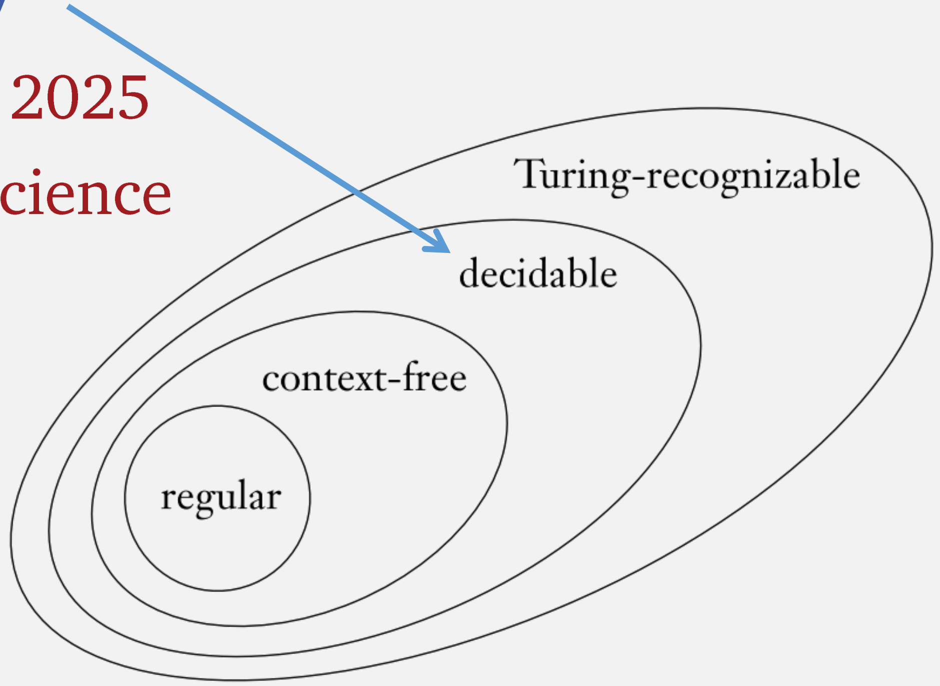
CS 420 / CS 620

Decidability

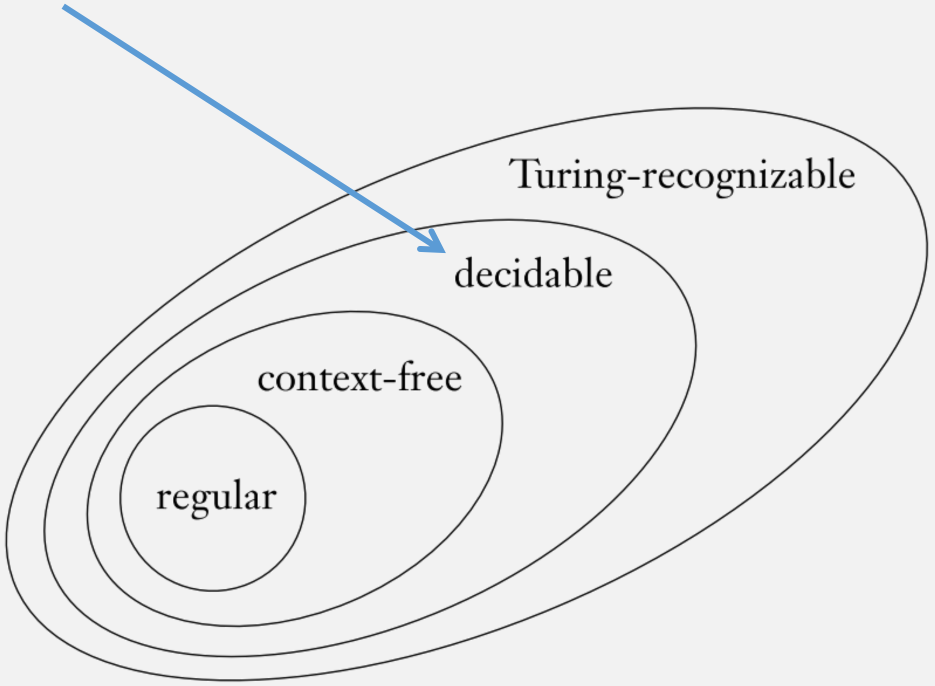
Wednesday, November 5, 2025

UMass Boston Computer Science



Announcements

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

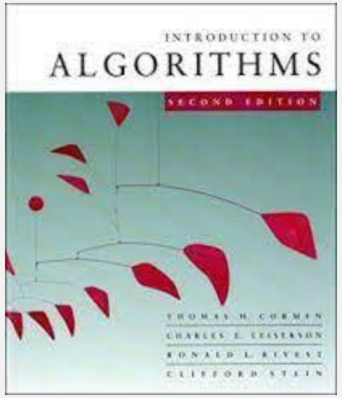


Previously: Turing Machines and Algorithms

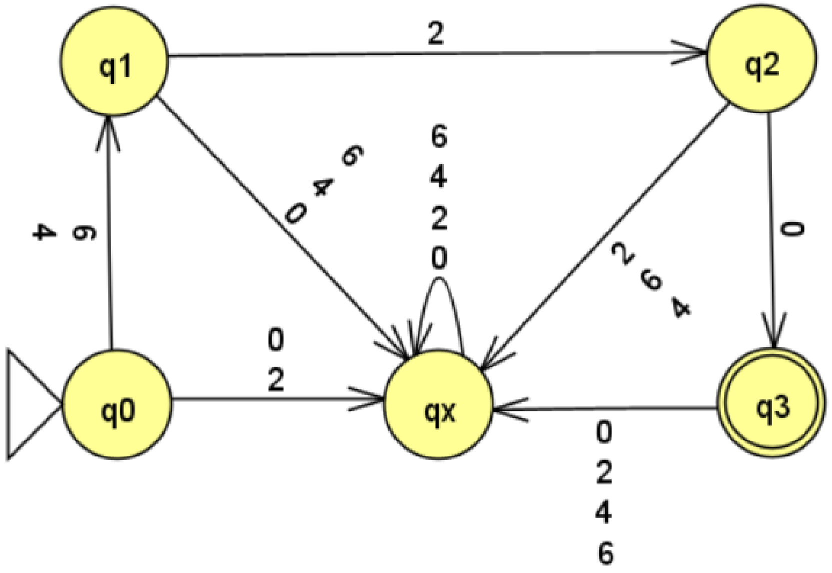
- **Turing Machines** can **express more "computation"** (than other prev machines)
  - Analogy: a **TM models** a (**`Python`**, **`Java`**) **program** (**`function`**)!

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

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



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**) … **<u>is itself</u>** (meta) **computation!**

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

Could you **write** a <u>**program**</u> (<u>**function**</u>) to **compute this algorithm?**

A function: **`DFAaccepts`**(B, w) "returns" TRUE if DFA B=(Q,Σ,δ_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`** $q_{\text{next}} = \delta_B(q_{\text{current}}, a_i)$ "**`get`** δ_B"
   b) **`Set`** $q_{\text{current}} = q_{\text{next}}$
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 **accepts!**

The language of **`DFAaccepts`**

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

Def: a **language** is a set of strings

How is this a set of strings???

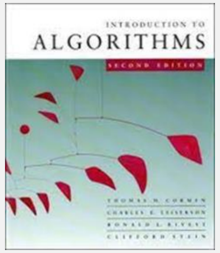
A function: **`DFAaccepts`**(B, w) "returns" TRUE if DFA B accepts string w

Interlude: Encoding Things into Strings

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

<u>Problem</u>: We sometimes want: **TM's** (program's) <u>**input**</u> to be "<u>**something else**</u>" …

- **set, graph, DFA, …?**



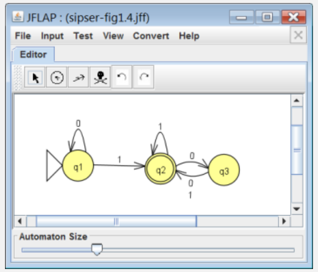
<u>Solution</u>: allow **encoding** "other kinds of input" <u>as a string</u>

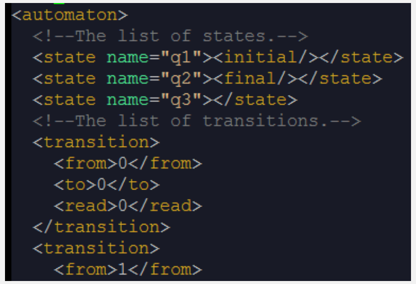
<u>Notation</u>: <SOMETHING> = string **encoding** for SOMETHING

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

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

<u>Example</u>: **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>
```

<u>Or</u>:

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

<u>(written as string)</u>

Interlude: High-Level TMs and Encodings

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

1. <u>Needs</u> to **say** the **type** of its **input**
   - E.g., **graph**, $\boxed{\textbf{DFA,}}$ etc.

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

2. <u>Doesn't need</u> to **say** <u>how</u> **input string** is **encoded**

   e.g.,

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

   - <u>Assume</u> **1: input is** a <u>valid</u> **encoding**
     - Invalid encodings **implicitly rejected**

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

   - <u>Assume</u> **2: TM** <u>knows</u> <u>how</u> to **parse** and **extract parts of input**

   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**

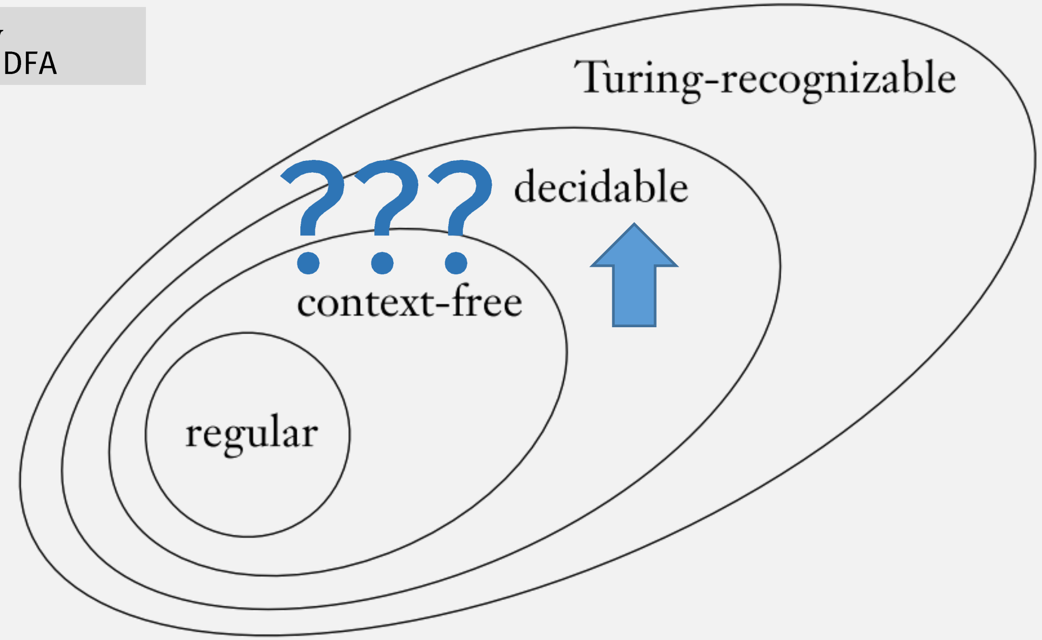
What "kind" of ~~computation~~ language is it?

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

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 <u>prove</u>:**

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



How to prove that a language is decidable?

How to prove that a language is decidable?

Statements	**Justifications**
1. If a **decider** decides a lang L, then L is a **decidable** lang	1. Definition of **decidable** langs
2. Define **decider** M = On input w … , M **decides** L	2. See M TM def, and Examples Table
3. L is a **decidable** language	3. By statements #1 and #2

Key step

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 <u>every step</u> in the TM halts
  - (Pay special attention to loops)

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

Thm: A_{DFA} is a decidable language

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

Decider input must match (encodings of) 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 $q_{\text{current}} = q_{\text{next}}$

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

Remember:

TM ~ program

Creating TM ~ programming

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} :

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

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: <u>Step #1</u> always halts because: **the simulation <u>input is always finite</u>,** so the <u>loop</u> has <u>finite iterations</u> **and always halts**

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

Termination Argument

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:
determining **accept** requires checking finite number of accept states

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

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

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?
$<D_1, w_1>$	Accept	Accept	Yes
$<D_2, w_2>$	Reject	Reject	No

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

Columns must match!

(especially important when machine could loop)

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 $M = (Q', \Sigma, \delta', q_0', F')$

This conversion is computation!

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

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

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

Turing Machine **NFA→DFA**

<u>New TM Variation!</u>
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. <u>Write to the tape:</u> DFA $M = (Q', \Sigma, \delta', {q_0}', F')$

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

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

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

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

$F' = \{R \in Q' |\ R$ 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

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

Decider for A_{NFA} :

"Calling" another TM. Must give correct arg type!

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

$N =$ "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

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**
  - $\mathrm{UNION}_{\mathrm{DFA}}$, $\mathrm{STAR}_{\mathrm{PDA}}$, ENC, 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 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

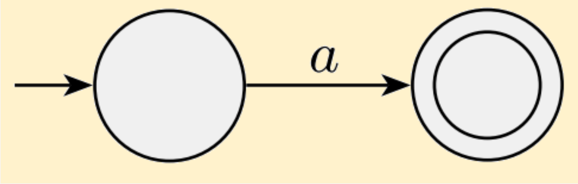
RegExpr→NFA

Does this conversion **always halt**, and **why?**

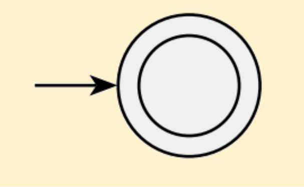
… so **guaranteed to always reach base case**(s)

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

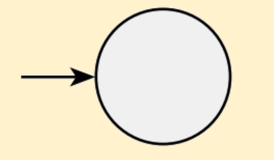
1. a for some a in the alphabet Σ,



2. ε,



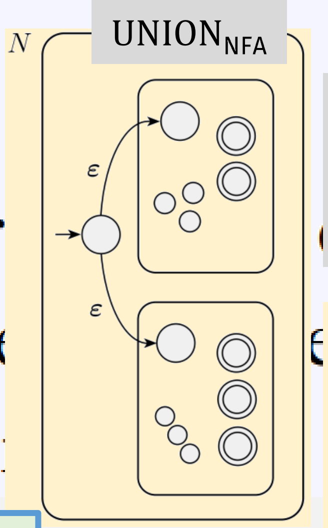
3. $\emptyset$,



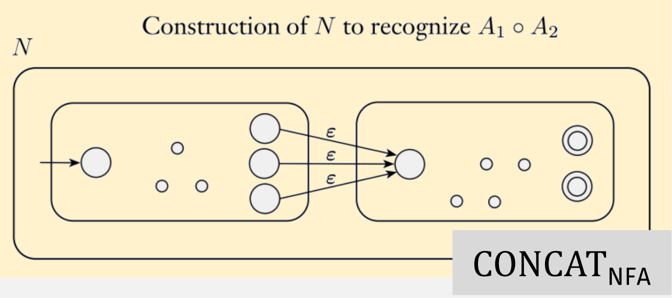
4. $(R_1 \cup R_2)$, where R_1 and R_2 ar

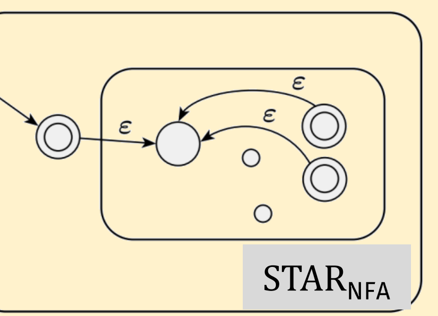
5. $(R_1 \circ R_2)$, where R_1 and R_2 are

6. (R_1^*), where R_1 is a regular exp



RegExpr→NFA($R_1 \cup R_2$) =
UNION$_{\text{NFA}}$(**RegExpr→NFA**(R_1),
RegExpr→NFA(R_2))





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

<u>Thm</u>: 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:**

- <u>Step 1:</u> 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**
- <u>Step 2:</u> always halts because: ***N* is a decider**

Decidable Languages About DFAs

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

- $A_{\mathsf{DFA}} = \{\langle B, w\rangle |\ B \text{ is a DFA that accepts input string } w\}$
  - Decider TM: implements B **DFA's extended δ fn** algorithm
- $A_{\mathsf{NFA}} = \{\langle B, w\rangle |\ B \text{ is an NFA that accepts input string } w\}$
  - Decider TM: uses **NFA→DFA** algorithm + A_{DFA} **decider**
- $A_{\mathsf{REX}} = \{\langle R, w\rangle |\ R \text{ is a regular expression that generates string } w\}$
  - Decider TM: uses **RegExpr→NFA** algorithm + A_{NFA} **decider**

Flashback: Why Study Algorithms About Computing

To predict what programs will do

(without running them!)

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

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

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., **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)

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

Analogy
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

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

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~~

Termination argument?

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

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

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

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

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"**?

<u>A Naïve Attempt (assume alphabet **{a}**):</u>

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

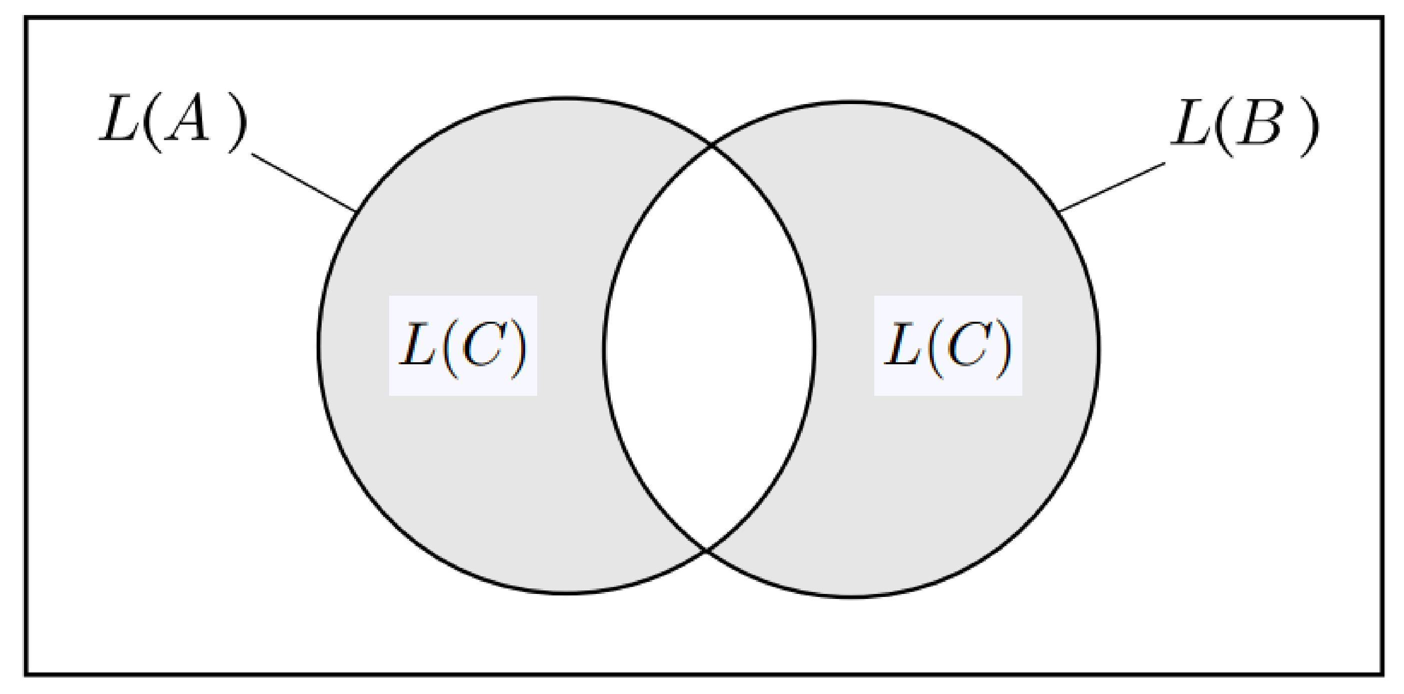
Can we **compute this <u>without</u> running the DFAs**, i.e., by only **examining the DFA's "source code"**?

Thm: EQ_{DFA} is a decidable language

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

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

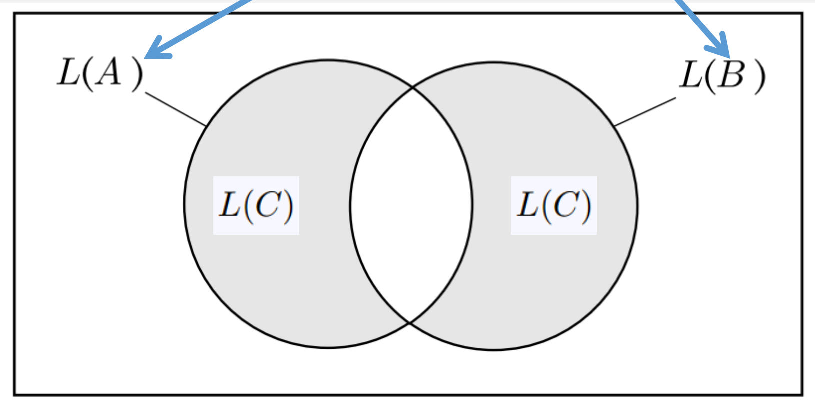
$$EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$$

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



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

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?

From 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.

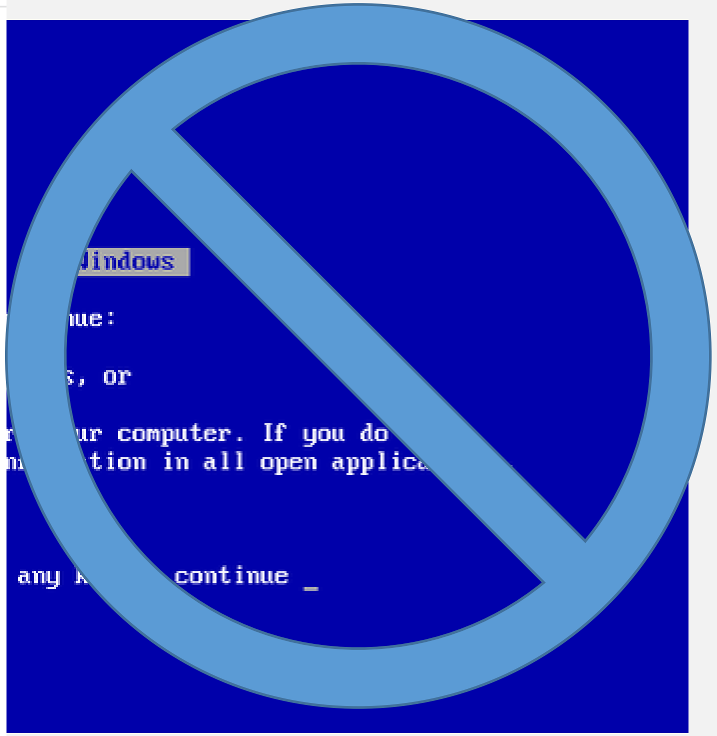
"Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we're building tools that can do actual proof about the software and how it works in order to guarantee the reliability." **Bill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

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

Lang of the DFA

- $EQ_{\mathsf{DFA}} = \{\langle A, B \rangle |\ A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$
  -  **Decider**: Uses complement and intersection closure construction + E_{DFA} decider

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

Next: Algorithms (Decider TMs) for CFLs?

- What can we predict about CFGs or PDAs?