CS 3719 (Theory of Computation and Algorithms) – Lecture 13

Antonina Kolokolova*

February 4, 2011

In the last lecture we proved one direction of the equivalence between the class of languages accepted by pushdown automata and a class of languages generated by context-free grammars. There, we have shown that for every context grammar we can construct a pushdown automata accepting the same language. In this lecture, we will prove the opposite direction: starting from a pushdown automaton we will construct a corresponding context-free grammar.

Lemma 8. *For every pushdown automaton $N = (Q, \Sigma, \Gamma, \delta, q_0, F$ there exists a context-free grammar G such that $\mathcal{L}(N) = \mathcal{L}(G)$.*

Proof. We need to create a grammar G that would generate a string w if and only if N accepts w. That is, the start symbol S of G would derive w if and only if there is a computation path (possibly with loops) from q_0 to some $q_a \in F$ in N. For simplicity, let us assume that there is just one accept state q_a (otherwise, introduce a new state and make ϵ-transitions from everything else in F to this new state q_a).

How do we encode a computational path in a PDA by a grammar? The is a problem: every rule of the grammar must be finite, and yet somehow it needs to deal with the stack of the PDA, which can be arbitrary large. One way to get around this problem is to look at the PDA as composed of pieces that start and end with the same stack content, and make up a rule for each such piece. In the examples we saw our PDAs would always accept when its stack is empty, so at least for the pair (q_0, q_a) the property that the stack is the same in both situations applies.

Now, how do we encode what a PDA is doing between two states q_0 and q_a, or, more generally, between q_i and q_j where its stack content is the same? At some point after the start state, if N uses its stack at all, it will put a symbol on the stack. Then sometime before the accept state it will pop a symbol off the stack. There are two possibilities here. Either the first transition from q_i puts an element (say, t) on the stack and the last transition to q_j takes that same element t out, or the element t is popped sometime before reaching q_j. In the

*The material in this set of notes came from many sources, in particular "Introduction to Theory of Computation" by Sipser and course notes of U. of Toronto CS 364.

first case, a rule encoding this part of the computation is of the form $A \to aBb$, where a, b are input symbols (possibly ϵ) corresponding to these transitions. Otherwise, if t could be popped somewhere before reaching q_j we can encode this computation by a rule of the form $A \to BC$ – the place where B ends and C starts corresponds to a state q_k with the same content of the stack as in q_i and q_j.

At this point, let us name the variables of the grammar. Since we are talking about encoding paths from one state to another, we introduce a variable $A_{q_i q_j}$ for every pair of states $q_i, q_j \in Q$. That is, if N has n states, then G will have n^2 variables. The start variable will be A_{q_0,q_a} (since we assumed there is only one accept state). The non-terminals of G are the input alphabet of N, so Σ is the same (after all, that is the alphabet of the string w). Now we just need to describe the rules R of G.

Before we proceed describing the rules formally, let us list all the simplifying assumptions about N that we need. Two of them we mentioned already.

1) N has exactly one accept state q_a.
2) The stack is empty when N accepts.
3) Every transition either pushes a symbol on the stack, or pops a symbol from the stack, but not both.

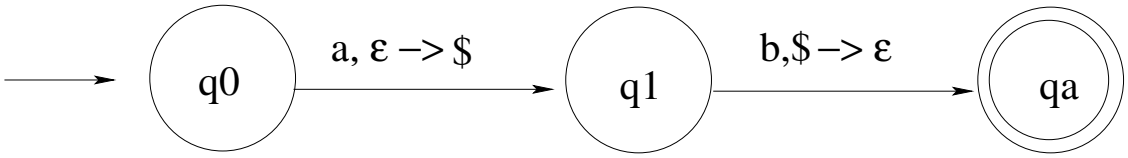
The last assumption helps us to talk separately about "pushing" transitions and "popping" transitions, and avoid discussing what happens on stretches of computation where stack is not used at all. Any automaton can be converted with one with these three properties: we have discussed how to deal with the first two, and for the last one replace every transition which does both by a separate pop and then (on ϵ input) push transition, and every transition that does neither with two pushing then popping an arbitrary dummy symbol.

Now, $V = \{A_{pq} | p, q \in Q\}$, $S = A_{q_0,q_a}$ and Σ is the same. The rules R of the grammar are the following.

1) $\forall p, q, r, s \in Q, t \in \Gamma, a, b \in \Sigma$, if $(r, t) \in \delta(p, a, \epsilon)$ and $(q, \epsilon) \in \delta(s, b, t)$ then add a rule $A_{pq} \to aA_{rs}b$.
2) $\forall p, q, r \in Q$, add a rule $A_{pq} \to A_{pr}A_{rq}$.
3) $\forall p \in Q$, add a rule $A_{pp} \to \epsilon$.

The first rule accounts for the case when there is a path putting t on the stack on the first transition out of p and popping the same t on the last transition into q. The second rule covers the case when a path can consist of several pieces returning to the same state of the stack. Finally, the last rule allows us to stop unwinding the recursion by stating that getting from a state to itself can be done with no input symbols seen.

Example 1. Consider the following very simple automaton accepting just a string ab. We will construct a grammar $G = \{V, \Sigma, R, S\}$ generating the same language.



It has 3 states, so $|V| = |Q|^2 = 9$. $V = \{A_{q_0,q_0}, A_{q_0,q_1}, A_{q_0,q_a}, A_{q_1,q_0}, A_{q_1,q_1}, A_{q_1,q_a}, A_{q_a,q_0}, A_{q_a,q_1}, A_{q_a,q_a}\}$.

Here, let $\Sigma = \{a, b\}$. The start state is $S = A_{q_0,q_a}$. Now we will describe the rules R.

The only symbol ever being put on the stack is \$. So there exists, just one, set of states and symbols giving us a rule of the first type: $p = q_0, q = q_a, r = s = q_1, t = \$$ and a, b are the actual symbols a, b from the alphabet (if we modify the example to have the automaton accepting 01 rather than ab, then we would have $a = 0, b = 1$). So we add a rule $A_{q_0,q_a} \to aA_{q_1,q_1}b$.

Now, for every triple of states we add a rule $A_{pq} \to A_{pr}A_{rq}$. There will be $3^3 = 27$ such rules, starting with $A_{q_0,q_0} \to A_{q_0,q_0}A_{q_0,q_0}$.

Finally, add rules $A_{q_0q_0} \to \epsilon$, $A_{q_1q_1} \to \epsilon$, and $A_{q_aq_a} \to \epsilon$.

In this grammar there is a derivation of ab: $A_{q_0q_a} \Rightarrow aA_{q_1q_1}b \Rightarrow a\epsilon b = ab$. You can check yourself that this is the only string this grammar generates.

To complete the proof we need to show that our construction really works: that is, that the resulting grammar generates every string in $\mathcal{L}(N)$ and does not generate any spurious ones. We will prove these claims in a more general form, for arbitrary start and end states.

Claim 9. *If string x can take N from state p to state q starting and ending with an empty stack, then A_{pq} generates x.*

Proof. The proof is by induction on the number of steps of computation of N on x.

Base case: 0 steps. Then $p = q$, and $x = \epsilon$. So the rule $A_{pq} \to \epsilon$ applies; thus, $A_{pq} \Rightarrow x$.

Induction step: Suppose there is a computation of length $k + 1$ of N on x starting at state p and ending in state q, where the stack is the same (empty) at the start and end of this computation. Consider two cases, as in the construction: either in this computation the first symbol t pushed on the stack was popped in the last step of this computation, or not. In the first case, the rule $A_{pq} \to aA_{rs}b$ applies for some a, b, r, s. Since we assumed that the part of the computation from state r to state s does not touch the symbol t, this computation would be the same on the empty stack. The length of this computation is $k - 1$, so the induction hypothesis applies. Therefore, if $x = ayb$ then $A_{rs} \stackrel{*}{\Rightarrow} y$, and so $A_{pq} \Rightarrow aA_{rs}b \stackrel{*}{\Rightarrow} ayb = x$.

Now suppose that t was popped of the stack, in this computation, before the transition

to state q. Then let r be a state in transition to which t was popped off the stack. By assumption, computations from q to r and from r to p both start and end with an empty stack, and both have length at most k, so the induction hypothesis applies. That is, let $x = yz$, where y is read on the path p to r and z from r to q. Then $A_{pr} \stackrel{*}{\Rightarrow} y$ and $A_{rq} \stackrel{*}{\Rightarrow} z$; thus, since the rule $A_{pq} \to A_{pr}A_{rq}$ is in the grammar G, $A_{pq} \Rightarrow A_{pr}A_{rq} \stackrel{*}{\Rightarrow} yz = x$. □

The corollary of this claim is that every string accepted by N is generated by G. Now it remains to show that every string not accepted by N is not generated by G, or equivalently, taking the contrapositive, that every string generated by G is accepted by N.

Claim 10. *If A_{pq} generates x, then there is a computational path in N on x from p to q starting and ending with the same stack content, which remains unchanged in the computation.*

Proof. The proof is by induction on the length of the derivation. For the base case, the only rule that applies is $A_{pp} \to \epsilon$. For the induction step, analyze the two possibilities for the first rule in the derivation. We omit the details; see Sipser's book Claim 2.30 for details. □

□