| **CS 4510 Automata and Complexity** | March 3 |
|---|---|
| Lecture 11 | |
| *Lecturer: Frederic Faulkner* | *Scribe(s):* |

# 1 CFG's and PDA's

**Theorem 1** *A language is context-free if and only if it is recognized by some PDA.*

What this tells us is that two rather different-looking models of computing, i.e. context-free grammars and pushdown automata, are actually equivalent! To prove this, we need to show two things: how to convert a PDA to a CFG, and how to convert a CFG to a PDA.

Of these two proofs, the transformation from CFG to PDA is fairly straightforward, and is omitted (but can be found in the book). We turn our attention to the case of converting a PDA to a CFG.

*Proof.* We start by showing how to convert a pushdown automata to a context-free grammar. Let $P$ be a pushdown automata. Before we begin, we require that

- P has a single accept state
- P always empties the stack before accepting
- On each transition, P either pushes or pops from the stack (but not neither or both)

If we have a pushdown automata that does not meet these requirements it is fairly easy to convert it so that it does. (See book for details.)

So assume that $P$ meets all of those requirements (either because it began that way or because we converted it), and we wish to construct a grammar that generates all the strings $P$ accepts.

The core idea of this transformation is that we will have a bunch of variables of the form $S_{pq}$ for each pair of states $p, q$ in $P$. We will then add rules so that $S_{pq}$ generates exactly those strings $x$ such that if $P$ is in state $p$ with an empty stack, then after reading $x$, $P$ is in state $q$ with an empty stack.

What would the starting variable of this grammar be? Well, we want to generate all the strings that cause $P$ to go from the start state $q_0$ to the single final state $q_f$ (see requirements). Furthermore, the stack is always empty when a machine starts reading a string, and we know from the requirements that $P$'s stack is empty when it accepts. So we want to generate the strings that cause $P$ to go from the start state to the final state and from an empty stack to an empty stack! But that will be exactly those strings generated by $S_{q_0 q_f}$ if we successfully set up our grammar according to our specifications, so that will be our start variable.

Now, let $p$ and $q$ be states of $P$, and lets think about the ways that we can get from $p$ with an empty stack to $q$ with an empty stack.

**CASE 1:** As we travel from $p$ to $q$, the stack becomes empty one or more times before we get to $q$. In this case, let $r$ be the first state we reach where the stack is empty. This

means that the first part of our string took us from $p$ with an empty stack to $r$ with an empty stack, while the remainder took us from $r$ to $q$ with an empty stack. So we can add the rule $S_{pq} \to S_{pr}S_{rq}$.

**CASE 2:** As we travel from $p$ to $q$ the stack is never empty until we reach $q$. Suppose that our path is the sequence of states $p, p_1, ..., q_1, q$. We know that from $p$ to $p_1$ we push some character $a$ onto the stack (since every transition either pushes or pops, see requirements), and that $a$ is never popped off the stack until we get to $q$ (since the stack is not empty until $q$). But this means that as we transition from $p_1$ to $q_1$, if the stack had started empty, it would have ended empty, since anything we put on top of the $a$ gets removed by the time we get to $q$. So we need the non-terminal $S_{p_1q_1}$. We add the rule $Spq \to s_1S_{p_1q_1}s_2$ where $s_1$ is the input character we read as we go from $p$ to $p_1$, and $s_2$ is the input character we read as we move from $q_1$ to $q$. (Note that either $s_i$ could be epsilon.)

**CASE 3:** $p = q$. For each state $p$, we add $S_{pp} \to \varepsilon$.

And we are done! Our start terminal $S_{q_0q_f}$ now generates exactly those strings which are accepted by $P$.