Announcements
1. Our midterm will be a 24-hour take-home open-book/notes. You can take it any time between 10/22 and 10/28? Click here to see a sample exam from fall 2017.

2. Homework assignment 6 is available. It is due Friday. By the end of this class I may announce that problem 5 is not due on Friday.

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

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

3. Last time I showed a machine that recognized this language (almost). Unfortunately, I made a mistake and the machine shown on the slides during last class removed the $ that indicated the bottom of the stack when it should not have done so.

- A corrected version of this machine appear below (and in the posted version of the slides from last class period).

- This version works by using an epsilon transition to push a pair symbols onto the stack during what it logically one transition from one of the \( \geq \) states to the other. This is easy enough to do in general, that we will let ourselves label a transition with a sequence of symbols to push on the stack even though Sipser’s formalism doesn’t officially allow this.

4. 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”.

Click here to view the slides for this class
• 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.

5. Once you realize that the machine is being nondeterministic in this case, you might notice some other possibilities.

6. Consider the state “1st letter”.
• If the next input is an a, this state pushes it onto the stack and goes to the “\(#As \geq \#Bs\)” state.
• Notice that if we just allowed an \( \epsilon \) transition from “1st letter” to “\(#As \geq \#Bs\)”, the “\(#As \geq \#Bs\)” state would push the next a onto the stack anyway,
• Notice also that the same is true for bs and the combination of the “1st letter” state and the “\(#Bs \geq \#As\)”

7. So, we can actually get rid of the “1st letter” state and let the start state guess whether the first letter will be an a or b by taking an epsilon transition to the appropriate \( \geq \) state:

8. Now, let’s think about the transitions between the two \( \geq \) states.
• Like the transitions from the “1st letter” state we just eliminated, each of these transitions consumes a letter of input and pushes the identical letter onto the stack.
• In addition, if we had just made the transition without consuming input or pushing anything on the stack, the destination states would consume and push the same symbols.

9. So, we can let the machine do more guessing. Whenever the stack becomes empty, we can just guess whether the remaining input will start with an a or b and epsilon transition to the appropriate \( \geq \) state. If we guess wrong, we can just jump back. This leads to:

10. Can you see where this is going? We have given the machine the option of jumping back and forth between the two \( \geq \) states at will. It is almost as if it doesn’t matter which one of the two \( \geq \) states the machine is in. In fact, it doesn’t. We can merge them giving:
11. As a final example, let’s consider how to construct a PDA that recognizes the language

\[ L_{\text{Unary-diff}} = \{ \#^{p_i} \#^{p_j} \ldots \#^{p_n} \# \mid |p_i| > 0 \text{ for some } i, j \text{ and } p_i \neq p_j \} \]

- This is the language of non-empty unary strings separated by pound signs which contain at least two unary substrings of different lengths.
- 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 it pushes \( p_i \) symbols on the stack to remember the value \( p_i \), 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 string of 1s.
- The first thing to observe, is that if there is any pair of \( 1^{p_i} \) and \( 1^{p_j} \) of different lengths, we can replace one of them with \( p_1 \) and still have a pair of distinct strings. This is because \( p_1 \) must be different from one of \( 1^{p_i} \) and \( 1^{p_j} \) and that string together with \( 1^{p_1} \) is just as good an example of two strings of different length as \( 1^{p_i} \) and \( 1^{p_j} \).
- Next, the trick is to have the machine guess which string is the one (or one of the ones) of a length different from the length of \( p_1 \).
- The machine below does this. In state “first w” it fills 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 1s 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 leaving state \( w' \) only if the number of 1s in the stack is different from the length of the selected string of ones.
- Depending on whether there were fewer 1s in the second substring (it reached a \( \# \) before the stack was empty) or more 1s in the second substring (the stack ran out of 1s while there was still another 1 in the input) it either moves to “find end” or “final”.
- We need states “find end” and “final” because to accept a PDA must read all the way to the end of the input and be in a final state when it gets there. Also for this language, the last symbol before the end must be a \( \# \). So...
- We go to and state in “find end” as long as there are 1s still remaining. It transfers to “final” each time it sees a \( \# \). If the \( \# \) is the end of input, the machine accepts. If it see another 1 it returns to “find end” where it can skip over 1s looking for another \( \# \) that is potentially the end of the input.
The Pumping Lemma for Context-Free Languages

1. Just as we used the Pumping Lemma to prove that certain languages were not regular, there is a pumping lemma for context-free languages that provides a way to show that certain languages are not context-free.

   **Lemma:** If $C$ is a context-free language, there is a number $p$ known as the pumping length such that if $w \in C$ and $|w| \geq p$, then we can partition $w$ into five substrings $w = uvxyz$ such that:

   - $|vxy| \leq p$
   - $|vy| > 0$
   - $\forall i \geq 0, uv^i xy^i z \in C$

2. Since a pushdown automaton only has finite states, the same argument that states must be repeated on long inputs that applies to finite automata applies to PDAs, but...

3. If a PDA enters the same state with different elements in its stack, it isn’t necessarily possible to repeat the process indefinitely by just duplicating the string that brought the PDA back to a previously visited state. The “effective state” of a PDA includes both its current state and its stack contents. There are infinitely many possible stack contents, so a machine may not return to the same effective state no matter how long an input is processed.

4. As a result, the Pumping Lemma for CFLs is not derived from an argument based on PDAs.

5. Instead, the proof of the Pumping Lemma for CFLs follows from two properties of parse trees:

   (a) If a non-terminal is repeated on some path from the root to the leaves of a parse tree for a sentence of a language, then we can increase or decrease the number of occurrences of the symbol on the path leading to the duplication of substrings as required in the Lemma,

   (b) If a parse tree is sufficiently large, it must have repeated symbols on some path.

6. Suppose that some variable $R$ appears more than once on a path from the root to some leaf of a parse tree relative to a CFG as shown in the figure below.

   ![Parse Tree Diagram]

   - In this case, we could replace the subtree rooted at the second occurrence of $R$ with a copy of the full tree rooted at the first occurrence of $R$ as shown below:

   ![Replacement Parse Tree Diagram]

   - If (as shown in the figure), $x$ was the string at the frontier of the subtree rooted at the second occurrence of $R$, $vxy$ was the string on the frontier of the subtree rooted at the first $R$, and $uvxyz$ was the frontier of the full parse tree, this shows that we could
construct a parse tree for \( uv^2xy^2z \) showing that this string must be in the language of the grammar.

- Simply repeating the process of replacing the last smallest subtree rooted at \( R \) with the subtree rooted at the preceding copy of \( R \) shows that \( uv^ixy^iz \) is in the language for all positive values of \( i \).

- Similarly, replacing the subtree rooted at the first occurrence of \( R \) with the subtree rooted at the second shows that \( uv^0xy^0z \) must be in the language.

7. All we need to show to prove that sufficiently large strings in a CFL can be pumped is that some variable must repeat along a path from the root to the leaves of the parse tree of any such string.

- Given a CFL \( C \) we know that we can find some CFG \( G = (V, \Sigma, R, S) \) such that \( L(G) = C \).

- Suppose that \( N \) is the largest number of symbols in the right-hand side of any rule in \( R \).

- Consider how the height of a parse tree is related to the length of the associated string \( w \):

  - If the parse tree has height 1, then it has at most \( N \) leaves:

    

    - If the parse tree has height 2, then it has at most \( N^2 \) leaves since each of the \( N \) symbols one step from the root can have produced at most \( N \) leaves.

    

    - In general, therefore, a tree of height \( H \) can have at most \( N^H \) leaves. In other words, if \( w \in L(G) \) and \( |w| \geq N^H \), then the parse tree for \( w \) must have some path containing more than \( H \) interior nodes.

    

    - Suppose that we choose \( w \) such that \( |w| \geq N^{|V|+1} \) where \( V \) is the set of variables/non-terminals of \( G \). Then the parse tree for \( w \) must contain some path of length greater than \( |V| \). That is, there must be more nodes in the path then there are variables in the grammar.

    

    - Just as in the proof of the Pumping Lemma for regular languages, we observe that the sequence of variables that label a path of length greater the number of variables must contain at least one repeated variable.
8. This is how we determine the pumping length $p$ described in the Pumping Lemma. If $p = N|V|+1$ then the parse tree for any string of length $p$ or greater will contain a path with a repeated variable and therefore the tree and the string can be pumped.

9. To ensure that the $vxy$ we are pumping satisfy all of the conditions of the Pumping Lemma, we have to be a bit careful. Rather than picking any parse tree for $w$, we have to pick the smallest one. This ensures we cannot have used a rule like $R \rightarrow R$ on the path we select (since otherwise we could get a smaller parse tree by removing it). This means that either $v$ or $y$ must be non-empty. Also, if there are multiple repeated variables, we must pick a pair as close to the leaves as possible to ensure that $|vxy| < p$. 