Announcements
1. Our midterm will be a 24-hour take-home open-book/notes. You can take it any time between 10/22 and 10/27.
2. Homework assignment 6 is due Friday.

Parse Trees
1. Even though we were discussing pushdown automata at the end of last class, I want to briefly return to context-free grammars to discuss an interesting property of this approach to describing languages.

2. In previous classes, we considered the context-free language

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

which is generated by the context-free grammar:

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

3. 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.

\[
\begin{align*}
L & \Rightarrow 1L1 \Rightarrow 11L11 \Rightarrow 11 + R11 \Rightarrow 11 + 1R111 \Rightarrow \\
& 11 + 1 = 111
\end{align*}
\]

4. 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 \(L_{add}\):

- 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:

\[ L \to L - 9 \mid 9 \]
\[ R \to 9 - R \mid 9 \]

but the parse trees induced on \(9 - 9 - 9\) by these two grammars suggest different interpretations of the string as an arithmetic expression:

\[
\begin{align*}
-18 &= L - L - 9 \\
0 &= R - R - 9
\end{align*}
\]

• 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:

\[ E \to E - E \mid 9 \]

• Relative to this grammar, the string \(9 - 9 - 9\) has two parse trees with fundamentally different structures:

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):

\[ L \to L - V \mid V \]
\[ V \to 0 \mid 9 \]

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

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:

```
         L
        /\  
       /   \  
      L - V /   \  
     /     /     
    /     /     
   V - 9 / 9 
  /     /    
 9     9 
```

• Thus, the grammar induces an unambiguous structure on this string (and all other strings in the language it describes).

### Pushdown Automata

1. Last time, I introduced a new model of computation call the pushdown automaton. It processes inputs sequentially while making state transitions, but it can store data in a stack as it reads the input and its transitions can depend on the symbol on top of the stack.

2. At the end of class we were exploring a simple example of a PDA that recognized the language \( \{1^n = 1^n : n \geq 0\} \).

   • The diagram below provides an informal description of a pushdown automaton that recognizes this language.

   ![Diagram of a pushdown automaton recognizing \( 1^n = 1^n \)]

   • Recall that this machine has one feature that is somewhat an artifact of the way Sipser chooses to describe PDAs — namely, his formalism provides no way for the machine to sense if its stack is empty. This will lead most of our machines to include a start state with just one transition that puts a recognizable symbol at the bottom of the stack before processing any input.

   • Also note that this trick depends on epsilon-transitions. In particular, for now, all PDAs are non-deterministic.

   • Let’s trace through the steps this machine takes processing the input \( 11=11 \).

3. To help solidify your understanding of this informal introduction to pushdown automata, I would like you to design a machine (or at least a modification of another machine) working with a fellow student, so...

   • 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 of a pushdown automaton recognizing \( L_{add} \)]
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.

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,
   - \(\delta: Q \times \Sigma \epsilon \times \Gamma \epsilon \rightarrow \mathcal{P}(Q \times \Gamma \epsilon)\) 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:
   - \(s_0 = \epsilon\)
   - \(\forall i, 1 \leq i \leq n, \exists h_i, p_i \in \Gamma\) and \(t_i \in \Gamma^*\) such that \(s_{i-1} = h_it_i, s_i = p_it_i\) and \((q_i, p_i) \in \delta(q_{i-1}, w_i, h_i)\)
   - \(s_n \in F\)

Thinking Nondeterministically

1. When we studied finite automata, we started with the deterministic model and then moved on to consider nondeterminism later. With pushdown automata, we have started right away using nondeterminism (at least in the form of epsilon transitions).

2. To reinforce the power of nondeterminism in this model, I want to explore a few solutions to the problem of building a pushdown automaton for the language

\[L_{eq\text{-}occur} = \{w \mid w \in \{a, b\}^* \text{ and } w \text{ contains as many } a\text{'s as } b\text{'s}\}\]

3. First, it should be clear that this cannot be a regular language since if we intersect \(L_{eq\text{-}occur}\) with the language of the regular expression \(a^*b^*\) we get \(\{a^n b^n \mid n \geq 0\}\) which is clearly not regular.

4. So, we should expect to have to use the stack in some way to keep track of how many letters of each type we have seen. Any suggestions?

5. If you are stumped, consider this machine and see if you can explain how it works intuitively:}

---

\footnote{The transitions between the two \(\geq\) states in this version of the machine for \(L_{eq\text{-}occur}\) violate Sipser’s formalism by pushing two symbols (a or b and \$) at the same time. We view these transitions as a shorthand for a series of transitions like those shown in the version of the same machine shown below:}
6. We cannot use one stack to simultaneously keep track of two separate counter — one for a’s and one for b’s. What this machine does instead is use the stack to keep track of how many more a’s than b’s have been seen when more a’s have been seen. However, when more b’s than a’s have been seen it instead uses the stack to keep track of how many more b’s than a’s.

7. What I really want you to notice is the peculiar way this machine uses nondeterminism.

- The only nondeterministic transitions in the machine are the $\epsilon$ transitions leaving “start” and leading to “final”.
- They both involve the peculiar fact that there is not explicit way to test for empty stack or end of input in Sipser’s model of a pushdown automaton.
- The first $\epsilon$-transition puts a marker at the bottom of the stack so that other states can tell when it is (near) empty.
- The transitions to “final” reflect the fact that there is no deterministic way to make a transition only when the machine reaches end of input.
- To accept an input, a PDA must reach a final state at a point where all of the input has been consumed.
  - Because of the way this machine uses the stack to balance the number of as and bs, it will be in either the “$\#A \geq \#B$” state or the “$\#B \geq \#A$” state when the input runs out.
  - Neither of these states can be final because we should not accept if any a’s or b’s are left in the stack.
  - The $\epsilon$ transitions from the $\geq$ states to “final” let the machine guess that it is at the end of the input whenever the stack is empty. If it guesses correctly, it will be able to accept the input. An incorrect guess will leave it in a final state with no way to consume the remaining input so that branch of nondeterminism will just expire.