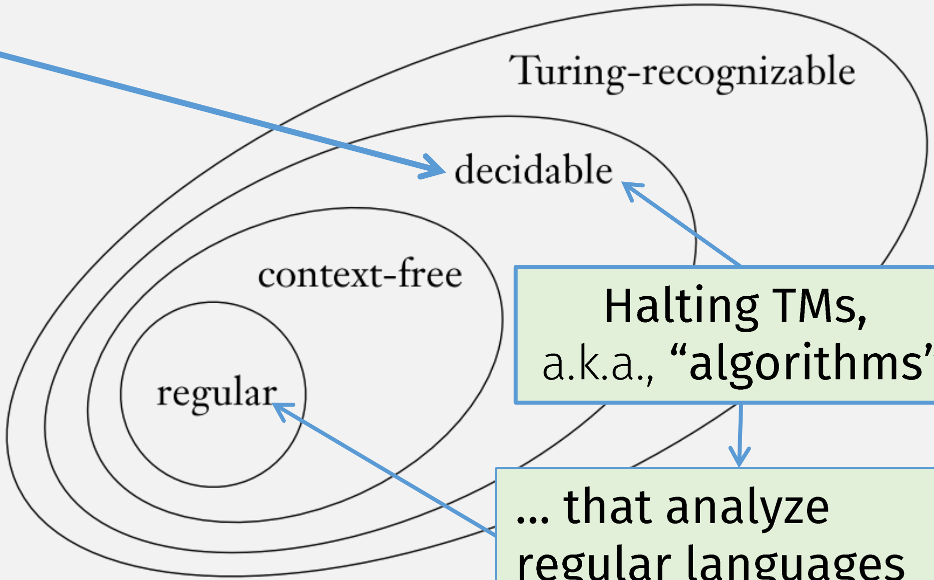
Last Time

Decidability for Regular Langs



Halting TMs,
a.k.a., “algorithms”

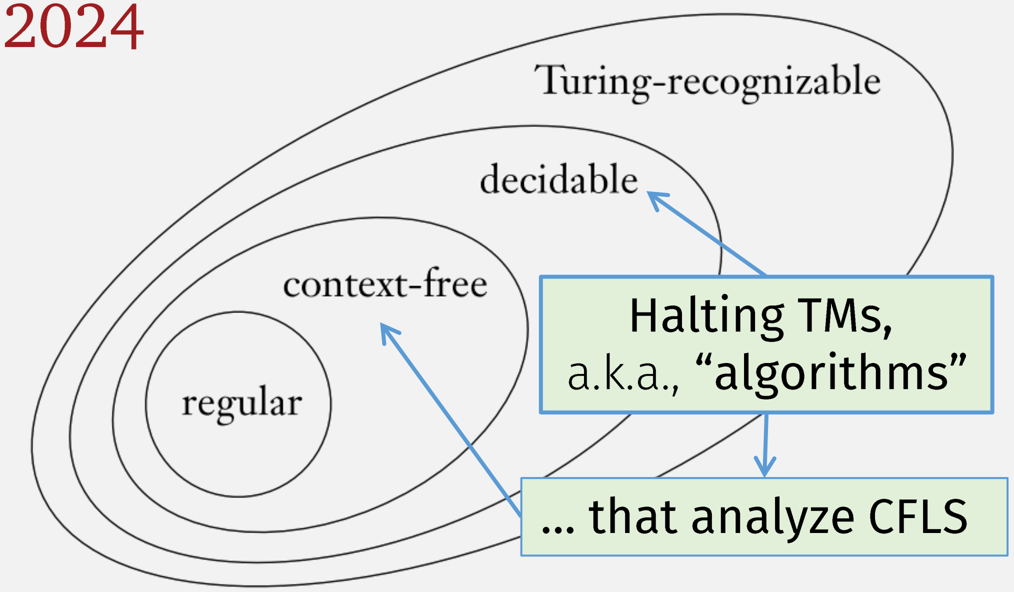
... that analyze
regular languages

Today

UMB CS 622

Decidability for CFLs

Wednesday, April 10, 2024



Announcements

- HW 7 in
  - ~~Due Wed April 10 12pm noon~~

- HW 8 out
  - Due Wed April 17 12pm noon

- No class next Monday 4/15!

Lecture Participation Question 4/10 (on gradescope)

- Which of the following rules are valid for a grammar in **Chomsky Normal Form**?

Last Time

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

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

Decider for A_{DFA} :

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

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

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*."

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*."

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)

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: <u>Step #2</u> always halts because:
determining **accept** requires checking <u>finite</u> number of accept states

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*."

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

"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

(especially important when machine could loop)

Columns must match!

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

This is what a "Equivalence Table" justification should look like!

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} :

Decider input <u>must match</u> (encodings of) strings in the language!

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

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

Flashback: NFA→DFA

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

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

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

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

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

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

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

This conversion is computation!

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

Turning Machine NFA→DFA

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

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

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

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

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

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

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

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

$A_{\text{NFA}} = \{\langle B, w\rangle \mid B$ 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: **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** are a **"library" of reusable TMs**
- **When creating a TM, use this "library" to help you!**
  - Just like libraries are useful when programming!
- E.g., **"Library" for DFAs:**
  - **NFA→DFA, RegExpr→NFA**
  - **Union 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 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 `RegExpr2NFA`

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

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

Flashback: RegExpr2NFA (hw4 problem 2)

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

$\text{RegExpr2NFA}(a) = N_a = (\{q_0, q_1\}, \Sigma, \delta, q_0, \{q_1\})$ where $\delta(q_0, a) = \{q1\}$ and $\delta(q, a) = \emptyset$ for all other q and a

2. ε,

3. $\emptyset$,

4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,

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

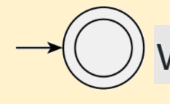
6. (R_1^*), where R_1 is a regular expression.

Flashback: RegExpr2NFA (hw4 problem 2)

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

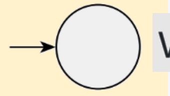
$\text{RegExpr2NFA}(a) = N_a = (\{q_0, q_1\}, \Sigma, \delta, q_0, \{q_1\})$ where $\delta(q_0, a) = \{q1\}$ and $\delta(q, a) = \emptyset$ for all other q and a

$\text{RegExpr2NFA}(\varepsilon) = N_\varepsilon = (\{q_0\}, \Sigma, \delta, q_0, \{q_0\})$



where $\delta(q_0, a) = \emptyset$ for all q and a

$\text{RegExpr2NFA}(\emptyset) = N_\emptyset = (\{q_0\}, \Sigma, \delta, q_0, \emptyset)$



where $\delta(q_0, a) = \emptyset$ for all q and a

4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,

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

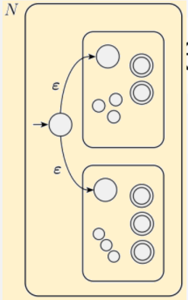
6. (R_1^*), where R_1 is a regular expression.

Flashback: RegExpr2NFA (hw4 problem 2)

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

$\text{RegExpr2NFA}(a) = N_a = (\{q_0, q_1\}, \Sigma, \delta, q_0, \{q_1\})$ where $\delta(q_0, a) = \{q1\}$ and $\delta(q, a) = \emptyset$ for all other q and a

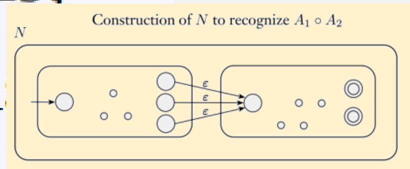
$\text{RegExpr2NFA}(\varepsilon) = N_\varepsilon = (\{q_0\}, \Sigma, \delta, q_0, \{q_0\})$ where $\delta(q_0, a) = \emptyset$ for all q and a



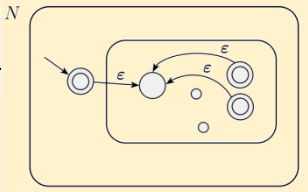
$\text{egExpr2NFA}(\emptyset) = N_\emptyset = (\{q_0\}, \Sigma, \delta, q_0, \emptyset)$ where $\delta(q_0, a) = \emptyset$ for all q and a

4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,

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



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



Flashback: RegExpr2NFA (hw4 problem 2)

... so guaranteed to always reach base case(s)

Does this conversion always halt, and why?

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

$\text{RegExpr2NFA}(a) = N_a = (\{q_0, q_1\}, \Sigma, \delta, q_0, \{q_1\})$ where $\delta(q_0, a) = \{q1\}$ and $\delta(q, a) = \emptyset$ for all other q and a

$\text{RegExpr2NFA}(\varepsilon) = N_\varepsilon = (\{q_0\}, \Sigma, \delta, q_0, \{q_0\})$ where $\delta(q_0, a) = \emptyset$ for all q and a

$\text{RegExpr2NFA}(\emptyset) = N_\emptyset = (\{q_0\}, \Sigma, \delta, q_0, \emptyset)$ where $\delta(q_0, a) = \emptyset$ for all q and a

$\text{RegExpr2NFA}(R_1 \cup R_2) = \text{UNION}_{\text{NFA}}(\text{RegExpr2NFA}(R_1), \text{RegExpr2NFA}(R_2))$

$\text{RegExpr2NFA}(R_1 \cdot R_2) = \text{CONCAT}_{\text{NFA}}(\text{RegExpr2NFA}(R_1), \text{RegExpr2NFA}(R_2))$

$\text{RegExpr2NFA}(R_1^*) = \text{STAR}_{\text{NFA}}(\text{RegExpr2NFA}(R_1))$

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

Thm: A_{REX} is a decidable language

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

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

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

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

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

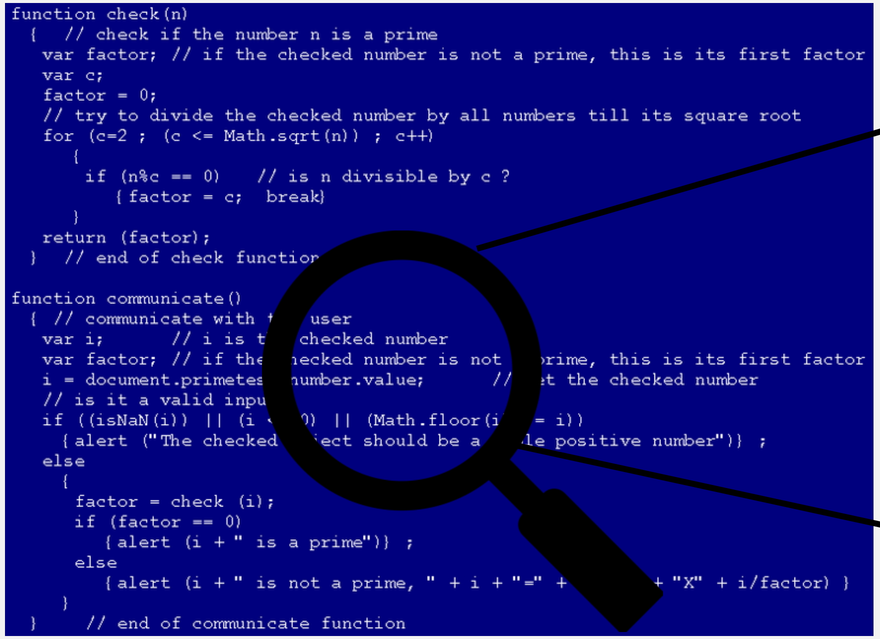
- $A_{\text{REX}} = \{\langle R, w\rangle \mid R \text{ is a regular expression that generates string } w\}$
  - Decider TM: uses **RegExpr2NFA** algorithm + A_{NFA} decider

Flashback: Why Study Algorithms About Computing

To predict what programs will do

(without running them!)

Not possible for all programs! But …



???

Predicting What <u>Some</u> Programs Will Do …

What if we: **look at <u>simpler</u> 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 question we are studying:
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

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

Decider:

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

1. Mark the start state of A.
2. $\boxed{\text{Repeat}}$ 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

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

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

~~(meta) Compute how the DFA would compute~~

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

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

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

- …

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

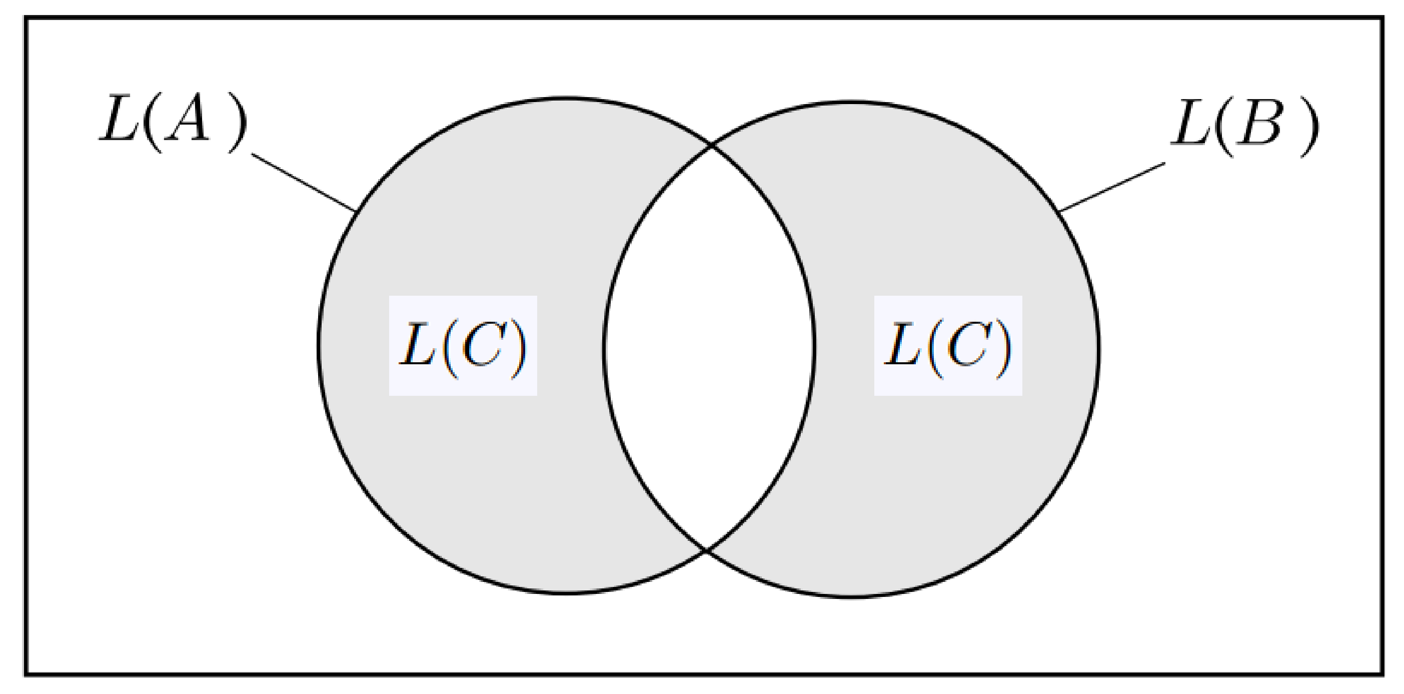
Can we compute this <u>without</u> running the DFAs?

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



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

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

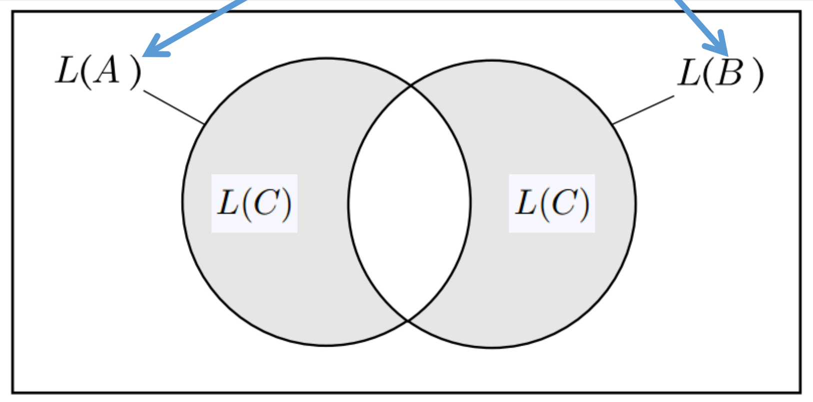
Thm: EQ_{DFA} is a decidable language

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

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: $E_{\text{DFA}} = \{\langle A\rangle |\ A \text{ is a DFA and } L(A) = \emptyset\}$
   - Because $L(C) = \emptyset$ iff $L(A) = L(B)$



Thm: EQ_{DFA} is a decidable language

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

TM input must use same string encoding as lang

Construct **decider** using 2 parts:

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

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

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

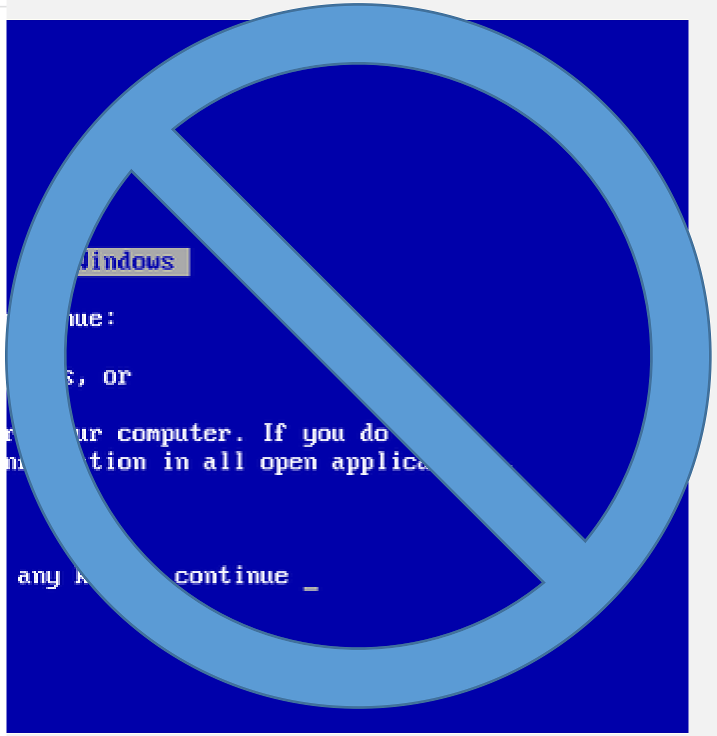
Termination argument?

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

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

Lang of the DFA

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

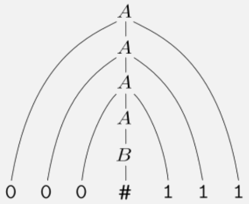
- What can we predict about CFGs or PDAs?

<u>Thm</u>: A_{CFG} is a decidable language

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

- This is a very practically important problem …
- … equivalent to:
  - **Algorithm** to <u>**parse**</u> "program" w for a programming language with grammar G?

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

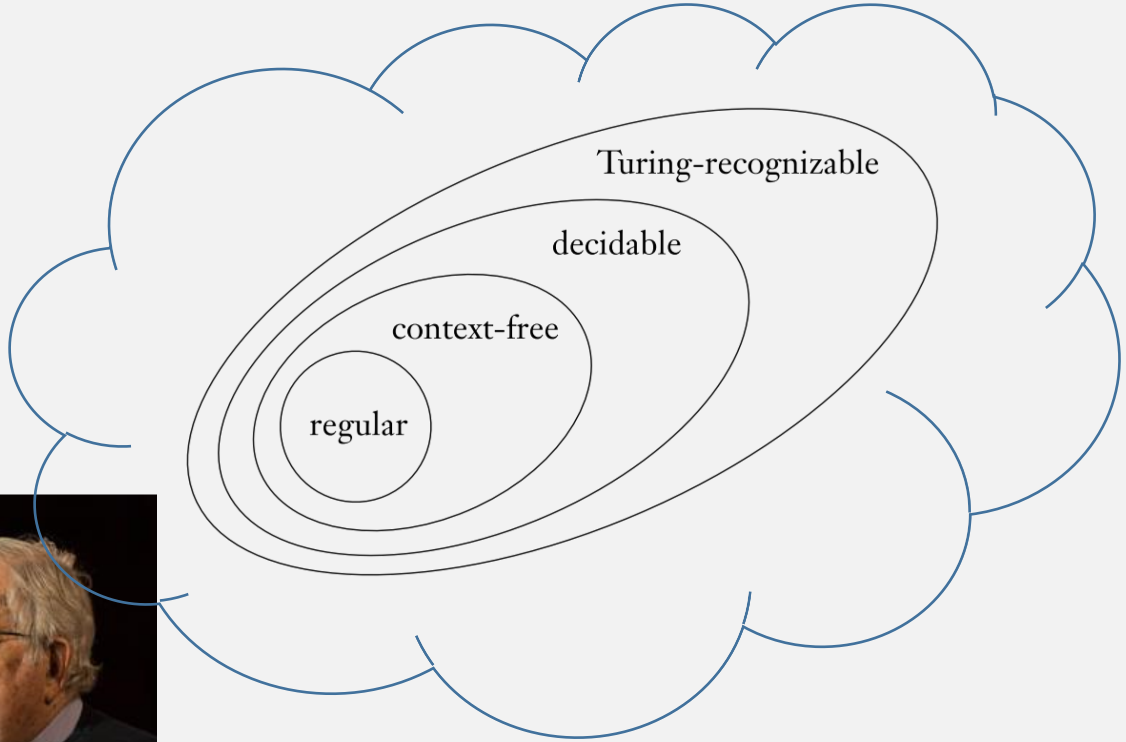


<u>Idea</u>: can the TM stop checking after some length?

- I.e., Is there upper bound on the number of derivation steps?

Chomsky Normal Form

Noam Chomsky



He came up with this <u>hierarchy</u> of languages

Chomsky Normal Form

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

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

(non-start) Variables only

2 rule shapes

Terminals only

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

Chomsky Normal Form Example

Makes the string long enough

Convert variables to terminals

- $S \rightarrow AB$
- $A \rightarrow AB$
- $A \rightarrow \mathbf{a}$
- $B \rightarrow \mathbf{b}$

- To generate string of length: 2
  - Use S rule: 1 time; Use A or B rules: 2 times
  - $S \Rightarrow AB \Rightarrow \mathbf{a}B \Rightarrow \mathbf{ab}$
  - Derivation total steps: 1 + 2 = $\boxed{3}$
- To generate string of length: 3
  - Use S rule: 1 time; A rule: 1 time; A or B rules: 3 times
  - $S \Rightarrow AB \Rightarrow AAB \Rightarrow \mathbf{a}AB \Rightarrow \mathbf{aa}B \Rightarrow \mathbf{aab}$
  - Derivation total steps: 1 + 1 + 3 = $\boxed{5}$
- To generate string of length: 4
  - Use S rule: 1 time ; A rule: 2 times; A or B rules: 4 times
  - $S \Rightarrow AB \Rightarrow AAB \Rightarrow AAAB \Rightarrow \mathbf{a}AAB \Rightarrow \mathbf{aa}AB \Rightarrow \mathbf{aaa}B \Rightarrow \mathbf{aaab}$
  - Derivation total steps: 3 + 4 = $\boxed{7}$
- ...

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

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

2 rule shapes

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

Chomsky Normal Form: Number of Steps

To generate a string of length n:

$n - 1$ steps: to generate n variables — Makes the string long enough

$+\ n$ steps: to turn each variable into a terminal — Convert string to terminals

Total: $2n - 1$ steps

(A *finite* number of steps!)

Chomsky normal form

$A \rightarrow BC$ — Use n-1 times

$A \rightarrow a$ — Use n times

Thm: A_{CFG} is a decidable language

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

Proof: create the decider:

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

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

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

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

Termination argument?

Thm: Every CFG has a Chomsky Normal Form

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

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

Chomsky normal form

$A \rightarrow BC$

$A \rightarrow a$

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

$$\Rightarrow$$

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

Thm: Every CFG has a Chomsky Normal Form

Chomsky normal form

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

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

$$
\begin{aligned}
S_0 &\to S \\
S &\to ASA \mid \mathtt{a}B \mid \mathbf{a} \\
A &\to B \mid S \mid \boldsymbol{\varepsilon} \\
B &\to \mathtt{b} \mid \varepsilon
\end{aligned}
$$

Then, add

First, remove

$$
\begin{aligned}
S_0 &\to S \\
S &\to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \mid \boldsymbol{S} \\
A &\to B \mid S \mid \varepsilon \\
B &\to \mathtt{b}
\end{aligned}
$$

Then add, to account for possibly empty A

Then, remove

Thm: Every CFG has a Chomsky Normal Form

Chomsky normal form

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

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

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

Remove, no add (same variable)

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

Remove, then add S RHSs to S_0

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

Remove, then add S RHSs to A

Termination argument of this algorithm?

Thm: Every CFG has a Chomsky Normal Form

Chomsky normal form

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

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

$$\begin{aligned}
S_0 &\to \boxed{ASA} \mid \boxed{\mathtt{a}B} \mid \mathtt{a} \mid SA \mid AS \\
S &\to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \\
A &\to \mathtt{b} \mid ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \\
B &\to \mathtt{b}
\end{aligned}$$

$$\Downarrow$$

$$\begin{aligned}
S_0 &\to \boxed{AA_1} \mid \boxed{UB} \mid \mathtt{a} \mid SA \mid AS \\
S &\to AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\
A &\to \mathtt{b} \mid AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\
A_1 &\to \boxed{SA} \\
\boxed{U \to \mathtt{a}} \\
B &\to \mathtt{b}
\end{aligned}$$

Thm: A_{CFG} is a decidable language

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

Proof: create the decider:

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

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

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

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

Thm: E_{CFG} is a decidable language

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

Recall:

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

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

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

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

Can we compute "reachability" for a CFG?

Thm: E_{CFG} is a decidable language

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

Proof: create **decider** that calculates reachability for grammar G

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

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

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

1. Mark all terminal symbols in G.
2. Repeat until no new variables get marked:
3. Mark any variable A where G has a rule $A \to U_1U_2 \cdots U_k$ and each symbol $U_1, \ldots, U_k$ has already been marked.
4. If the start variable is not marked, *accept*; otherwise, *reject*."

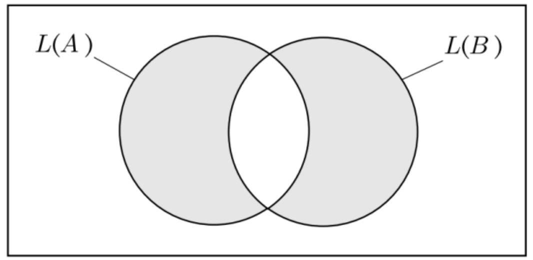
Termination argument?

Thm: EQ_{CFG} is a decidable language?

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

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

- Used Symmetric Difference



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

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

Intersection of CFLs is Not Closed!

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

- Then **intersection of these CFLs should be a CFL:**

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

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

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

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

Complement of a CFL is not Closed!

- Assume CFLs closed under complement, then:

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

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

$$\overline{L(G_1)} \cup \overline{L(G_2)} \text{ context-free}$$ Union of CFLs is closed

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

$$L(G_1) \cap L(G_2) \text{ context-free}$$ DeMorgan's Law!

But intersection is not closed for CFLS (prev slide)

Thm: EQ_{CFG} is a decidable language?

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

- No! 
  - There's no algorithm to decide whether two grammars are equivalent!

- **It's not recognizable either!** (Can't create any TM to do this!!!)
  - (details later)

- I.e., **this is an impossible computation!**

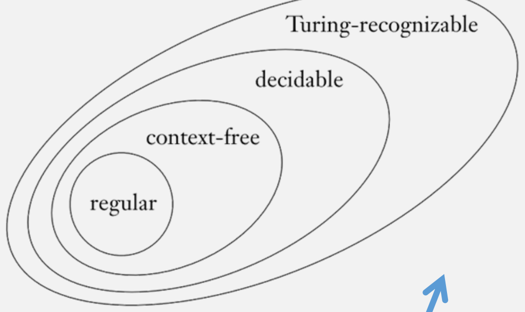
Summary Algorithms About CFLs

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

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

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

The Limits of Turing Machines?

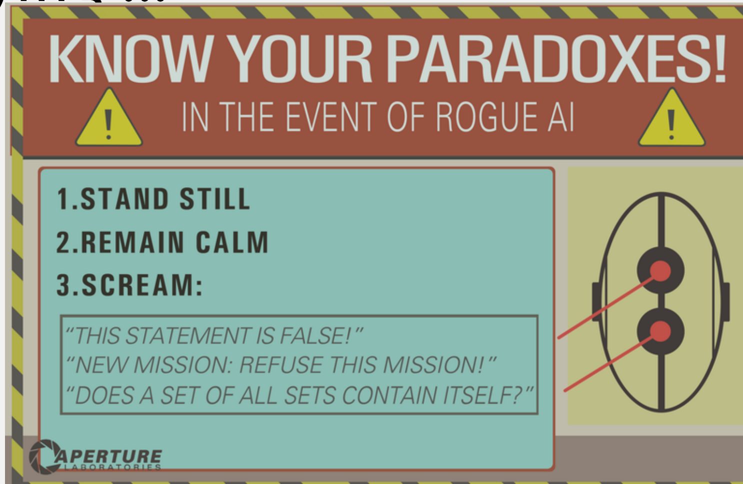


- **TMs represent all possible "computations"**
  - I.e., **any (**`Python, Java, …`**) program you write is a TM**

- But **some things are not computable?** I.e., some langs are out here **?**

- **To explore the limits of computation, we have been studying …
  … computation about other computation …**
  - <u>Thought</u>: **Is there a decider (algorithm) to determine whether a TM is an decider?**

Hmmm, this doesn't feel right …



Next time: Is A_{TM} decidable?

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