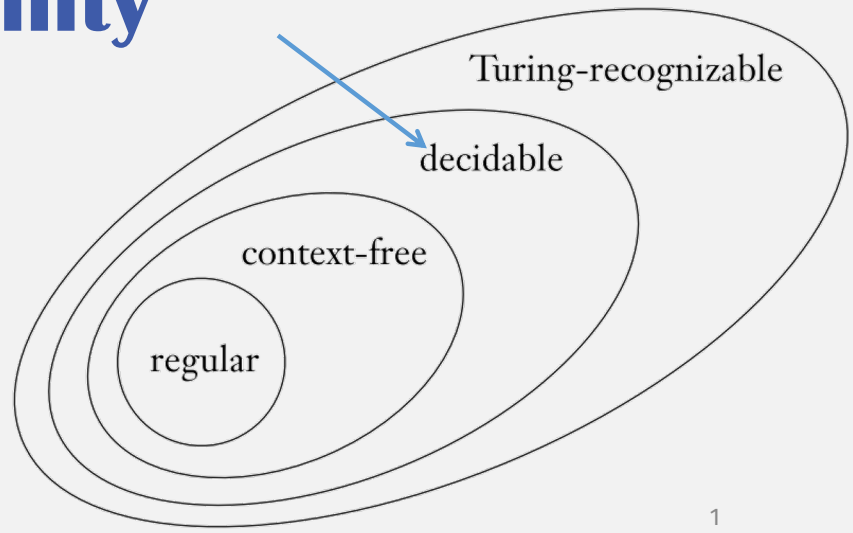
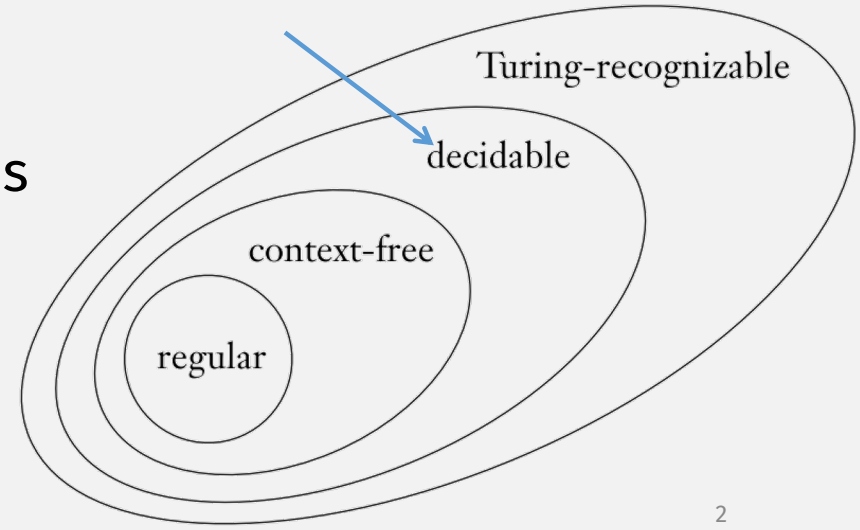
Decidability



Announcements

- HW 7
  - Covers PDAs, PDA<->CFG material
  - “Iterative development”
- HW 8 & 9 A (simpler) variant of TMs



Turing Machines and Algorithms

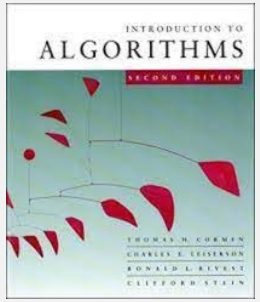
- Turing Machines can express any "computation"
  - I.e., **a Turing Machine is just a** (Python, Java, Racket, …) **program**!

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

Remember:
TMs = programs

- Algorithms are an important class of programs
  - In this class, an algorithm is any program that always halts

- So deciders model algorithms!



Algorithms (i.e., Decidable Problems) about Regular Languages

The “run” algorithm as a Turing Machine

- HW2’s “run” function is a Turing Machine.
  - Remember: (Python) **programs = Turing Machines**

- What is the language recognized by this Turing Machine?
  - I.e., what are the inputs?

The language of the “run” function

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

Interlude: Encoding Things into Strings

- A Turing machine's input is always a string
- So anything we want to give to TM must be **encoded** as string
- Notation: <Something> = encoding for Something, as a string
  - E.g., Something might be a DFA
  - Can you think of a string "encoding" for DFAs????
    - Used in HW1, HW2, …
- Use a tuple to combine multiple encodings, e.g., <B,w> (from prev slide)

Interlude: Informal TMs and Encodings

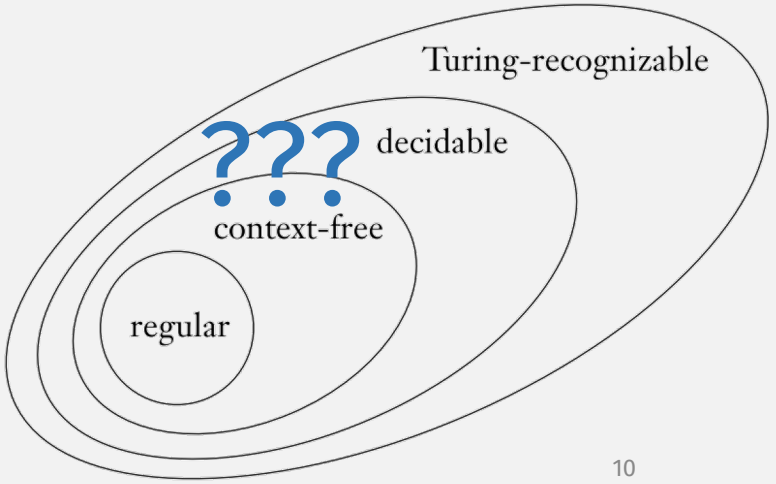
- An informal TM description:
  - Doesn't need to describe exactly how input string is encoded
  - Assumes input is a "valid" encoding
    - Invalid encodings are automatically rejected

The language of the “run” function

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

- “run” program is a Turing machine
- But is it a decider or recognizer?
  - I.e., is it an algorithm?
- To show it’s an algo, need to prove:

A_{DFA} is a decidable language



How to prove that a language is decidable?

- Create a Turing machine that decides that language!

Remember:

- A decider is Turing Machine that always halts, and, for any input, either accepts or rejects it.

How to Design Deciders

- If TMs = Programs …
- … then **Creating** a TM = Programm**ing**

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

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" =

- Start in the starting state "q0" ...
- For each input char x ...
  - Call delta fn with current state and x to compute "next state"

Remember:
TMs = programs
Creating TM = programming

- This is a decider (i.e., it always halts) because the input is always finite
- **This is just the answer to HW2's "run" function!**
  - I.e., you already "proved" this!

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 an NFA and w is a string:

1. Convert NFA B to an equivalent DFA C, using the procedure for this conversion given in Theorem 1.39.
2. Run TM M on input $\langle C, w\rangle$. (from prev slide)
3. If M accepts, *accept*; otherwise, *reject*."

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

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

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

How to Design Deciders, Part 2

- If TMs = Programs …
- … then **Creating** a TM = Programm**ing**

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

Hint:

- Previous (constructive) theorems are a "library" of reusable TMs
- When creating a TM, try to use these theorems to help you
  - Just like you use libraries when programming!
- E.g., "Library" for DFAs:
  - NFA->DFA, Regexp->NFA,
  - union, intersect, star, homomorphism, FLIP,
  - 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:

$P =$ "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 for this conversion given in Theorem 1.54.
2. Run TM N on input $\langle A, w\rangle$.
3. If N accepts, *accept*; if N rejects, *reject*."

This is a decider because:

- Step 1 always halts because converting 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

DFA TMs Recap (So Far)

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

- $A_{\text{DFA}} = \{\langle B, w\rangle | \ B \text{ is a DFA that accepts input string } w\}$
  - Deciding TM = program = HW2 “run” function
- $A_{\text{NFA}} = \{\langle B, w\rangle | \ B \text{ is an NFA that accepts input string } w\}$
  - Deciding TM = program = HW3 NFA->DFA + DFA “run”
- $A_{\text{REX}} = \{\langle R, w\rangle | \ R \text{ is a regular expression that generates string } w\}$
  - Deciding TM = program = In class Regexp->NFA + NFA->DFA + DFA “run”

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.

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

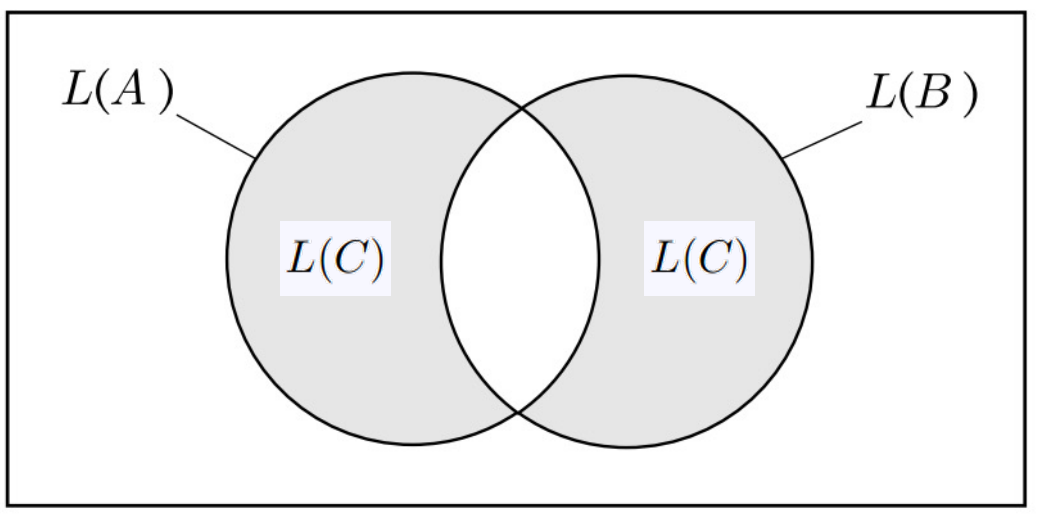
we check if accept states are "reachable" from start state

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

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

Construct decider using 2 ingredients:

- 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
- decider (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*."

Summary: Decidable DFA Langs (i.e., algorithms)

- $A_{\text{DFA}} = \{\langle B, w\rangle |\ B \text{ is a DFA that accepts input string } w\}$
- $A_{\text{NFA}} = \{\langle B, w\rangle |\ B \text{ is an NFA that accepts input string } w\}$
- $A_{\text{REX}} = \{\langle R, w\rangle |\ R \text{ is a regular expression that generates string } w\}$
- $E_{\text{DFA}} = \{\langle A\rangle |\ A \text{ is a DFA and } L(A) = \emptyset\}$
- $EQ_{\text{DFA}} = \{\langle A, B\rangle |\ A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$

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

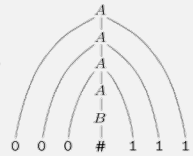
Algorithms (i.e., Decidable Problems) about CF Languages

Thm: A_{CFG} is a decidable language

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

- This a is very practically important problem …
- … equivalent to:
  - Is there an **algorithm to parse a programming language** with grammar G?

- A Decider for this problem could … ?
  - Try every possible derivation of G, and check if it's equal to w?
  - But this might never halt
    - e.g., if there is a rule like: S -> 0S or S -> S
  - This TM would be a recognizer but not a decider



- Idea: can the TM stop checking after some length?
  - i.e., Is there upper bound on the number of derivation steps?

Chomsky Normal Form

DEFINITION 2.8

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

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

Variables only

2 kinds of rules

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: Number of Steps

- To generate a string of length n:
  - $n - 1$ steps: to generate n variables
  - + n steps: to turn each variable into a terminal
  - Total: $2n - 1$ steps

Chomsky normal form

$A \to BC$

$A \to a$

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

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

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

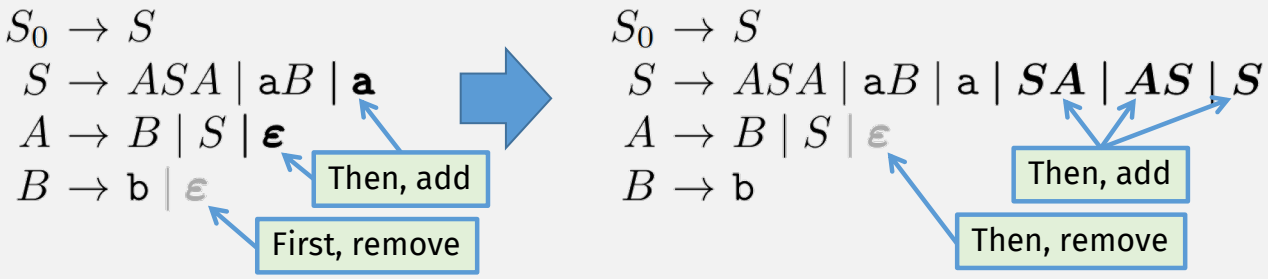
Thm: Every CFG has a Chomsky Normal Form

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: $R \to uvAw$, $R \to uAvw$, $R \to uvw$



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 $A \to u$

$S_0 \to S$
$S \to ASA \mid aB \mid a \mid SA \mid AS \mid S$
$A \to B \mid S$
$B \to b$

Remove, no add (same variable)

$S_0 \to S \mid \boldsymbol{ASA} \mid \boldsymbol{aB} \mid \boldsymbol{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS}$
$S \to ASA \mid aB \mid a \mid SA \mid AS$
$A \to B \mid S$
$B \to b$

Remove, then add S RHSs to S_0

$S_0 \to ASA \mid aB \mid a \mid SA \mid AS$
$S \to ASA \mid aB \mid a \mid SA \mid AS$
$A \to S \mid b \mid \boldsymbol{ASA} \mid \boldsymbol{aB} \mid \boldsymbol{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS}$
$B \to b$

Remove, then add S RHSs to A

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 $W \to w$, $X \to x$, $Y \to y$, $Z \to z$

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

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

To the JFLAP!!

Caveats

- JFLAP reserves ‘(’ and ‘)’ for their own use in this conversion
- CNF conversion forbids start ⟶ ε
- JFLAP gives you 26 variables in your grammar – so e.g. don’t use `types`

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 $2n - 1$ 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*."

Thm: E_{CFG} is a decidable language

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

Recall:

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

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

Thm: E_{CFG} is a decidable language

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

- Create decider that calculates reachability for grammar G
  - Except go backwards, start from terminals, to avoid looping

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

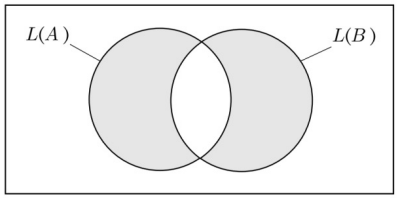
1. Mark all terminal symbols in G.
2. Repeat until no new variables get marked:
3. Mark any variable A where G has a rule $A \to U_1 U_2 \cdots U_k$ and each symbol $U_1, \ldots, U_k$ has already been marked.
4. If the start variable is not marked, *accept*; otherwise, *reject*."

Thm: EQ_{CFG} is a decidable language?

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

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

- 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!

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

- Not a CFL!
  - See textbook example 2.36

Complement of a CFL is not Closed!

- If CFLs closed under complement:

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

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

$$\overline{L(G_1)} \cup \overline{L(G_1)} \text{ context-free}$$

$$\overline{\overline{L(G_1)} \cup \overline{L(G_1)}} \text{ context-free}$$

$$L(G_1) \cap L(G_2) \text{ context-free}$$

DeMorgan's Law!

Thm: EQ_{CFG} is a decidable language?

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

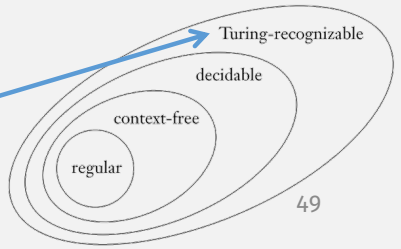
- CFLs aren't closed under intersection …
- No!
  - You can't decide whether two CF grammars represent the same lang!
- It's not recognizable either!
  - (But we won't learn how to prove this until Chapter 5)

Decidability of CFGs Recap

- $A_{\text{CFG}} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates string } w\}$
  - Convert grammar to Chomsky Normal Form
  - Then check all possible derivations of length 2|w| - 1 steps
- $E_{\text{CFG}} = \{\langle G\rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}$
  - Compute “reachability” of start variable from terminals
- $EQ_{\text{CFG}} = \{\langle G, H\rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$
  - We couldn’t prove that this is decidable!
  - (So you cant use this theorem when creating another decider)

The Limits of Turing Machines?

- So TMs can express any "computation"
  - I.e., any (Python, Java, Racket, …) program you write is a Turing Machine
- So why do we focus on TMs that process other machines?
- Because in 3800, we also want to study the limits of computation
  - And a good way to test the limit of a computational model is to see what it can compute about other computational models …
- So what are the limits of TMs? I.e., what's here?
  - Or out here?



Next time: A_{TM} is undecidable ???

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

A_{TM} = the problem of computers simulating other computers, e.g.:

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

1. Simulate M on input w.
2. If M ever enters its accept state, *accept*; if M ever enters its reject state, *reject*."

I.e., will machines take over the world?

Check-in Quiz

On gradescope