Week 9 Tutorial

*Context-Free Grammars, TMs*
Announcements

• Reminder that the first round of revisions on the Week 8 Take Home Exam are due Saturday noon PDT.

• We have re-opened all of the lecture quiz and tutorial submissions and you may make up any missed assignments without penalty. You need to complete 6/10 tutorials and get at least 63/81 on the lecture quizzes to pass.

• Pset 9 will be optional. We will calculate your pset average score both with and without pset9 and take the larger of the two.
Part 1: *CFGs Warmup*
Let $\Sigma = \{y, d\}$ and let $DOGWALK = \{w \in \Sigma^* | w$ describes a series of steps where you and your dog arrive at the same point $\}$

Here are some incorrect CFGs for $DOGWALK$:

1. $S \to YSD \mid DSY \mid \varepsilon$
   $Y \to yY \mid \varepsilon$
   $D \to dD \mid \varepsilon$

2. $S \to ySd \mid dSy \mid \varepsilon$

3. $S \to ydS \mid dyS \mid \varepsilon$

4. $S \to ySd \mid dSy \mid ydS \mid dyS \mid \varepsilon$

1) Explain why none of these grammars are correct by identifying an example string in the language of the grammar but not in $DOGWALK$ or a string that’s in $DOGWALK$ that’s not in the language of the grammar.

Fill in answer on Gradescope!
Let $\Sigma = \{y, d\}$ and let $DOGWALK = \{w \in \Sigma^* | w$ describes a series of steps where you and your dog arrive at the same point $\}$

Here are some incorrect CFGs for $DOGWALK$:

1. $S \rightarrow YSD \mid DSY \mid \varepsilon$
   $Y \rightarrow yY \mid \varepsilon$
   $D \rightarrow dD \mid \varepsilon$

   This grammar generates the string $dd$, which is not in $DOGWALK$.

   Takeaway: related quantities can't be built independently. If two parts of your string have to match up, they need to be built together.

2. $S \rightarrow ySd \mid dSy \mid \varepsilon$

3. $S \rightarrow ydS \mid dyS \mid \varepsilon$

4. $S \rightarrow ySd \mid dSy \mid ydS \mid dyS \mid \varepsilon$
Let $\Sigma = \{ y, d \}$ and let $DOGWALK = \{ w \in \Sigma^* \mid w$ describes a series of steps where you and your dog arrive at the same point $\}$.

Here are some incorrect CFGs for $DOGWALK$:

1. $S \to YSD \mid DSY \mid \varepsilon$
   $Y \to yY \mid \varepsilon$
   $D \to dD \mid \varepsilon$

   This grammar can’t generate the string $yddy$, which is in $DOGWALK$.

   Takeaway: make sure you don’t unintentionally impose additional restrictions. While we need the number of $y$s and $d$s to be the same, it doesn’t matter what order they come in.

2. $S \to ySd \mid dSy \mid \varepsilon$

3. $S \to ydS \mid dyS \mid \varepsilon$

4. $S \to ySd \mid dSy \mid ydS \mid dyS \mid \varepsilon$
Let $\Sigma = \{ y, d \}$ and let $DOGWALK = \{ w \in \Sigma^* \mid w$ describes a series of steps where you and your dog arrive at the same point $\}$

Here are some incorrect CFGs for $DOGWALK$:

1. $S \rightarrow YSD \mid DSY \mid \varepsilon$
   $Y \rightarrow yY \mid \varepsilon$
   $D \rightarrow dD \mid \varepsilon$

This grammar can't generate the string $yydd$, which is in $DOGWALK$.

Takeaway: similar to the previous option, this grammar restricts the ordering of $ys$ and $ds$.

2. $S \rightarrow ySd \mid dSy \mid \varepsilon$

3. $S \rightarrow ydS \mid dyS \mid \varepsilon$

4. $S \rightarrow ySd \mid dSy \mid ydS \mid dyS \mid \varepsilon$
Let $\Sigma = \{y, d\}$ and let $DOGWALK = \{w \in \Sigma^* | w$ describes a series of steps where you and your dog arrive at the same point $\}$

Here are some **incorrect** CFGs for $DOGWALK$:

1. $S \rightarrow YSD \mid DSY \mid \varepsilon$
   $Y \rightarrow yY \mid \varepsilon$
   $D \rightarrow dD \mid \varepsilon$

   This grammar can't generate the string $yydddddyy$, which is in $DOGWALK$.

   Takeaway: don't try to patch up a CFG by adding in more productions.

   In CFG design, you're looking for a general rule that captures the language.

   For this particular example, simply listing off all permutations of $y, d,$ and $S$ isn't a great approach because you can't be sure that you've covered everything.

2. $S \rightarrow ySd \mid dSy \mid \varepsilon$

3. $S \rightarrow ydS \mid dyS \mid \varepsilon$

4. $S \rightarrow ySd \mid dSy \mid ydS \mid dyS \mid \varepsilon$
Part 2: Designing CFGs
Storing Information in Nonterminals

- **Key idea:** Different non-terminals should represent different states or different types of strings.
  - For example, different phases of the build, or different possible structures for the string.
  - Think like the same ideas from DFA/NFA design where states in your automata represent pieces of information.
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$. 

- Examples:

  - $\varepsilon \in L$
  - $abb \in L$
  - $bab \in L$
  - $aababa \in L$
  - $bbb\ldots \in L$
  - $a \notin L$
  - $a \notin L$
  - $b \notin L$
  - $ababab \notin L$
  - $aabaaaaaa \notin L$
  - $bbb\ldots \notin L$
  - $b \notin L$
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.

- Examples:

  - $\varepsilon \in L$
  - $a \not\in L$
  - $a b b \in L$
  - $b \not\in L$
  - $b a b \in L$
  - $a b a b \not\in L$
  - $a a b a b a \in L$
  - $a a b a \not a a a a a \not\in L$
  - $b b b b \in L$
  - $b b b b \not\in L$
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same \}. 

2) Create a CFG for the language above.

*Fill in answer on Gradescope!
Storing Information in Nonterminals

Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.  

One approach:

<table>
<thead>
<tr>
<th>aaa</th>
<th>bab</th>
<th>Observation 1:</th>
<th>Strings in this language are either: the first third is $a$s or the first third is $b$s.</th>
</tr>
</thead>
<tbody>
<tr>
<td>abb</td>
<td>bbb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>aaabab</td>
<td>bbabbb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>aababa</td>
<td>bbbaaaaaa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>aaaaaaaa</td>
<td>bbbbbabaa</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$. 

- One approach:

  | $aaa$ | $bab$ |
  | $abb$ | $bbb$ |
  | $aaabab$ | $bbabbb$ |
  | $aababa$ | $bbbaaaaaa$ |
  | $aaaaaa\ldots$ | $bbbbbb\ldots$ |
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* | |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.  
- One approach:

<table>
<thead>
<tr>
<th>aaa</th>
<th>bab</th>
<th>Observation 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>abb</td>
<td>bbb</td>
<td>Amongst these</td>
</tr>
<tr>
<td>aaabab bbbabb</td>
<td>strings, for every $a$ I</td>
<td></td>
</tr>
<tr>
<td>aababa bbbAAAAA</td>
<td>have in the first third,</td>
<td></td>
</tr>
<tr>
<td>aaaaaaaaa bbbbbbabaa</td>
<td>I need two other</td>
<td></td>
</tr>
<tr>
<td>aaaaaaaaaaa bbbbbbabaa</td>
<td>characters in the last</td>
<td></td>
</tr>
<tr>
<td>aaaaaaaa bbbbabaa</td>
<td>two thirds.</td>
<td></td>
</tr>
</tbody>
</table>
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.  

- One approach:

  - $aaa$
  - $abb$
  - $aaabab$

  $bab$
  $bbb$
  $bbabbb$

Observation 2:

Amongst these strings, for every $a$ I have in the first third, I need two other characters in the last two thirds.

This pattern of "for every $x$ I see here, I need a $y$ somewhere else in the string" is very common in CFGs!
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* | |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.  

- One approach:

  \begin{align*}
  &aaa & bab \\
  &abb & bbb \\
  &aaabab & bbabbb \\
  &aababa & bbbaaaaaa \\
  &aaaaaaaa & bbbbbbabaaaa \\
  \end{align*}

  \textbf{Observation 2:} Amongst these strings, for every $a$ I have in the first third, I need two other characters in the last two thirds.

  $A \rightarrow aAXX$ \hspace{1em} $\varepsilon$ \hspace{1em} $X \rightarrow a \hspace{1em} b$
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.  
- One approach:

  $\begin{align*}
  &\text{aaa} \\
  &\text{abb} \\
  &\text{aaabab} \\
  &\text{aababa} \\
  &\text{aaaaaaaaa}
  \end{align*}$

  $A \rightarrow aAXX \mid \varepsilon$

  $X \rightarrow a \mid b$

Here the nonterminal $A$ represents "a string where the first third is $a$'s" and the nonterminal $X$ represents "any character".
Storing Information in Nonterminals

• Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same }.

• One approach:

\[
\begin{align*}
\text{aaa} & \quad \text{bab} \\
\text{abb} & \quad \text{bbb} \\
\text{aaabab} & \quad \text{bbabbb} \\
\text{aababa} & \quad \text{bbbaaaaa} \\
\text{aaaaaaaaa} & \quad \text{bbbbbabaa} \\
\end{align*}
\]

$A \rightarrow aAXX \mid \varepsilon \quad X \rightarrow a \mid b$
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$. 
- One approach:

```
  aaa       bab
  abb       bbb
  aaabab    bbabbb
  aababa    bbbaaaaaa
  aaaaaaaaa  bbbbbabaaa
```

$B \rightarrow bBXX \mid \varepsilon$  \hspace{1cm} $X \rightarrow a \mid b$
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.

- Tying everything together:

  $S \rightarrow A \mid B$
  $A \rightarrow aAXX \mid \epsilon$
  $B \rightarrow bBXX \mid \epsilon$
  $X \rightarrow a \mid b$
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.

- Tying everything together:

  $S \rightarrow A \mid B$
  
  $A \rightarrow aAXX \mid \varepsilon$
  
  $B \rightarrow bBXX \mid \varepsilon$
  
  $X \rightarrow a \mid b$

Overall strings in this language either follow the pattern of $A$ or $B$. 
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{ w \in \Sigma^* | |w| \equiv_3 0 \text{ and all the characters in the first third of } w \text{ are the same} \}$. 

- Tying everything together:

$$
S \rightarrow A \mid B \\
A \rightarrow aAAXX \mid \varepsilon \\
B \rightarrow bBXX \mid \varepsilon \\
X \rightarrow a \mid b
$$

$A$ represents “strings where the first third is $a$’s”
Storing Information in Nonterminals

- Let \( \Sigma = \{a, b\} \) and let \( L = \{w \in \Sigma^* \mid |w| \equiv_3 0 \) and all the characters in the first third of \( w \) are the same \}.

- Tying everything together:

  \[
  \begin{align*}
  S & \rightarrow A \mid B \\
  A & \rightarrow aAXX \mid \varepsilon \\
  B & \rightarrow bBXX \mid \varepsilon \\
  X & \rightarrow a \mid b
  \end{align*}
  \]

  \( B \) represents “strings where the first third is \( b \)’s”
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.

- Tying everything together:

\[
\begin{align*}
S & \rightarrow A \mid B \\
A & \rightarrow aAXX \mid \varepsilon \\
B & \rightarrow bBXX \mid \varepsilon \\
X & \rightarrow a \mid b
\end{align*}
\]

$x$ represents "either an a or a b"
Part 3: Turing Machines
TMs and Programs

• Though TMs are formally defined using states, transitions, and a tape, we can describe the behavior of what TMs can do by writing pseudocode and abstract away the details of how exactly it’s operating.

• Throughout the rest of the course, we’ll switch back and forth between these two different models of TM behavior.
Some Closure Properties of $R$

Because $R$ corresponds to decidable problems, languages in $R$ are precisely the languages for which you can write a method

```cpp
bool inL(string w)
```

such that

- for any string $w \in L$, calling $\text{inL}(w)$ returns true.
- for any string $w \notin L$, calling $\text{inL}(w)$ returns false.
Some Closure Properties of $\mathcal{R}$

Because $\mathcal{R}$ corresponds to decidable problems, languages in $\mathcal{R}$ are precisely the languages for which you can write a method

$$\text{bool inL(string w)}$$

such that

- for any string $w \in L$, calling $\text{inL}(w)$ returns true.
- for any string $w \notin L$, calling $\text{inL}(w)$ returns false.

3a) Let $L_1$ and $L_2$ be decidable languages over the same alphabet $\Sigma$. Prove that $L_1 \cup L_2$ is also decidable. To do so, suppose that you have methods $\text{inL1}$ and $\text{inL2}$ matching the above conditions, then write a method $\text{inL1} \cup \text{inL2}$ with the appropriate properties. Briefly justify why your construction is correct.

*Fill in answer on Gradescope!*
Some Closure Properties of $\mathbb{R}$

Consider this method:

```cpp
bool inL1uL2(string w) {
    return inL1(w) || inL2(w);
}
```

Theorem:
If $L_1$ and $L_2$ are decidable, then $L_1 \cup L_2$ is decidable.

Proof:
Consider the above piece of code. If given a string $w \in L_1 \cup L_2$, then we know that either $w \in L_1$ or $w \in L_2$ (or both). Therefore, at least one of $\text{inL1}(w)$ and $\text{inL2}(w)$ will return true. Since $\text{inL1}$ and $\text{inL2}$ always return values, this means that the expression will always eventually evaluate the call to the method that returns true, so this method returns true. On the other hand, if it's given a string $w \not\in L_1 \cup L_2$, then we know that $w \not\in L_1$ and $w \not\in L_2$. Therefore, $\text{inL1}(w)$ and $\text{inL2}(w)$ will return false, so the overall method returns false. Overall, we've seen that this method returns true if $w \in L_1 \cup L_2$ and false otherwise, so the method is a decider for $L_1 \cup L_2$, showing that $L_1 \cup L_2$ decidable, as required. ■
Some Closure Properties of $\mathbb{R}$

Consider this method:

```cpp
bool inL1uL2(string w) {
    return inL1(w) || inL2(w);
}
```

**Theorem:** If $L_1$ and $L_2$ are decidable, then $L_1 \cup L_2$ is decidable.

**Proof:** Consider the above piece of code. If given a string $w \in L_1 \cup L_2$, then we know that either $w \in L_1$ or $w \in L_2$ (or both). Therefore, at least one of $\text{inL1}(w)$ and $\text{inL2}(w)$ will return true. Since $\text{inL1}$ and $\text{inL2}$ always return values, this means that the expression will always eventually evaluate the call to the method that returns true, so this method returns true. On the other hand, if it's given a string $w \notin L_1 \cup L_2$, then we know that $w \notin L_1$ and $w \notin L_2$. Therefore, $\text{inL1}(w)$ and $\text{inL2}(w)$ will return false, so the overall method returns false. Overall, we've seen that this method returns true if $w \in L_1 \cup L_2$ and false otherwise, so the method is a decider for $L_1 \cup L_2$, showing that $L_1 \cup L_2$ decidable, as required. ■
Some Closure Properties of \textbf{RE}

Because \textbf{RE} corresponds to recognizable problems, languages in \textbf{RE} are precisely the languages for which you can write a method

\begin{verbatim}
bool inL(string w)
\end{verbatim}

such that

- for any string $w \in L$, calling $\text{inL}(w)$ returns true.
- for any string $w \notin L$, calling $\text{inL}(w)$ does not return true (it could return false or enter an infinite loop).
Some Closure Properties of \textbf{RE}

Let $L_1$ and $L_2$ be recognizable languages over the same alphabet $\Sigma$. We want to prove that $L_1 \cup L_2$ is also recognizable. To do so, suppose that you have methods $\text{inL1}$ and $\text{inL2}$ matching the conditions from the previous slide. Imagine we wrote the same pseudocode:

```c
bool inL1uL2(string w) {
    return inL1(w) || inL2(w);
}
```

3b) While it’s true that $L_1 \cup L_2$ is also recognizable, our previous strategy won’t work. Identify the issue with this purported recognizer for $L_1 \cup L_2$ and try to come up with a solution.

\textit{Fill in answer on Gradescope!}
Some Closure Properties of RE

Imagine we wrote the same pseudocode:

```java
bool inL1uL2(string w) {
    return inL1(w) || inL2(w);
}
```

⚠️⚠️

Our recognizer for $L_1$ could go into an infinite loop! Let $L_1$ be $\emptyset$. We could write an `inL1` that looks like:

```java
bool inL1(string w) {
    while (true) {
        // loop forever
    }
}
```
Some Closure Properties of \textbf{RE}

Imagine we wrote the same pseudocode:

```cpp
bool inL1uL2(string w) {
    return inL1(w) || inL2(w);
}
```

⚠️

Our recognizer for $L_1$ could go into an infinite loop! Let $L_1$ be $\emptyset$. We could write an $\text{inL1}$ that looks like:

```cpp
bool inL1(string w) {
    while (true) {
        // loop forever
    }
}
```

What happens when we run $\text{inL1uL2}$ on a string $w \in L_2$? $\text{inL1}$ will loop forever and so $\text{inL1uL2}$ will also loop forever even though it’s supposed to return true.
Some Closure Properties of \textbf{RE}

One way we could address this is to limit the number of steps we run both \texttt{inL1} and \texttt{inL2}. We could write the following two helper methods:

```cpp
bool simulateStepsOfL1(w, s) {
    return value of running s steps of \texttt{inL1}(w);
}

bool simulateStepsOfL2(w, s) {
    return value of running s steps of \texttt{inL2}(w);
}
```
Some Closure Properties of \textbf{RE}

One way we could address this is to limit the number of steps we run both \textsc{inL1} and \textsc{inL2}. We could write the following two helper methods:

\begin{verbatim}
    bool simulateStepsOfL1(w, s) {
        return value of running s steps of inL1(w);
    }

    bool simulateStepsOfL2(w, s) {
        return value of running s steps of inL2(w);
    }
\end{verbatim}

And then \textsc{inL1uL2} would look like:

\begin{verbatim}
    bool inL1uL2(string w) {
        for (s = 0 to infinity) {
            if (simulateStepsOfL1(w, s) || simulateStepsOfL2(w, s)) {
                return true;
            }
        }
    }
\end{verbatim}
Bonus, optional exercise: Repeat what you did in 3a) and 3b), except proving that the R and RE languages are closed under concatenation.