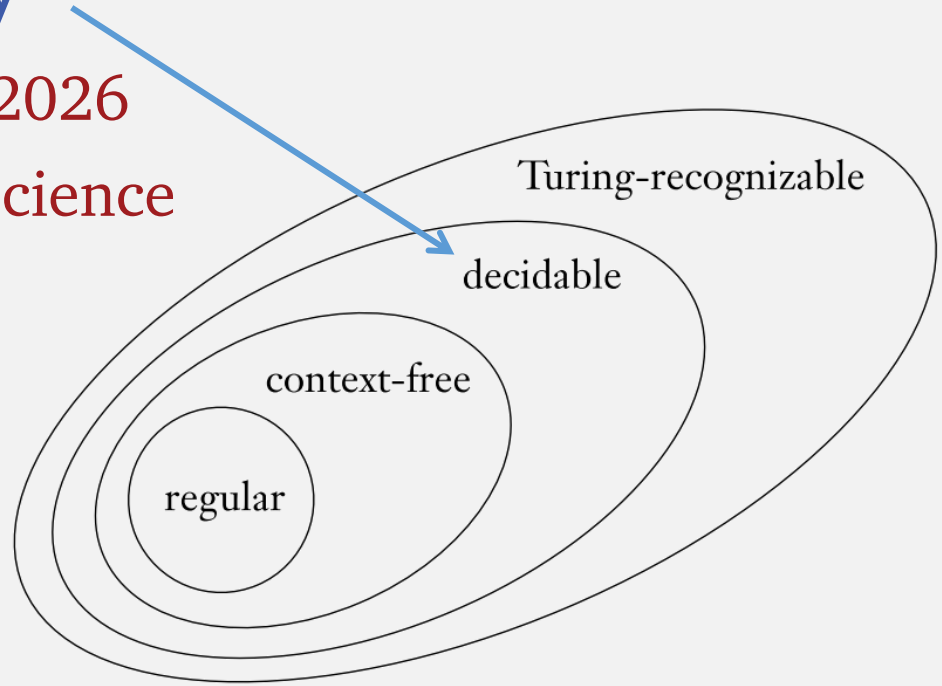
CS 420

Decidability

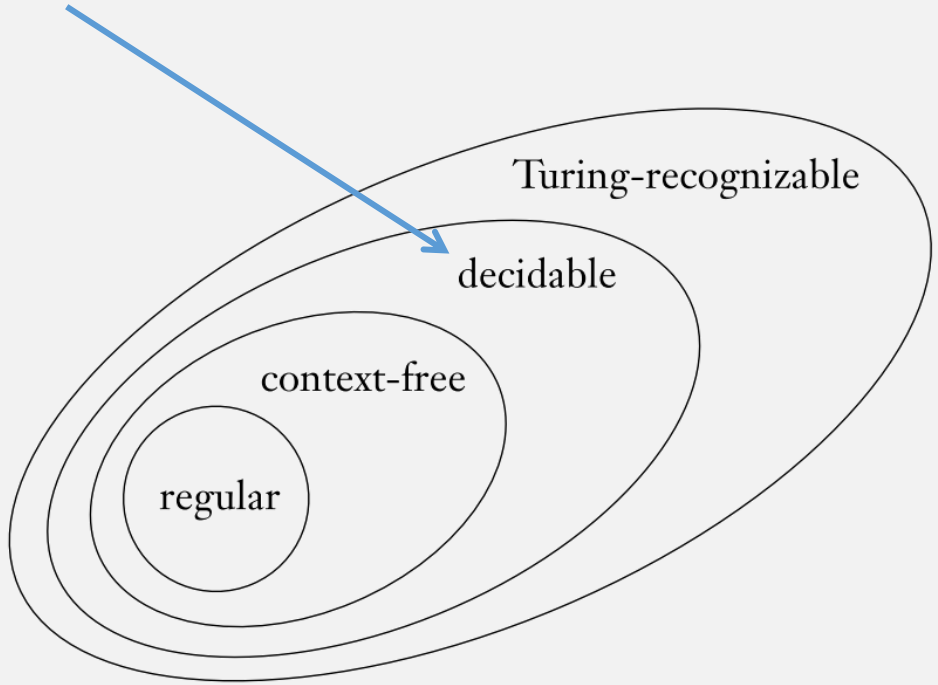
Tuesday, November 10, 2026

UMass Boston Computer Science



Announcements

- HW 8
  - Due: **Thurs 11/12 12:30pm EST**



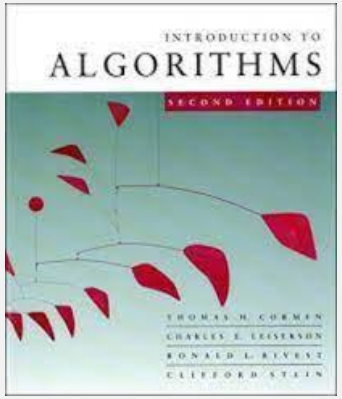
Previously: Turing Machines and Algorithms

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

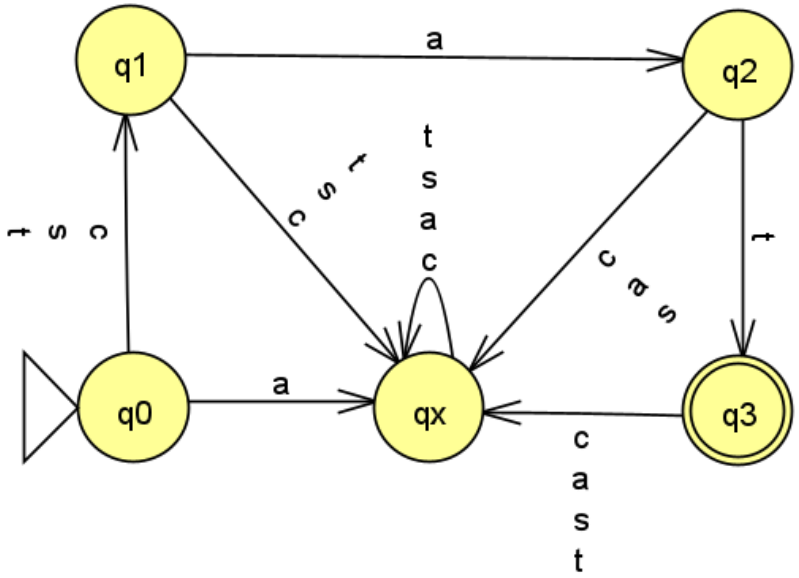
- 2 classes of Turing Machines
  - **Recognizers**: may loop forever
  - **Deciders**: always halt

Today

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



Flashback: HW 1, Problem 2



1. Come up with 2 strings that are accepted by the DFA. These strings are said 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.

Your task:
"compute" how a DFA computes

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 it?

A function: **DFAaccepts**(B,w) "returns" TRUE if DFA B accepts w ... where $(Q,\Sigma,\delta_B,q_0,F)$ = `get-DFA-parts` B

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

This is "**computing**": whether we have **accepting computation** $\hat{\delta}(q_0, w) \in F$!!

The language of **DFAaccepts**

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

How is this language a set of strings???

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

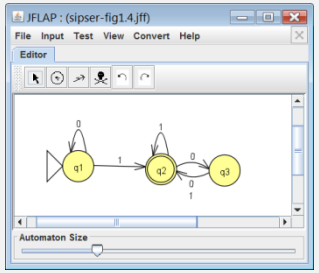
Solution: allow **encoding** "other kinds of input" as a string

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

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

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

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>
```

Or:

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

(written as string)

Interlude: High-Level TMs and Encodings

A high-level TM description:

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

   let 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 input** is a valid **encoding**
   - Invalid **encodings** implicitly **rejected**

   Definition of TM M can write: $(Q, \Sigma, \delta, q_0, F)$ = `get-DFA-parts` B

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

- Can parse and extract parts of input

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

TM M_{DFA} =

"On Definition of B IS-A DFA AND w IS-A string:
TM M ca where $(Q, \Sigma, \delta, q_0, F)$ = `get-DFA-parts` B

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

The language of **DFAaccepts**

language

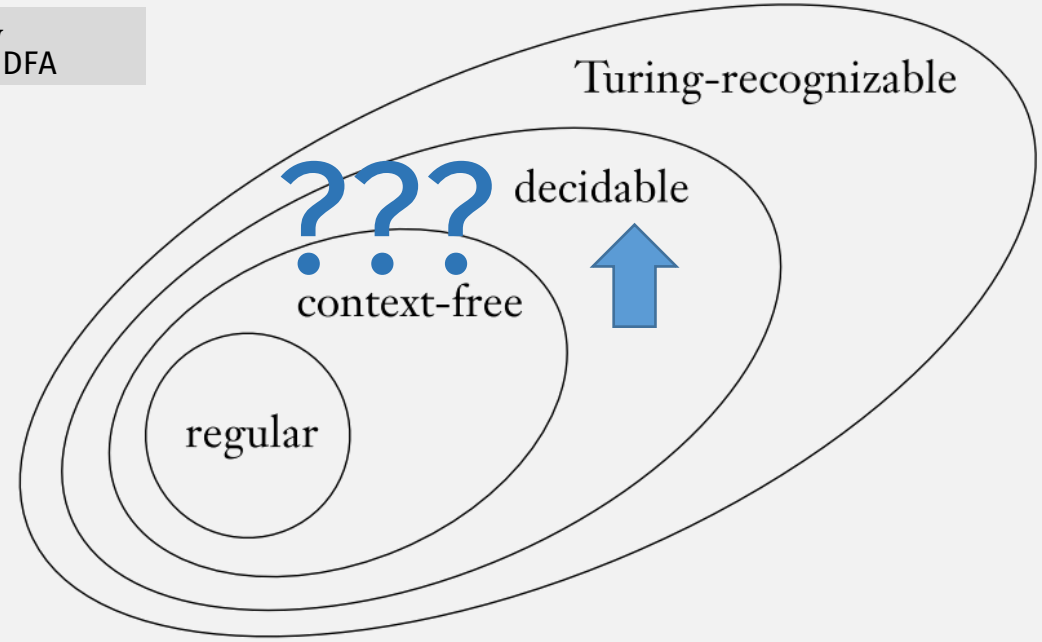
What "kind" of computation is it?

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

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 <u>prove</u>:

A_{DFA} is a decidable language



How to prove that a language is decidable?

How to prove that a language is decidable?

Statements

Key step

1. let M = On input w ... IS-A **Decider**
2. M **decides** L
3. L IS-A **Decidable** Language

DefDecideLang

IF M IS-A **Decider** AND M **decides** L
THEN L IS-A **Decidable** Language

Justifications

1. **BY** DefDecider
2. **BY** ExTable1 COMPARING M,L
3. **APPLY** DefDecideLang **TO** **mk-AND-proof** Proof1 Proof2

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., let M = **On input** $\langle B, w \rangle$**,** 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

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

Key step

Decider for A_{DFA} :

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

let 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" B on input w =
- let $(Q, \Sigma, \delta, q_0, F)$ = `get-DFA-parts` B
- let ("current" state) q_{current} = (start state) q_0
- for each input char x in w ...
  - let $q_{\text{next}} = \delta(q_{\text{current}}, x)$
  - 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

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

Decider for A_{DFA} :

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

let $M =$ 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} :

let 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" B on input w =
- let $(Q, \Sigma, \delta, q_0, F)$ = `get-DFA-parts` B
- let ("current" state) q_{current} = (start state) q_0
- for each input char x in w ...
  - let $q_{\text{next}} = \delta(q_{\text{current}}, x)$
  - 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

DefDecider

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} :

let 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)

How to prove that a language is decidable?

Statements

☑ 1. let M = On input w … IS-A **Decider**

2. M **decides** L

3. L IS-A **Decidable** Language

DefDecideLang

IF M IS-A **Decider** AND M **decides** L
THEN L IS-A **Decidable** Language

Justifications

1. **BY** DefDecider

(don't forget **Termination Argument!**)

2. **BY** A_{DFA}ExTable COMPARING M,L

TODO!

3. **APPLY** DefDecideLang **TO** `mk-AND-proof` Proof1 Proof2

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} :

let 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*."

(helper) "called" machine column(s)

"Actual" behavior

"Expected" behavior

let A_{DFA}ExTable =

Str <B,w>	B on input w?	M on <B,w>	In A_{DFA} lang?
$\langle D_1, w_1\rangle$	Accept	Accept	Yes
$\langle D_2, w_2\rangle$	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)

can this column loop???

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

Examples Table

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!

let N = "On input $\langle B, w\rangle$, where B IS-A 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

CONVERT$_{\text{NFA-DFA}}$: NFA → DFA

CONVERT$_{\text{NFA-DFA}}$ N = M where

$(Q_{\text{NFA}}, \Sigma, \delta_{\text{NFA}}, q_{0\text{NFA}}, F_{\text{NFA}})$ = **get-NFA-parts** N

M = **mk-DFA** $(Q_{\text{DFA}}, \Sigma, \delta_{\text{DFA}}, q_{0\text{DFA}}, F_{\text{DFA}})$ where

$$Q_{\text{DFA}} = \mathcal{P}(Q_{\text{NFA}})$$

For $qs \in Q_{\text{DFA}}$ and $a \in \Sigma$

$$\delta_{\text{DFA}}(qs, a) = \bigcup_{s \in S} \varepsilon\text{-REACHABLE}(s) \quad \text{where } S = \bigcup_{q \in qs} \delta_{\text{NFA}}(q, a)$$

$$q_{0\text{DFA}} = \varepsilon\text{-REACHABLE}(q_{0\text{NFA}})$$

$$F_{\text{DFA}} = \{\, qs \in Q_{\text{DFA}} \mid qs \text{ contains accept state of } N \,\}$$

let CONVERT$_{\text{NFA-DFA}}$ = On input <N>, where N IS-A NFA

1. Write to the tape: M where

$(Q_{\text{NFA}}, \Sigma, \delta_{\text{NFA}}, q_{0\text{NFA}}, F_{\text{NFA}})$ = `get-NFA-parts` N

M = **mk-DFA** $(Q_{\text{DFA}}, \Sigma, \delta_{\text{DFA}}, q_{0\text{DFA}}, F_{\text{DFA}})$ where

$Q_{\text{DFA}} = \mathcal{P}(Q_{\text{NFA}})$

For $qs \in Q_{\text{DFA}}$ and $a \in \Sigma$

$\delta_{\text{DFA}}(qs, a) = \bigcup_{s \in S} \varepsilon\text{-REACHABLE}(s)$ where $S = \bigcup_{q \in qs} \delta_{\text{NFA}}(q, a)$

$q_{0\text{DFA}} = \varepsilon\text{-REACHABLE}(q_{0\text{NFA}})$

$F_{\text{DFA}} = \{ qs \in Q_{\text{DFA}} \mid qs \text{ contains accept state of } N \}$

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

As a TM!

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} :

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

N = "On input $\langle B, w\rangle$, where B IS-A NFA AND w IS-A string:

1. Convert NFA B to an equivalent DFA C, using the procedure CONVERT$_{\text{NFA-DFA}}$
2. Run TM M on input $\langle C, w\rangle$.
3. If M accepts, *accept*; otherwise, *reject*."

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

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-A NFA AND w IS-A string:

1. let C = $\text{CONVERT}_{\text{NFA-DFA}}$ B
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: $\text{CONVERT}_{\text{NFA-DFA}}$ **is decider** (finite number of NFA states)
- Step 2 always halts because: M **is a decider** (prev A_{DFA} thm)
- Step 3 always halts because: N **has finite number of accept states**

How to Design Deciders, Part 2

Hint:

- **Previous theorems** 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:**
  - $\text{CONVERT}_{\text{NFA-DFA}}$, **RegExpr→NFA**
  - Union operation, intersect, star, UP, REV, BOS
  - 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

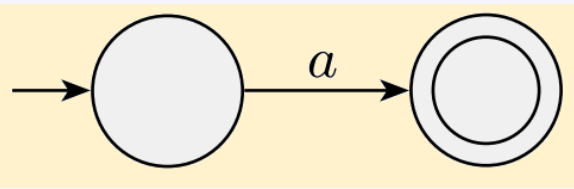
Remember:
TMs ~ programs
Creating TM ~ programming
Previous theorems ~ library

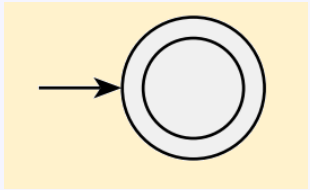
RegExpr→NFA (sketch)

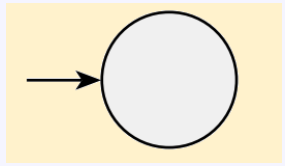
(this function is recursive!)

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

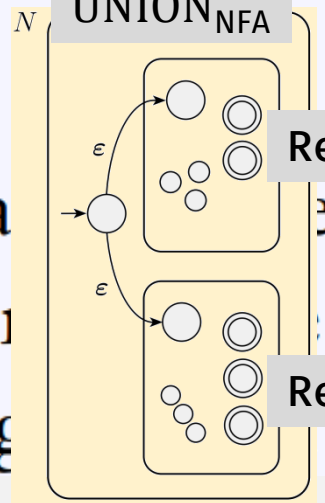
1. a for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 a… regular e…
5. $(R_1 \circ R_2)$, where R_1 a… regular ex…
6. (R_1^*), where R_1 is a reg… ession.







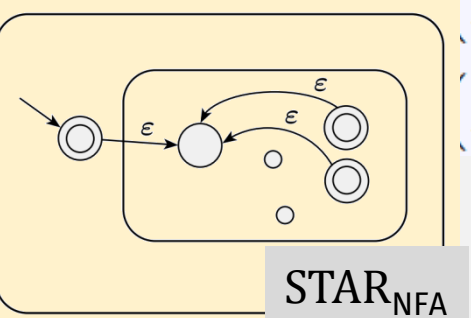
UNION$_{NFA}$



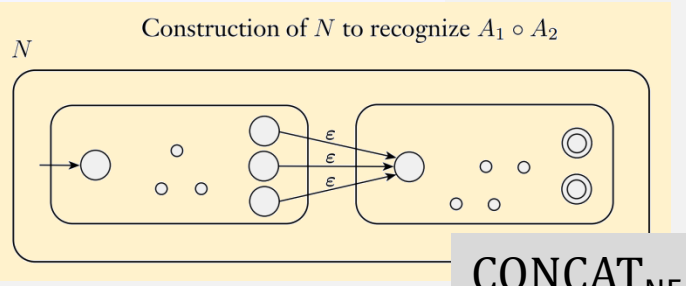
RegExpr→NFA(R_1

RegExpr→NFA(R_2)

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



STAR$_{NFA}$



CONCAT$_{NFA}$

RegExpr→NFA (sketch)

(this function is recursive!)

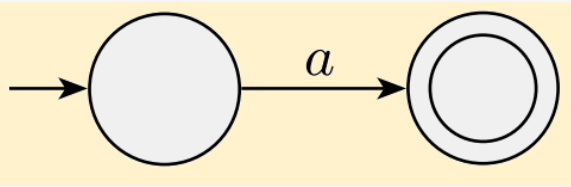
… so guaranteed to always reach base case(s)

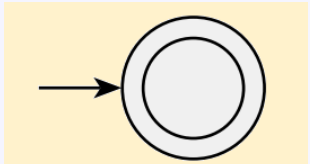
Does this conversion always halt, and why?

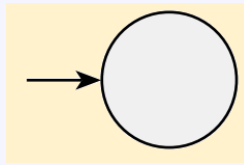
Yes, because recursive call only happens on "smaller" regular expressions …

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

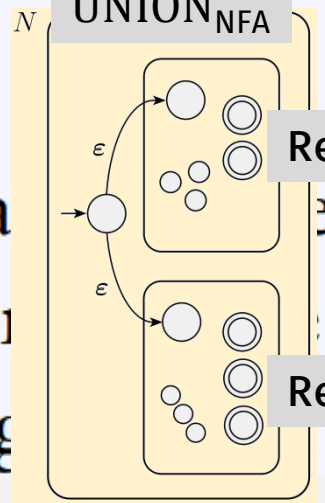
1. a for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 a… regular e…
5. $(R_1 \circ R_2)$, where R_1 a… regular ex…
6. (R_1^*), where R_1 is a reg… ession.







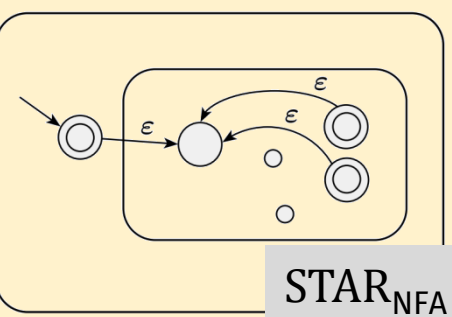
UNION$_{NFA}$



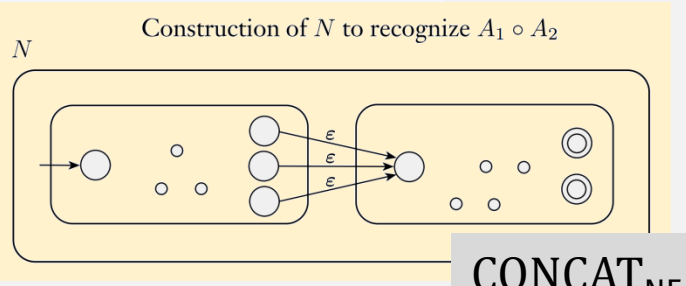
RegExpr→NFA(R_1)

RegExpr→NFA(R_2)

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



STAR$_{NFA}$



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. let A = **RegExpr→NFA** R
2. Run TM N on input $\langle A, w\rangle$.
3. If N accepts, *accept*; if N rejects, *reject*."

(from prev slide)

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**
- Step 3 always halts because: **N has finite number of accept states**

Decidable Languages About DFAs

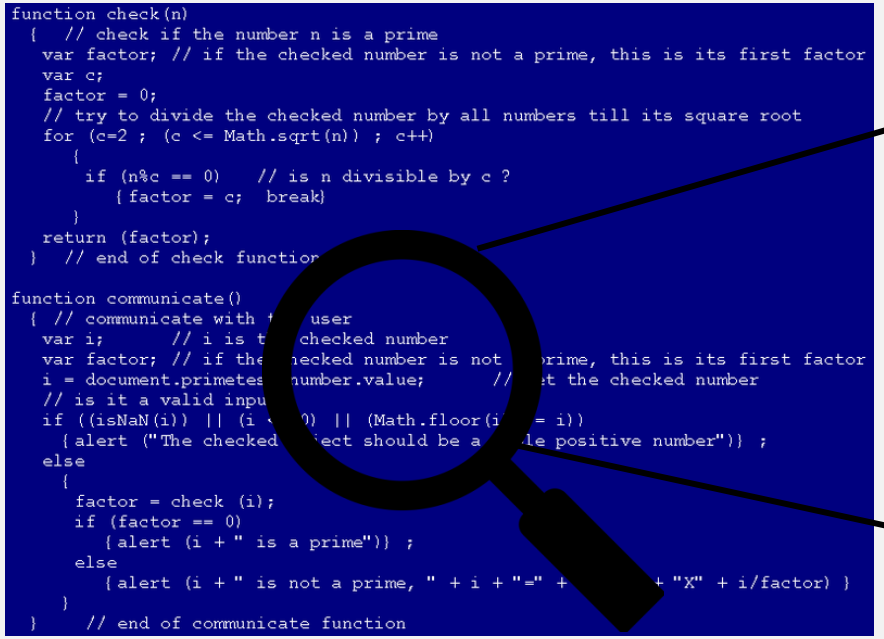
- $A_{\text{DFA}} = \{\langle B, w\rangle |\ B \text{ is a DFA that accepts input string } w\}$
  - Decider TM: implements B DFA's extended δ fn
- $A_{\text{NFA}} = \{\langle B, w\rangle |\ B \text{ is an NFA that accepts input string } w\}$
  - Decider TM: uses $\text{CONVERT}_{\text{NFA-DFA}}$ algorithm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

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

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

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

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

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

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

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** <u>two DFAs are "equivalent"</u>?

<u>A Naïve Attempt</u> (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

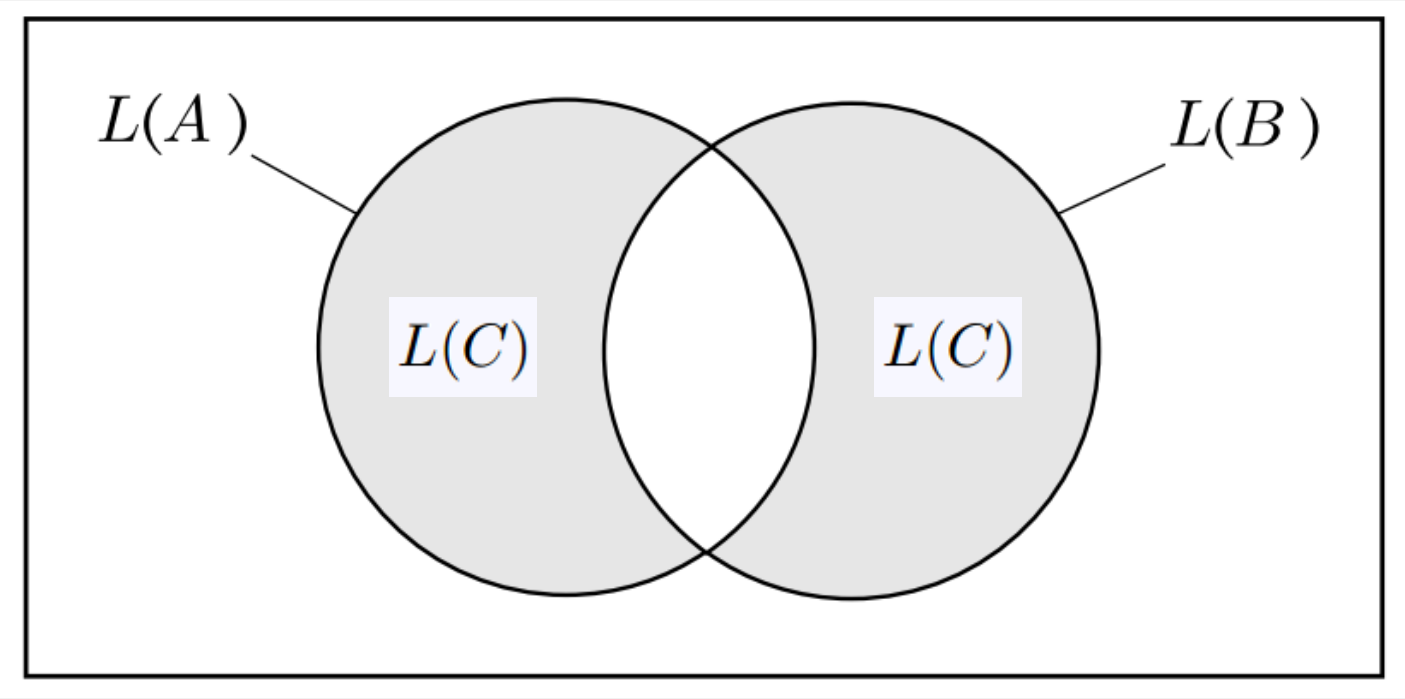
Can we **compute this <u>without</u> running the DFAs**, i.e., by only **examining the DFA's "source code"?**

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

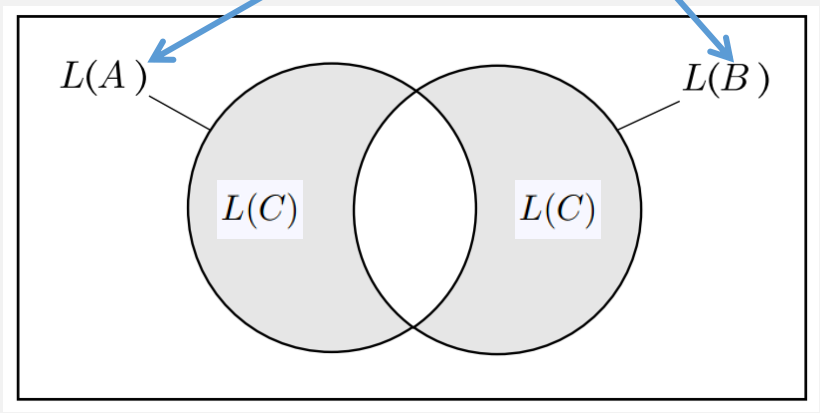
$$EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$$

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

F = "On input $\langle A, B\rangle$, where A and B IS-A 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

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