Announcements
1. Our midterm will be a 24-hour take-home open-book/notes. You can take it any time between 10/24 and 10/28.
2. Homework assignment 6 is due Friday.

Pushdown Automata and CFGs
1. Last time we discussed the operation of a new type of automaton, the pushdown automaton. Today, I want to start by thinking about a few examples of PDAs that recognize simple languages.
   - First, I would like you to think about how to construct a PDA that recognizes $L_{add} = \{1^i + 1^j = 1^{i+j} \mid i, j \geq 0\}$
     
     The answer to this question should look something like:

     ![Diagram 1]

     The first two states to the right of the start state push as many 1s on the stack as there are before the = sign and make sure that there is exactly one + sign. The last state makes sure that the number of 1s after the equal sign matches those seen before.

   - Next, what languages do you believe are described by the following two PDAs:

     ![Diagram 2]

     Actually, this is a trick question. Both PDAs accept the same language:
     
     $L_{eq\_occur} = \{w \mid w \in \{a, b\}^* \text{ and } w \text{ contains as many } a's \text{ as } b's\}$

     The machine with the states named A and B was my first attempt and may be a bit easier to understand. The idea is that as one scanned the symbols of some input $w$, one could plot the difference between the number of as and bs (i.e., $\#(a) - \#(b)$).

     This value must start as 0 and end as 0 if $w$ is in the language. It may be positive or negative in between. Between any point where it is positive and one where it is negative (or vice versa) it must equal 0 at some symbol. The idea behind this machine is that it will spend its time in state A while $\#(a) - \#(b)$ is positive, spend...
As a final example, let's consider how to construct a PDA that recognizes the language $L_{\text{diff}} = \{#p_i# | n ≥ 2, \text{ for some } i, j \; p_i \neq p_j\}$

- This is an interesting example because to solve it you have to take non-determinism seriously.
- The machine cannot check every pair of $1^{p_i}$ and $1^{p_j}$ because if if pushes $p_i$ symbols on the stack to remember the value of $p_i$, once it clears the state to see if $p_i$ equals $p_j$ the $p_i$ symbols are gone and cannot be compared to any other symbol of $1$s.
- The first thing to observe, is that if there is any pair of $1^{p_i}$ and $1^{p_j}$ of different lengths, one of them can be $p_1$. This is because $p_1$ must be different from one of $1^{p_i}$ and $1^{p_j}$ and that string together with $1^{p_i}$ is just as good an example of two strings of length of different length as $1^{p_i}$ and $1^{p_j}$.
- Next, the trick is to build a machine that just guesses which string is the one (or one of the ones) of different length.
- The machine below does this. In state “first w” it fill the stack with as many ones as it finds in the first string. In state “pick w” it uses non-determinism to guess which # precedes a string of $1$s of a different length. It can also use non-determinism to skip “pick w” and move right to $w'$ if it guesses that the second substring does not match the first. In state $w'$, it verifies its guess reaching the final state only if the number of $1$s in the stack is different from the length of the selected string of ones.

PDAs Formally

1. A pushdown automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where:

   - $Q$ is a finite set of states,
   - $\Sigma$ is a finite input alphabet,
   - $\Gamma$ is a finite stack alphabet,
   - $\delta : Q \times \Sigma \times \Gamma \to P(Q \times \Gamma)$ is the transition function, and
   - $F \subset Q$ is the set of final or accepting states.

2. We say that a PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ accepts a string $w = w_1w_2...w_n, w_i \in \Sigma$ if $\exists q_1, ..., q_n \in Q$ and $s_0, s_1, ..., s_n \in \Gamma^*$ such that:
   - $s_0 = \epsilon$
   - $\forall i, 1 ≤ i ≤ n, \exists h_i, p_i \in \Gamma$ and $t_i \in \Gamma$ such that $s_{i-1} = h_it_i, s_i = p_it_i$ and $(q_i, p_i) \in \delta(q_{i-1}, w_i, h_i)$
   - $s_n \in F$
1. Last class, we considered the context-free language

\[ L_{\text{add}} = \{1^i + 1^j = 1^{i+j} \mid i, j \geq 1\} \]

which is generated by the context-free grammar:

\[
\begin{align*}
L & \rightarrow 1L1 \mid + R \\
R & \rightarrow 1R1 \mid =
\end{align*}
\]

2. We have seen that we can establish the fact that a string belongs to the language of a grammar like this by describing a derivation of the string from the start symbol of the grammar.

\[
L \Rightarrow 1L1 \Rightarrow 11L11 \Rightarrow 11 + R11 \Rightarrow 11 + 1R111 \Rightarrow 11 + 1 = 111
\]

3. As an alternative to using derivations to show that a given string belongs to the language of a grammar G is to discover a parse tree for the string relative to the grammar.

- To illustrate this idea, consider the parse tree for “11+1=111” relative to our grammar for \( L_{\text{add}} \):

![Parse Tree Diagram]

- In general, we say that a labeled tree is a parse tree relative to a context-free grammar \( G = (V, \Sigma, R, S) \) for a sentential form \( w \in (\Sigma \cup V)^* \) if:
  - The root is labeled with the start symbol, \( S \).
  - All interior nodes of the tree a labeled with symbols in the alphabet of variables, \( V \).
  - All leaves are labeled with terminal symbols and the sequence of symbols found as one visits the leaves in order form left to right along the frontier forms the string \( w \).
  - If an interior node is labeled with \( L \in V \) and its children are labeled with the symbols in the string \( \beta \in (\Sigma \cup V)^* \) then \( V \rightarrow \beta \in R \).

**Ambiguity**

1. The grammatical structure that a grammar induces on a sentential form can play a role in the way we associate meaning with the string.

- Consider the string 9 − 9 − 9 − 9 (which is intended to be interpreted as a very simple arithmetic expression involving three subtractions).

- This string clearly belongs to the language of both of the following grammars:

\[
\begin{align*}
L & \rightarrow L - 9 \mid 9 \\
R & \rightarrow 9 - R \mid 9
\end{align*}
\]

but the parse trees induced on 9−9−9−9 by these two grammars suggest different interpretations of the string as an arithmetic expression:
The tree on the left suggests the expression should be interpreted with left associativity for the subtraction operations so that the result is equal to $((9 - 9) - 9) = -18$. The tree on the left suggests right associativity so that the result would be $(9 - (9 - (9 - 9))) = 0$.

2. In the example above, as long as we know which of the two grammars should be used we would know how to interpret input strings. There are, however, grammars for which a single string may have both multiple derivations and multiple parse trees.

- Consider the following grammar for the sort of simple subtraction languages we have been discussing:

$$E \rightarrow E - E \mid 9$$

- Relative to this grammar, the string $9 - 9 - 9$ has two parse trees with fundamentally different structures:

- The tree on the left suggests left-association so that $9 - 9 - 9 = (9 - 9) - 9 = -9$ while the tree on the right suggests right-association so that $9 - 9 - 9 = 9 - (9 - 9) = 9$.

3. If for any string $w$, a context-free grammar induces two or more parse trees with distinct structures, we say the grammar is ambiguous. Otherwise we say the grammar is unambiguous.

4. It should be made clear that ambiguity requires that a string in a language have two different parse trees relative to a grammar, not just two different derivations.

- Consider the following grammar which describes a language similar to the language described by the grammars in the previous examples (in the new grammar we allow two values — 0 and 9 — instead of just 9’s in our expressions):

  $L \rightarrow L - V \mid V$
  $V \rightarrow 0 \mid 9$

- Because one of the rules in this grammar has multiple variables on its right side, this grammar will derive sentential forms containing multiple variables. In a derivation, this will give us the freedom to choose which variable to expand in the next step, leading to multiple derivations for most strings in the language.

- As an example, the string $9 - 9 - 9$ can be derived as:
\[ L \Rightarrow L - V \Rightarrow L - V - V \Rightarrow V - V - V \Rightarrow 9 - V - V \Rightarrow 9 - 9 - V \Rightarrow 9 - 9 \Rightarrow \]

or

\[ L \Rightarrow L - V \Rightarrow L - 9 \Rightarrow L - V - 9 \Rightarrow L - 9 - 9 \Rightarrow V - 9 - 9 \Rightarrow 9 - 9 - 9 \Rightarrow \]

• Despite these and several other possible derivations for \( 9 - 9 - 9 \), there is only one possible parse tree for \( 9 - 9 - 9 \) relative to this grammar:

\begin{itemize}
  \item Thus, the grammar induces an unambiguous structure on this string (and all other strings in the language it describes).
\end{itemize}

5. Actually, there are several, provably-equivalent ways to define ambiguity. In particular:

**Definition:** We say that a derivation is leftmost if at each step a rule is applied to expand the leftmost variable in the current sentential form.

**Theorem:** Given a context-free grammar \( C \) and a string \( w \), there are two distinct parse trees for \( w \) relative to \( C \) if and only if there are two distinct left-most derivations of \( w \) relative to \( C \).

• We will not worry about the details of the proof of this theorem, but the intuition is that given a parse tree, there is an algorithm (traverse the tree so that each node’s subtrees are visited left to right) that can be used to generate a corresponding left-most derivation. Given two distinct parse trees, this algorithm generates different left-most derivations. Similarly, given a left-most derivation, there is an algorithm that builds a parse tree in such a way that distinct derivations must produce distinct trees.