Decidability for Regular Langs

Last Time

Halting TMs, a.k.a., “algorithms”

... that analyze regular languages
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?
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., \( M = \text{On input } <B, w>, \) where \( B \) is a DFA and \( w \) is a string: ...

• A **Decider** is a TM ... that must always **halt**
  • Can only: **accept** or **reject**
  • Cannot: go into an infinite loop

• So a **Decider** definition must include: an **extra termination argument**:
  • Explains how every step in the TM halts
  • (Pay special attention to loops)

• Remember our analogy: **TMs ~ Programs** ... so **Creating a TM ~ Programming**
  • To design a TM, think of how to write a program (function) that does what you want
**Thm:** $A_{DFA}$ is a decidable language

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

**Decider for $A_{DFA}$:**

For input $(B, w)$, 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 $q_{\text{current}} = \text{start state } q_0$
- For each input char $x$ in $w$ ...
  - Define $q_{\text{next}} = \delta(q_{\text{current}}, x)$
  - Set $q_{\text{current}} = q_{\text{next}}$

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

“Calling” the DFA (with an input argument)

Remember:

| TM ~ program |
| Creating TM ~ programming |
**Thm:** $A_{DFA}$ is a decidable language

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

**Decider for $A_{DFA}$:**

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

**NOTE:** A TM must declare “function” parameters and types ... (don’t forget it)

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

... which can be used (properly!) in the TM description
**Thm:** $A_{DFA}$ is a decidable language

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

**Decider for $A_{DFA}$:**

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

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 $q_{current} = \text{start state } q_0$
- For each input char $x$ in $w$...
  - Define $q_{next} = \delta(q_{current}, x)$
  - Set $q_{current} = q_{next}$

**Termination Argument:** Step #1 always halts because: the simulation input is always finite, so the loop has finite iterations and always halts

**Deciders must have a termination argument:**

Explains how every step in the TM halts (we typically only care about loops)
**Thm:** $A_{DFA}$ is a decidable language

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

**Decider for $A_{DFA}$:**

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

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)
**Thm:** $A_{DFA}$ is a decidable language

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

**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) "called" machine column(s)

(Actual) behavior

(Expected) behavior

<table>
<thead>
<tr>
<th>Example Str</th>
<th>$B$ on input $w$?</th>
<th>$M$?</th>
<th>In $A_{DFA}$ lang?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle D_1, w_1 \rangle$</td>
<td>Accept</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>$\langle D_2, w_2 \rangle$</td>
<td>Reject</td>
<td>Reject</td>
<td>No</td>
</tr>
</tbody>
</table>

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!

(new especially important when machine could loop)
**Thm:** $A_{\text{NFA}}$ is a decidable language

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

**Decider for $A_{\text{NFA}}$:**

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

\[N = \text{“On input } \langle B, w \rangle, \text{ where } B \text{ is an NFA and } w \text{ is a string:} \]

1. Convert NFA $B$ to an equivalent DFA $C$, using the procedure for $\text{NFA} \rightarrow \text{DFA}$ ???
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. \( F' = \{R \in Q' | R \text{ contains an accept state of } N\} \)
Flashback: NFA $\rightarrow$ DFA

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

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

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

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

$q_0' = \{q_0\}$

$F' = \{R \in Q' | R \text{ contains an accepting state}\}$

New TM Variation!
Doesn’t accept or reject, Just writes “output” to tape

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

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

**Decider for $A_{NFA}$:**

1. Convert NFA $B$ to an equivalent DFA $C$, using the procedure $\text{NFA} \rightarrow \text{DFA}$.
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.

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)
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 $\rightarrow$ DFA, RegExpr $\rightarrow$ NFA
  - Union operation, intersect, star, decode, reverse
  - Deciders for: $A_{DFA}, A_{NFA}, A_{REX}, ...$
Thm: $A_{\text{REX}}$ 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 $\text{RegExpr2NFA}$

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

NOTE: A TM must declare “function” parameters and types ... (don’t forget it)

Remember:

TM$s \sim$ programs
Creating TM $\sim$ programming
Previous theorems $\sim$ library
Flashback: **RegExpr2NFA** (hw4 problem 2)

A regular expression is a string that describes a set whose members share some common characteristics. The function `RegExpr2NFA(a)` can be defined as follows:

\[ \text{RegExpr2NFA}(a) = \mathcal{N}_a = \left( \{q_0, q_1\}, \Sigma, \delta, q_0, \{q_1\} \right) \]

where

- \( \delta(q_0, a) = \{q_1\} \)
- \( \delta(q, a) = \emptyset \) for all other \( q \) and \( a \)

The function `RegExpr2NFA` can be evaluated for different regular expressions as follows:

1. \( \varepsilon \),
2. \( \emptyset \),
3. \( (R_1 \cup R_2) \), where \( R_1 \) and \( R_2 \) are regular expressions,
4. \( (R_1 \circ R_2) \), where \( R_1 \) and \( R_2 \) are regular expressions, or
5. \( (R_1^*) \), where \( R_1 \) is a regular expression.
Flashback: **RegExp2NFA** (hw4 problem 2)

- \( R \) is a **regular expression** if \( R \) is

\[
\text{RegExp2NFA}(a) = N_a = (\{ q_0, q_1 \}, \Sigma, \delta, q_0, \{ q_1 \})
\]

where \( \delta(q_0, a) = \{ q_1 \} \) and \( \delta(q, a) = \emptyset \) for all other \( q \) and \( a \)

\[
\text{RegExp2NFA}(\varepsilon) = N_\varepsilon = (\{ q_0 \}, \Sigma, \delta, q_0, \{ q_0 \})
\]

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

\[
\text{RegExp2NFA}(\emptyset) = N_\emptyset = (\{ q_0 \}, \Sigma, \delta, q_0, \emptyset)
\]

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: \textbf{RegExpr2NFA} (hw4 problem 2)

**Definition:**

A regular expression \( R \) is a **regular expression** if

\[
\text{RegExpr2NFA}(a) = N_a = (\{q_0, q_1\}, \Sigma, \delta, q_0, \{q_1\})
\]

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

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

where \( \delta(q_0, a) = \{q_1\} \)
and \( \delta(q, a) = \emptyset \) for all other \( q \) and \( a \)

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

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

\[
4. (R_1 \cup R_2), \text{ where } R_1 \text{ and } R_2 \text{ are regular expressions.}
\]

\[
5. (R_1 \circ R_2), \text{ where } R_1 \text{ and } R_2 \text{ are regular expressions.}
\]

\[
6. (R_1^*), \text{ where } R_1 \text{ is a regular expression.}
\]
Flashback: RegExpr2NFA (hw4 problem 2)

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

Does this conversion always halt, and why?

\[ R \text{ is a regular expression if } R \text{ is} \]

\[
\begin{align*}
\text{RegExpr2NFA}(a) &= N_a = (\{q_0, q_1\}, \Sigma, \delta, q_0, \{q_1\}) \quad \text{where } \delta(q_0, a) = \{q_1\} \\
\text{RegExpr2NFA}(\varepsilon) &= N_{\varepsilon} = (\{q_0\}, \Sigma, \delta, q_0, \{q_0\}) \quad \text{where } \delta(q_0, a) = \emptyset \text{ for all } q \text{ and } a \\
\text{RegExpr2NFA}(\emptyset) &= N_{\emptyset} = (\{q_0\}, \Sigma, \delta, q_0, \emptyset) \quad \text{where } \delta(q_0, a) = \emptyset \text{ for all } q \text{ and } a \\
\text{RegExpr2NFA}(R_1 \cup R_2) &= \text{UNION}_{\text{NFA}}(\text{RegExpr2NFA}(R_1), \text{RegExpr2NFA}(R_2)) \\
\text{RegExpr2NFA}(R_1 \cdot R_2) &= \text{CONCAT}_{\text{NFA}}(\text{RegExpr2NFA}(R_1), \text{RegExpr2NFA}(R_2)) \\
\text{RegExpr2NFA}(R_1^*) &= \text{STAR}_{\text{NFA}}(\text{RegExpr2NFA}(R_1))
\end{align*}
\]

Yes, because recursive call only happens on “smaller” regular expressions ...
**Thm:** $A_{\text{REX}}$ is a decidable language

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

**Decider:**

$$P = \text{"On input } \langle R, w \rangle, \text{ where } R \text{ is a regular expression and } w \text{ is a string:}\)$$

1. Convert regular expression $R$ to an equivalent NFA $A$ by using the procedure $\text{RegExpr} \to \text{NFA}$
2. Run TM $N$ on input $\langle A, w \rangle$ (from prev slide)
3. If $N$ accepts, accept; if $N$ rejects, reject.”

**Termination Argument:** This is a decider because:

- **Step 1:** always halts because: converting a reg expr to NFA is done recursively, where the reg expr gets smaller at each step, eventually reaching the base case
- **Step 2:** always halts because: $N$ is a decider
Decidable Languages About DFAs

- \( A_{DFA} = \{ \langle B, w \rangle \mid B \text{ is a DFA that accepts input string } w \} \)
  - Decider TM: implements \( B \) DFA’s extended \( \delta \) fn

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

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

RANSOMWARE ATTACK

YOUR FILES HAVE BEEN ENCRYPTED

???
Predicting What Some Programs Will Do ...

What if we: look at simpler computation models ... like DFAs and regular languages!
**Thm:** $E_{\text{DFA}}$ is a decidable language

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

$E_{\text{DFA}}$ is a language of DFA descriptions, i.e., $(Q, \Sigma, \delta, q_0, F)$ ...

... where the language of each DFA must be $\emptyset$, 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:** $E_{\text{DFA}}$ 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

- 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: $\text{EQ}_{\text{DFA}}$ is a decidable language

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

i.e., Can we compute whether two DFAs are “equivalent”?

Replacing “DFA” with “program” = A “holy grail” of computer science!
Thm: $E_{DFA}$ is a decidable language

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

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 without running the DFAs?
Thm: $EQ_{DFA}$ is a decidable language

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

Trick: Use Symmetric Difference
Symmetric Difference

\[ L(C) = (L(A) \cap L(B)) \cup (\overline{L(A)} \cap L(B)) \]

\[ L(C) = \emptyset \text{ iff } L(A) = L(B) \]
**Thm:** $\mathcal{EQ}_{\text{DFA}}$ is a decidable language

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

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 $C = \text{Union, intersection, negation of machines } A \text{ and } B$

2. **Decider $T$ (from “library”) for:** $E_{\text{DFA}} = \{ \langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset \}$
   - Because $L(C) = \emptyset \iff L(A) = L(B)$

**NOTE:** This only works because: regular langs closed under **negation**, i.e., set complement, **union** and **intersection**
**Thm:** $EQ_{\text{DFA}}$ 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 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 $C = \text{Union, intersection, negation of machines } A \text{ and } B$

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 $L(A) = L(B)$

**Termination argument?**

\[ F = \text{“On input } \langle A, B \rangle, \text{ where } A \text{ and } B \text{ are DFAs:} \]
1. Construct DFA $C$ as described.
2. Run TM $T$ deciding $E_{\text{DFA}}$ on input $\langle C \rangle$.
3. If $T$ accepts, accept. If $T$ rejects, reject.”
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

Overview of Static Driver Verifier Research Platform

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
Summary: Algorithms About Regular Langs

- $A_{DFA} = \{ \langle B, w \rangle \mid B \text{ is a DFA that accepts input string } w \}$
  - **Decider:** Simulates DFA by implementing extended $\delta$ function

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

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

- $E_{DFA} = \{ \langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset \}$
  - **Decider:** Reachability algorithm

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

---

Remember:
- TMs $\sim$ programs
- Creating TM $\sim$ programming
- Previous theorems $\sim$ library
Next: Algorithms (Decider TMs) for CFLs?

- What can we predict about CFGs or PDAs?
Thm: $A_{CFG}$ is a decidable language

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

- This is a very practically important problem ...
- ... equivalent to:
  - Algorithm to parse “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 might never halt
    - E.g., what if there are rules like: $S \rightarrow \emptyset S$ or $S \rightarrow S$
    - 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
Noam Chomsky

He came up with this hierarchy of languages
A context-free grammar is in **Chomsky normal form** if every rule is of the form

\[ A \rightarrow BC \]
\[ A \rightarrow a \]

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 Example

- $S \rightarrow AB$
- $A \rightarrow AB$
- $A \rightarrow a$
- $B \rightarrow b$

- **To generate string of length: 2**
  - Use $S$ rule: 1 time; Use $A$ or $B$ rules: 2 times
  - $S \Rightarrow AB \Rightarrow aB \Rightarrow ab$
  - Derivation total steps: $1 + 2 = 3$

- **To generate string of length: 3**
  - Use $S$ rule: 1 time; $A$ rule: 1 time; $A$ or $B$ rules: 3 times
  - $S \Rightarrow AB \Rightarrow AAB \Rightarrow aAB \Rightarrow aaB \Rightarrow aab$
  - Derivation total steps: $1 + 1 + 3 = 5$

- **To generate string of length: 4**
  - Use $S$ rule: 1 time; $A$ rule: 2 times; $A$ or $B$ rules: 4 times
  - $S \Rightarrow AB \Rightarrow AAB \Rightarrow AAAB \Rightarrow aAAB \Rightarrow aaAB \Rightarrow aaaB \Rightarrow aab$
  - Derivation total steps: $3 + 4 = 7$

- ...
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

(A finite number of steps!)
**Thm:** $A_{CFG}$ is a decidable language

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

**Proof:** create the decider:

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

1. Convert } G \text{ to an equivalent grammar in Chomsky normal form.
2. List all derivations with } 2n - 1 \text{ steps, where } n \text{ is the length of } w; \text{ except if } n = 0, \text{ then instead list all derivations with one step.
3. If any of these derivations generate } w, \text{ 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

```
S → ASA | aB
A → B | S
B → b | ε

S_0 → S
S → ASA | aB
A → B | S
B → b | ε
```
Thm: Every CFG has a 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

2. Remove all “empty” rules of the form $A \rightarrow \varepsilon$
   - $A$ must not be the start variable
   - Then for every rule with $A$ on RHS, add new rule with $A$ deleted
     - E.g., If $R \rightarrow uAv$ is a rule, add $R \rightarrow uv$
     - Must cover all combinations if $A$ appears more than once in a RHS
       - E.g., if $R \rightarrow uAvAw$ is a rule, add 3 rules: $R \rightarrow uvAw$, $R \rightarrow uAvw$, $R \rightarrow uvw$
**Thm:** Every CFG has a 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
2. Remove all “empty” rules of the form $A \rightarrow \epsilon$
   - $A$ must not be the start variable
   - Then for every rule with $A$ on RHS, add new rule with $A$ deleted
     - E.g., If $R \rightarrow uAv$ is a rule, add $R \rightarrow uv$
     - Must cover all combinations if $A$ appears more than once in a RHS
       - E.g., if $R \rightarrow uAvAw$ is a rule, add 3 rules: $R \rightarrow uvAw$, $R \rightarrow uAvw$, $R \rightarrow uvw$
3. Remove all “unit” rules of the form $A \rightarrow B$
   - Then, for every rule $B \rightarrow u$, add rule $A \rightarrow u$
Termination argument of this algorithm?

**Thm:** Every CFG has a 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
2. Remove all “empty” rules of the form $A \rightarrow ε$
   - $A$ must not be the start variable
   - Then for every rule with $A$ on RHS, add new rule with $A$ deleted
     - E.g., if $R \rightarrow uAv$ is a rule, add $R \rightarrow uv$
     - Must cover all combinations if $A$ appears more than once in a RHS
     - E.g., if $R \rightarrow uAvAw$ is a rule, add 3 rules: $R \rightarrow uvAw, R \rightarrow uAvw, R \rightarrow uvw$
3. Remove all “unit” rules of the form $A \rightarrow B$
   - Then, for every rule $B \rightarrow u$, add rule $A \rightarrow u$
4. Split up rules with RHS longer than length 2
   - E.g., $A \rightarrow wxyz$ becomes $A \rightarrow wB, B \rightarrow xC, C \rightarrow yz$
5. Replace all terminals on RHS with new rule
   - E.g., for above, add $W \rightarrow w, X \rightarrow x, Y \rightarrow y, Z \rightarrow z$
Thm: \( A_{\text{CFG}} \) is a decidable language

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

Proof: create the decider:

\[
S = \text{"On input } \langle G, w \rangle \text{, where } G \text{ is a CFG and } w \text{ is a string:}
1. \text{ Convert } G \text{ to an equivalent grammar in Chomsky normal form.}
2. \text{ List all derivations with } 2n - 1 \text{ steps, where } n \text{ is the length of } w; \text{ except if } n = 0, \text{ then instead list all derivations with one step.}
3. \text{ If any of these derivations generate } w, \text{ accept; if not, reject."
}\]

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:** $E_{\text{CFG}}$ is a decidable language.

Recall:

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

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

$T =$ “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

Can we compute “reachability” for a CFG?
**Thm:** $E_{CFG}$ is a decidable language.

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

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

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

Let $R$ = “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 $A \rightarrow U_1 U_2 \cdots U_k$ and each symbol $U_1, \ldots, U_k$ has already been marked.
4. If the start variable is not marked, accept; otherwise, reject.”

**Termination argument?**
Thm: $EQ_{\text{CFG}}$ 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

  \[
  L(C) = \emptyset \iff L(A) = L(B)
  \]

- where $C = \text{complement, union, intersection of machines } A \text{ and } B$

- 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:

\[
A = \{ a^m b^n c^n \mid m, n \geq 0 \}
\]

\[
B = \{ a^n b^n c^m \mid m, n \geq 0 \}
\]

• But \( A \cap B = \{ a^n b^n c^n \mid n \geq 0 \} \)

• ... which is not a CFL! (So we have a contradiction)
Complement of a CFL is not Closed!

- **Assume** CFLs closed under complement, then:

  \[
  \text{if } G_1 \text{ and } G_2 \text{ context-free, then:}
  \]

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

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

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

  \[
  \overline{L(G_1)} \cap \overline{L(G_2)} \text{ context-free} \quad \text{DeMorgan’s Law!}
  \]

  **But intersection is not closed for CFLS (prev slide)**
**Thm:** $EQ_{\text{CFG}}$ is a decidable language?

\[ EQ_{\text{CFG}} = \{ (G, H) | G \text{ and } H \text{ are CFGs and } L(G) = L(H) \} \]

- 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

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

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

- \( EQ_{CFG} = \{ \langle G, H \rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H) \} \)
  - We couldn’t prove that this is decidable!
  - (So you can’t 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 ...
  • Thought: Is there a decider (algorithm) to determine whether a TM is an decider?

  Hmm, this doesn’t feel right...
Next time: Is $A_{TM}$ decidable?

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