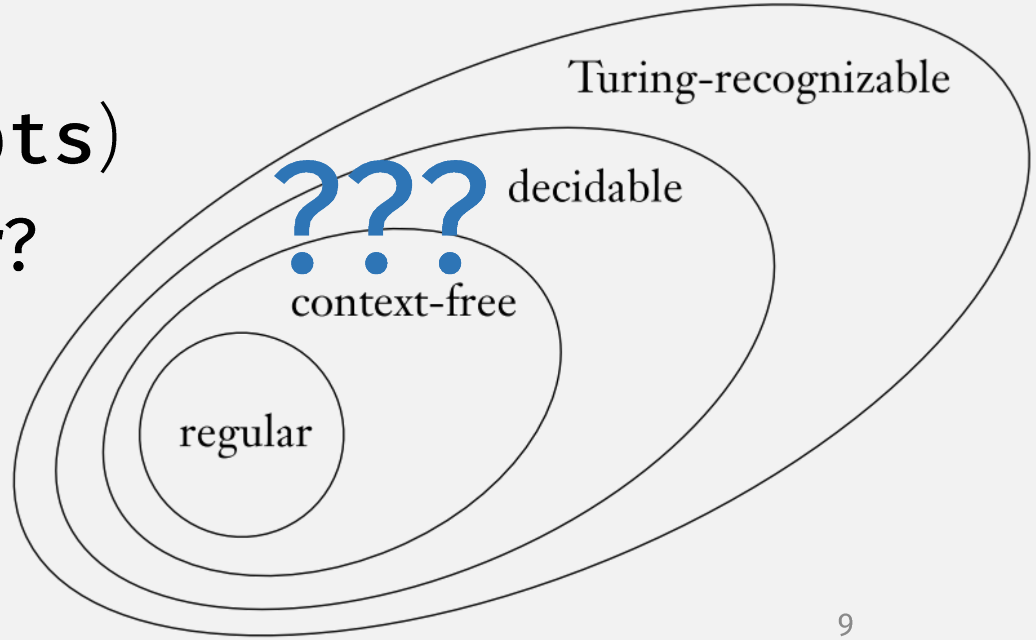
1) Define "current" state q_{current} = start state q_0
2) For each input char a_i … in w
   a) Define $q_{\text{next}} = \delta(q_{\text{current}}, a_i)$
   b) Set $q_{\text{current}} = q_{\text{next}}$
3) **Accept** if q_{current} is an accept state

The language of **`DFAaccepts`**

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

- A_{DFA} has a Turing machine (**`DFAaccepts`**)
- But is that 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
2. Define **decider** M = On input w … , M **decides** L

Key step

3. L is a **decidable** language

Justifications

1. Definition of **decidable** langs
2. See M def, and examples
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

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

Decider input must match strings in the language!

Key step

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*."

"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 $q_{\text{next}} = \delta(q_{\text{current}}, x)$
  - Set $q_{\text{current}} = q_{\text{next}}$

Remember:
TM ~ program
Creating TM ~ programming

Thm: A_{DFA} is a decidable language

$A_{\mathsf{DFA}} = \{\langle B, w\rangle |\ B$ 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!)

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

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

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 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_{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*."

Termination Argument: **Step #2 always halts because we are checking only the state of the result** (there's no loop)

Deciders must have a **termination argument:**
Explains how <u>every step</u> in the TM halts (we typically only care about 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*."

Example String	In the A_{DFA} language?	Accepted by *M*?
<*B*, *w*> where *B* accepts *w*	Yes	Yes
<*B*, *w*> where *B* rejects *w*	No	No

Columns #2 and #3 must match

A good set of examples needs some Yes's and some No's

This is what a "See Examples" justification should look like!

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

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

This conversion is computation

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

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

Turing Machine NFA→DFA

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 $a \in \Sigma$,

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

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

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

Why is this guaranteed to halt?

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

Thm: A_{NFA} is a decidable language

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

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

Decider for A_{NFA} :

"Calling" another TM. Must give correct arg 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*."

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 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

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

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

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

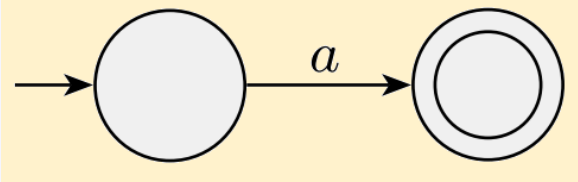
Flashback: RegExpr→NFA

Does this conversion always halt, and why?

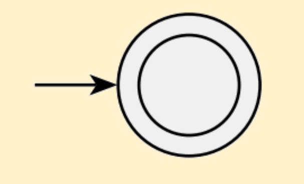
… so guaranteed to always reach base case(s)

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

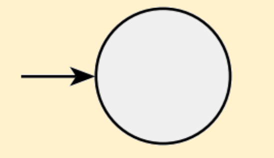
1. a for some a in the alphabet Σ,



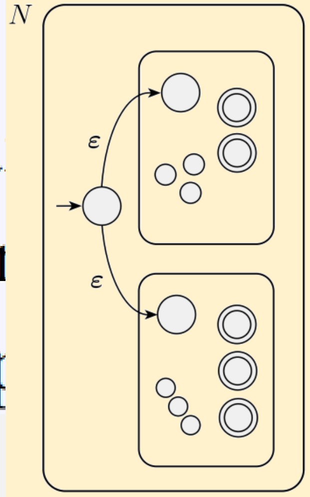
2. ε,

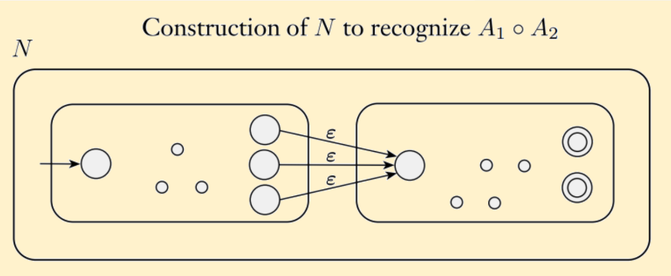


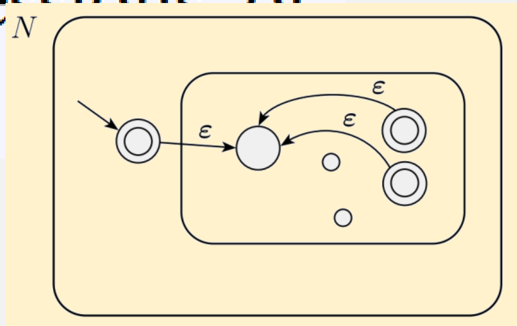
3. $\emptyset$,



4. $(R_1 \cup R_2)$, where R_1 and R_2 a
5. $(R_1 \circ R_2)$, where R_1 and R_2 ar 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

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

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** (from prev slide)
2. Run TM N on input $\langle A, w\rangle$.
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)

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

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

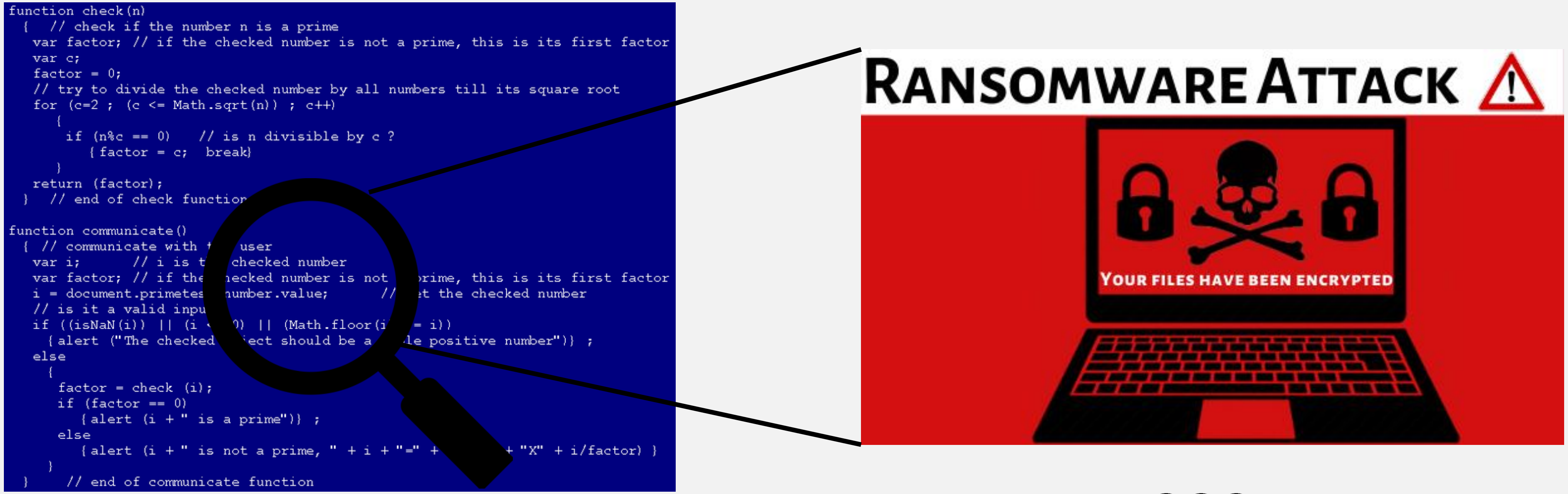
- $A_{\mathsf{NFA}} = \{\langle B, w\rangle \mid B \text{ is an NFA that accepts input string } w\}$
  - Deciding TM uses **NFA→DFA** + DFA decider

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

Flashback: Why Study Algorithms About Computing

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

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

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)

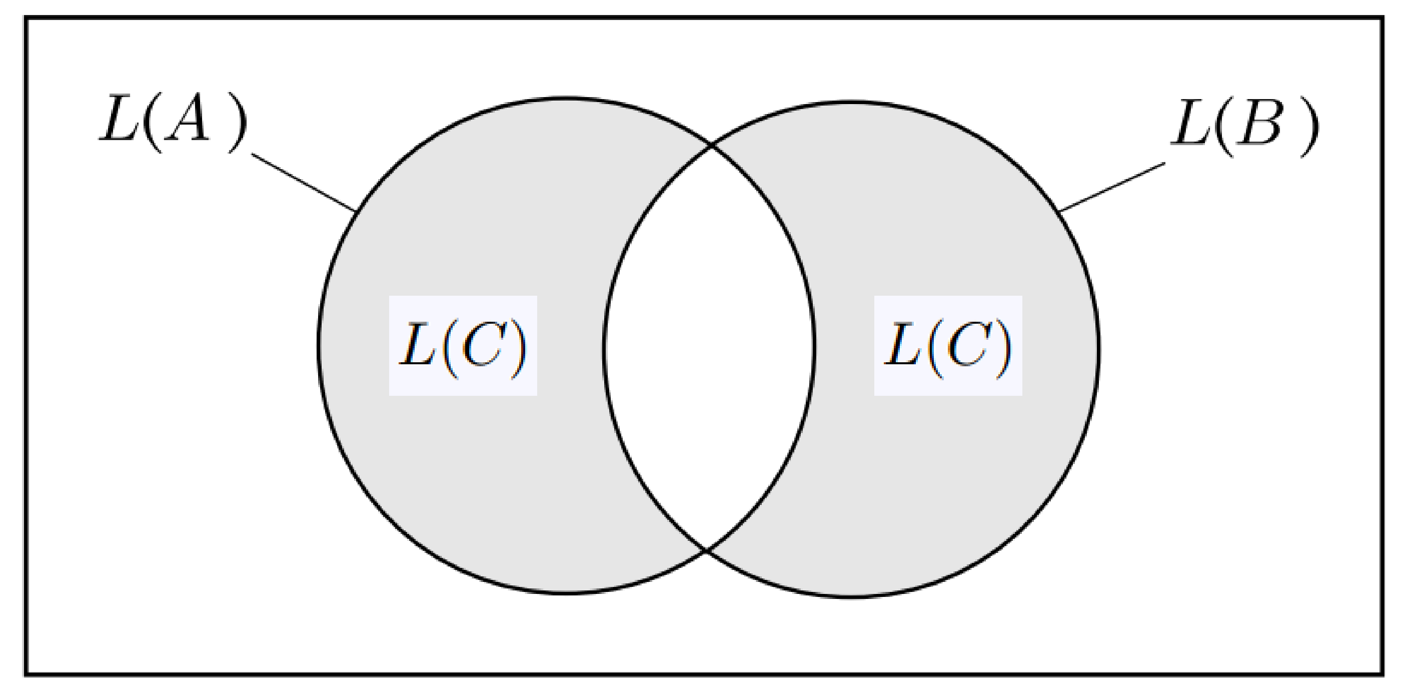
Can we compute this without running the DFAs?

Thm: EQ_{DFA} is a decidable language

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

Trick: Use Symmetric Difference

Symmetric Difference



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

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

<u>Thm</u>: 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.

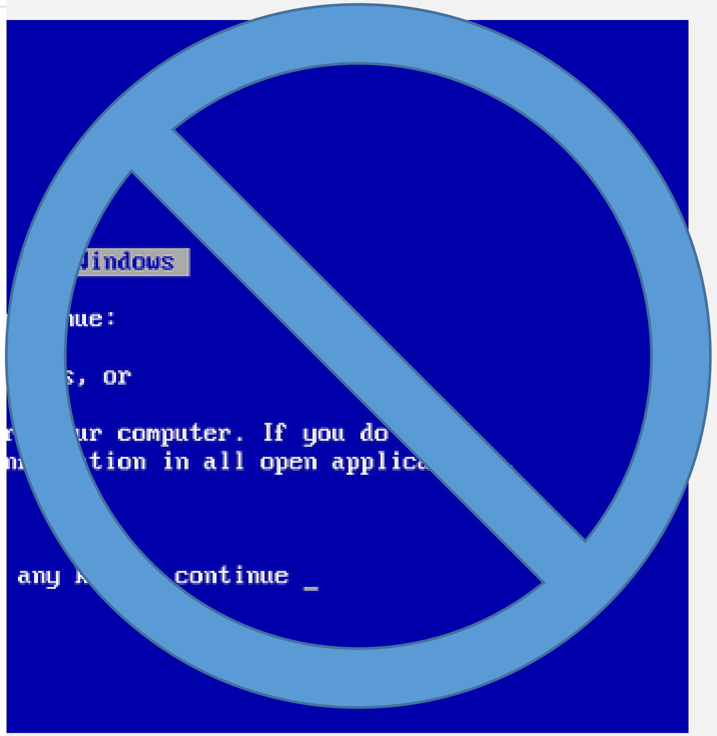
"Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we're building tools that can do actual proof about the software and how it works in order to guarantee the reliability." **Bill 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:
TM ~ program
Creating TM ~ programming
Previous theorems ~ library

Next Time: Algorithms (Decider TM) for CFLs?

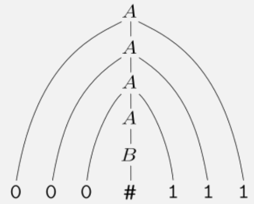
- What can we predict about CFGs or PDAs?

Thm: A_{CFG} is a decidable language

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

- 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 4/3

On gradescope