Main topics of the week:

- **Construction of CFG from a PDA**
- **Pumping Lemma for CFLs**

Constructing a CFG from a PDA.

Recall that we make some simplifying assumptions about our PDA:

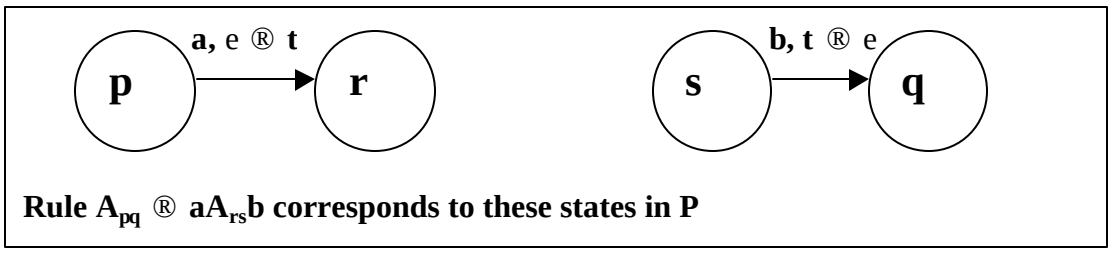
1. There is a single accept state.
2. The stack is emptied before accepting.
3. Each transition either pushes or pops a symbol from the stack, but not both.

The gist of our grammar construction is to define a variable for each pair of states such that the variable generates exactly the strings that move the PDA between the two states, leaving the stack in the same condition (i.e., strings that move between P between the states starting and ending with an empty stack.) Suppose that we can construct such a grammar. If the pair is the start state and accept state, then the strings that move between them with empty stack are exactly those strings accepted by P (remember that we guaranteed that the stack is emptied before acceptance). The reason for talking about an arbitrary pair of states is that we will prove this equivalence between generated strings and moving through P with empty state by induction. The goal is to get it to be true for the start and accept state, and we accomplish this by an induction proof that shows it is true for every pair.

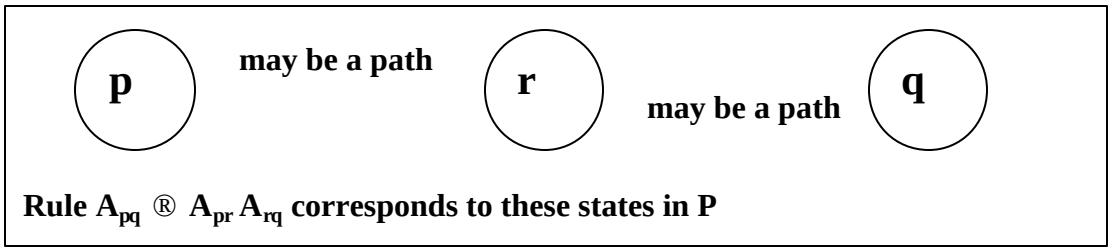
We define our grammar G as follows, where $P = \{Q, \Sigma, \Gamma, \delta, q_{start}, \{q_{accept}\}\}$:

1. The variables of G are $\{ A_{pq} \mid p,q \in Q\}$ [one variable for each state pair]
2. The start variable of G is $A_{start,accept}$ [this will show that G matches P]
3. For each $p,q,r,s \in Q$ and $t \in \Gamma$ and $a,b \in \Sigma_\varepsilon$ where $(r,t) \in \delta(p,a,\varepsilon)$ and $(q,\varepsilon) \in \delta(s,b,t)$, we have the rule $A_{pq} \rightarrow aA_{rs}b$ [if there are states where we initially push a symbol from p, and pop that symbol to get to q, add this to the grammar]
4. For every $p,q,r \in Q$, we have the rule $A_{pq} \rightarrow A_{pr}A_{rq}$ [rule for every indirect path – but adding no terminals]
5. For every $p \in Q$, we have the rule $A_{pp} \rightarrow \varepsilon$ [null termination for the diagonal pairs]

The following pictures might help to see these rules.



Rule A_{pq} ® $aA_{rs}b$ corresponds to these states in P



Rule A_{pq} ® $A_{pr}A_{rq}$ corresponds to these states in P

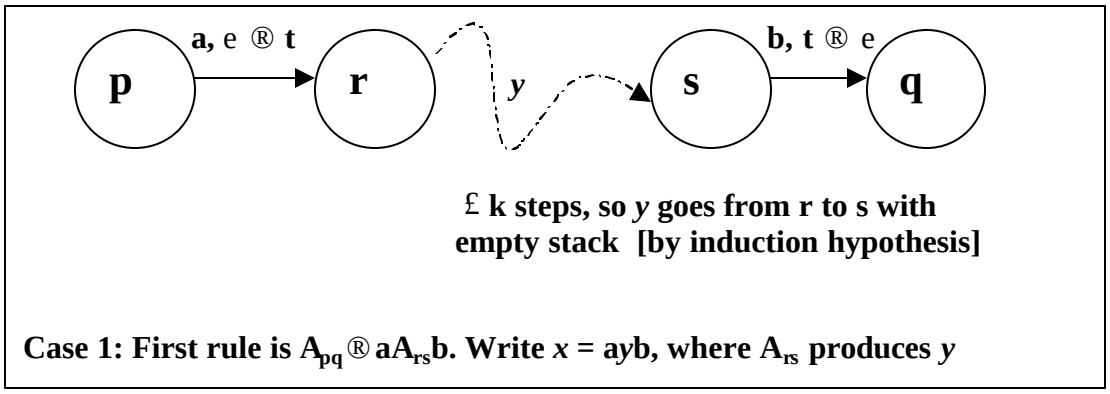
First Claim: If A_{pq} generates the string x, then x can bring P from p with empty stack to q with empty stack.

Proof: This proof will proceed by induction on the number of steps in the derivation:

If the derivation has one step, then there is a single rule that substitutes just terminals. In our definition of G, the only such rules are the ones from (5), i.e., $A_{pp} \rightarrow \varepsilon$. Thus the generated string must be ε, and ε certainly can take p with empty stack to p with empty stack. (It's just the transition $\varepsilon, \varepsilon \rightarrow \varepsilon$ which has no effect.)

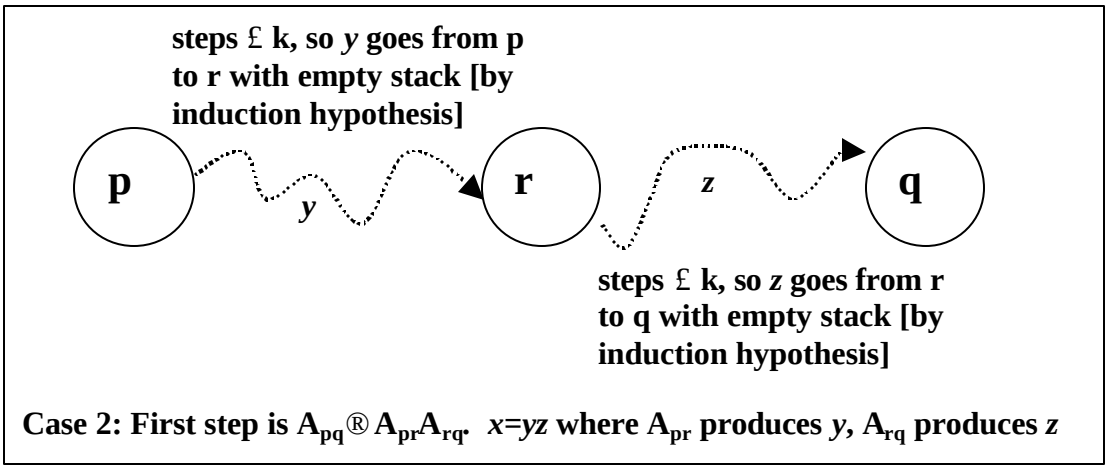
For the induction step, assume the claim hold for any derivation with k or fewer steps ($k \geq 1$). Suppose we have a derivation with k+1 steps. Since our rules have just two forms (in 3 and 4), we'll consider each of these possibilities:

If the first step uses a rule $A_{pq} \rightarrow aA_{rs}b$, think about the portion of x minus the a and b, i.e., x=ayb. This portion, called y, is generated by A_{rs} with k steps (all but the first step in the derivation for x). By our induction assumption, this means that y must take r with empty stack to s with empty stack. By our definition of G, this means that if we start with x=ayb in p with empty stack, we can transition to r with input a and pushing t, then transition somehow to s with input y, leaving the stack alone, so that t is still on the stack when we transition from s to q on input b, popping t, and thus x gets us to q with empty stack.



Case 1: First rule is A_{pq} ® $aA_{rs}b$. Write *x* = a*y*b, where A_{rs} produces *y*

For the second case, where the first step is a rule $A_{pq} \rightarrow A_{pr}A_{rq}$, we must have x = yz, where y and z are the portions generated by the two variables on the right. Since the total steps in the derivation is k+1, each of A_{pr} and A_{rq} can have at most k steps. So again by the induction hypothesis, y goes from p to r with empty stack, and z goes from r to q with empty stack. Obviously, then x goes from p to q with empty stack.



Case 2: First step is A_{pq}® $A_{pr}A_{rq}$. *x=yz* where A_{pr} produces *y*, A_{rq} produces *z*

This completes the induction proof, so our first claim is now proved.

Second Claim: If x brings P from p with empty stack to q with empty stack, then A_{pq} generates x.

Proof: Again we will use induction, this time on the number of steps in the computation taking place in P.

For the basis, consider the case where the computation has zero steps. This means it starts and ends in the same state p. So we need to show that A_{pp} generates x. Since there are zero steps in the computation, no input is read, hence x must be the empty string ε. By our clever definition of G (rule 5), A_{pp} just happens to generate ε.

For the induction step, we assume that the claim holds if the number of steps in the computation is k or fewer, and suppose we have a computation that takes k+1 steps. There are two possibilities: the stack may be empty only at the beginning and end, or it may be empty somewhere in between as well.

For the first case, suppose t is the symbol pushed on the first transition of the computation. Then since this is the case where the stack is never empty, and we know that P cannot push and pop in one step, t must still be there at the last transition. You can see where we're going – let a and b be the input for these first and last transitions, r the second state, and s the next to last state. Then we appeal to our definition of G to know it has the rule $A_{pq} \rightarrow aA_{rs}b$. We can realize x as aby, and know that y brings r to s, leaving t on the stack. So of course y brings r with empty stack to s with empty stack. Since we are taking out the first and last steps, the number of steps in this computation is certainly no more than k, hence A_{rs} generates y by the induction hypothesis. Combine this with the rule above, and we get a derivation for x.

For the second case where the stack does become empty in the middle, let r be the state where that happens, and y and z be the corresponding breakdown of the input x. The number of steps on either side of r is no more than k, so the induction assumption says there must be derivations such that A_{pr} generates y and A_{rq} generates z. Since we were clever enough to include all pair rules, we know the rule $A_{pq} \rightarrow A_{pr}A_{rq}$ is in G, thus we have our derivation of x.

So where are we? We have proved our two claims about pairs of states and the grammar derivations. In particular for the start and accept states this means that we have proved our theorem about the equivalence of pushdown automata and CFLs.

Pumping Lemma for CFLs. For any context free language A there is a pumping length p such that for any string $s \in A$ of length $\geq p$, there are substrings u, v, x, y, and z with s = uvxyz and satisfying:

1. $uv^ixy^iz \in A$ for all $i \geq 0$
2. $|vy| > 0$
3. $|vxy| \leq p$

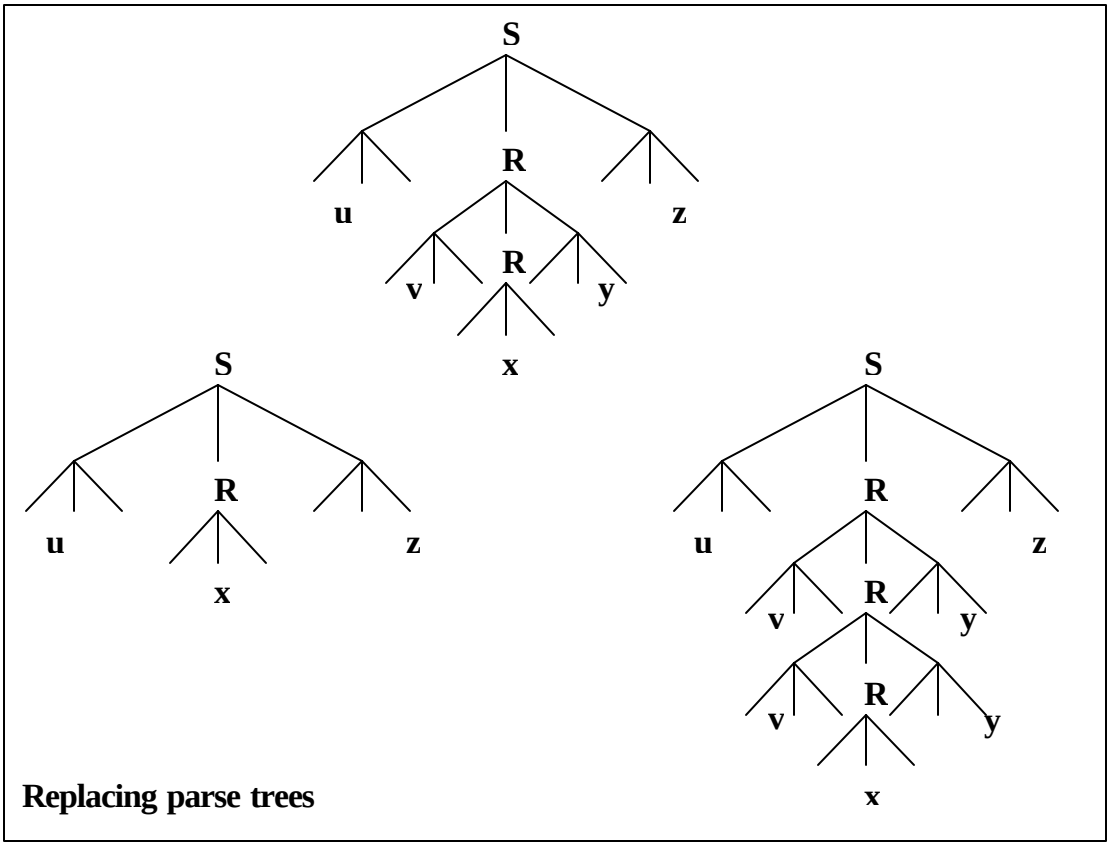
So basically, s can be divided into five pieces, and the two middle flanking pieces v and y can be pumped (together, not individually). The second condition requires that at least one of v and y not be the empty string, and the third condition keeps the middle part close to the pumping length, even for a very long string. As before, this third condition will sometimes be useful in proofs.

The basic idea of this pumping lemma is that a sufficiently long string will have a tall enough parse tree so that we can find a repeated variable on a path through the tree. Because of the way variable substitution is context free in derivations, we can either collapse the tree between the repeated variable (thus cutting out the flanks of the subtree) or repeat the subtree more times at the second occurrence. The picture shows what is going on.

Let G be the grammar for CFL A, and let b be the largest number of symbols in the right hand side of all rules in G. Certainly $b \geq 2$ (otherwise, A would just consist of a finite number of strings of terminals and the lemma would be vacuously true.) What is the significance of b? In a parse tree, it represents the maximum number of children of any node. This means that in the worst case, we have b leaves at depth 1, b^2 leaves at depth 2, and in general, at most b^h leaves in a parse tree of height h. And this translates into a string of length at most b^h when the height of the tree is h. Put another way, this says that if the string is $> b^h$, then the height must be >h, i.e., $\geq h+1$.

So we choose our pumping length as follows. If |V| is the number of variables in G, we choose $p = b^{|V|+2}$. From what we saw above, any string of length $\geq p$ (which is certainly $>b^{|V|+1}$) must require that the height of the tree is at least |V|+2.

Given a string s ∈ A with length at least p, we choose our parse tree carefully, since there may be several for the same string. In particular, we want the tree to be one with the smallest number of nodes. By what we have seen about the height, we know now that there must be a path through the tree of length at least |V|+2, which means it has at least |V|+1 variables (only the leaf of this path is a terminal). This in turn implies that some variable must be repeated among the |V|+1 variables of the nodes. We'll pick a repeating variable R such that R repeats among the lowest |V|+1 variables (i.e., is closest to the bottom).



Replacing parse trees

Now we can divide up s. We let u and z be the portions of s that occur outside the subtree rooted by the higher occurrence of R, we let x be the portion generated by the subtree rooted by the lower occurrence of R, and we let v and y be the portions generated by the upper subtree, but excluding the nested lower subtree. As mentioned earlier, either instance of R may be substituted with either subtree and still obtain a valid tree. When we replace the upper (larger) tree with the lower (smaller) tree, we eliminate u and y. When we replace the lower tree by the upper tree, we replace x by uxy, thus getting the pumping up.

To see that both v and y are not the empty string, suppose that they were. Then the substitution of the small tree for the large tree would still produce s, yet it would have fewer nodes (having eliminated an R), contrary to our choice of the tree. So at least one of v or y must be non-empty.

Finally, for the third condition, consider the length of vxy, which is generated by the upper tree. The height of this tree is $\leq |V|+2$ since we made sure to choose R within the bottom $|V|+1$ variables (and then there's the leaf to give height $\leq |V|+2$). By our earlier argument, the length of the string generated (namely vxy) cannot exceed $b^{|V|+2}$, which is of course our choice for p.

Example of a Non-Context Free Language. The language $A = \{a^i b^j c^k \mid k > i \text{ and } k > j\}$ is not a context free language. Suppose it is context free and let p be the pumping length and let $s = a^p b^p c^{p+1}$. When this is realized as uvxyz, since $|vxy| \leq p$, vxy cannot span more than two different symbols. If v and y contain no c's, just a's and b's, then pumping up as uvvxyyz would increase the count of a's and/or b's, but not c's, and this would mean the string uvvxyyz is not in the language A since the count of c's would not exceed both the counts of a's and b's. Otherwise, v and y must contain c's, (and might contain b's, but no a's). Then pumping down as uxz would decrease the count of c's, but leave the a's alone, again meaning that uxz is not in the language since it would not have less a's than c's. In either case we have reached a contradiction, so A must not be context free.