CS 3719 (Theory of Computation and Algorithms) – Lecture 12

Antonina Kolokolova[*]

February 1, 2011

Last class we saw that regular languages are a subclass of context-free languages. You may be wondering what other main types of (grammar-defined) languages there are. In mid-50s, Noam Chomsky has classified grammars into 4 main types. Type 3 is regular; type 2 context-free. The other two are context-sensitive and then unrestricted grammars. The last type is equivalent to Turing machines we will soon study; context-sensitive languages correspond to linear bounded automata which we will skip in this course.

Every context-free grammar can be transformed into one in Chomsky Normal Form: there, only rules of the form $A \to BC$ and $A \to a$ are allowed (as well as $S \to \epsilon$ for the start symbol S; however, in Chomsky Normal Form S cannot occur in the body of any of the rules). Here, A, B, C, S are variables and $a \in \Sigma$. We will skip a proof that every context-free grammar can be converted into one in normal form; see e.g. Sipser's textbook for the proof.

0.1 Equivalence between pushdown automata and context-free languages

Now we come to the main theorem of this chapter, one relating pushdown automata and context-free languages.

Theorem 7. *A language A is context-free if and only if there exists a pushdown automata N such that $\mathcal{L}(N) = A$.*

Proof. There are two directions of this theorem, the "if" and the "only if".

We will start by showing that if A is context free then there exists a pushdown automaton N such that $\mathcal{L}(N) = A$. That is, from a grammar $G = (V, \Sigma, R, S)$ generating A (G exists by the definition of context-free languages) we will construct a pushdown automaton $N = (Q, \Sigma, \Gamma, \delta, q_0, F)$ accepting A. Note that Σ is the same in both cases: given a string w of terminals of G we need to determine whether $w \in A$.

[*]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.

Let us analyze how to check a grammar generates a string w. Starting from the start symbol, apply (non-deterministically) a rule with the start symbol as a head: $S- > u_1$, where u is some string of terminals and non-terminals. Now, pick a (usually leftmost) non-terminal in u_1 (say, A_1 and substitute it with the body of a rule with A_1 in the head; call the new string u_2. If it is possible to arrive, after repeating this process a while, to a string of terminals equal to w, then G generates w (that is, some $u_k = w$ where $S \overset{*}{\Rightarrow} u_k$. Notice that if we are trying to determine if w is in the language we don't need to wait until all symbols in u_k are terminals: the moment there is a terminal at the beginning of the string, we can start comparing. For example, if there is a rule $S \rightarrow aAb$, then we can check if our w starts with an a: if not, reject, and otherwise proceed, removing the first symbol of w as well as the first symbol of aAb from consideration: that is, now we want to know whether $Ab \overset{*}{\Rightarrow} w_2 \dots w_n$.

This is the main idea behind building N. Another idea is that it is not necessary to remember which rule was applied before, or where we are in the derivation, as long as we know the intermediate string u_i and know how much of w we already matched. Thus, there is no need to use many states, all the work is done with the stack, which will keep (pretty much) our last derived string.

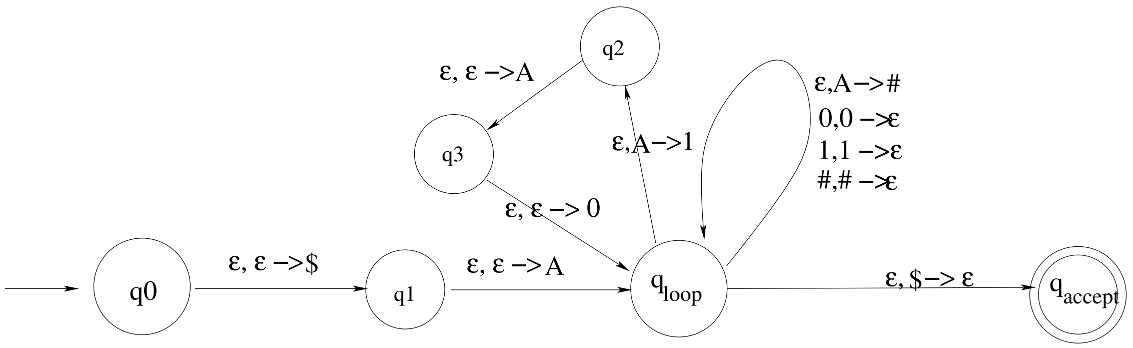
In short, the process of verifying that a string can be generated by a grammar consist of repeating the following until the input string ends (starting with just a string $u_0 = S$). If the first letter of the derived string so far, u_k, is a non-terminal (say A), then pick a rule of the form $A \rightarrow v$ (where v is a sequence of terminals/non-terminals or ϵ) and set u_{k+1} to be u_k with the first letter A replaced with string v. For example, if $u_k = BabCaB$, and there is a rule $B \rightarrow AccBC$, then can set $u_{k+1} = AccBCabCaB$. If there are several possible rules with A in the head, pick one non-deterministically. Now, if the first letter of u_k is a terminal, then match it with the next input letter, and if matched successfully, set u_{k+1} to be u_k without the first letter.

Now we will implement this idea as a pushdown automaton by making one core state q_{loop} with a transition for every rule in R: for R_i of the form $A_i \rightarrow v_i$, put $\delta(q_{loop}, \epsilon, A_i) \rightarrow \{(q_{loop}, v_i)\}$. Also, for every $a \in \Sigma$, add a transition $(q_{loop}, a, a) = \{(q_{loop}, \epsilon\}$. Then add a start state q_0 with a single transition $(q_0, \epsilon, \epsilon) = \{(q_{loop}, S\$)\})$ and an accept state q_{accept} with a single transition $(q_{loop}, \epsilon, \$) \rightarrow \{(q_{accept}, \epsilon)\}$. That is, if the stack became empty, go to accept state and accept if there are no more input symbols. The starting transition just puts the empty string marker and the start symbol of the grammar onto the stack.

You may notice that what we have constructed is not quite a PDA, because in one transition we put whole sequences of symbols onto the stack, whereas by definition we can put only one symbol at the time. This can be done using a PDA by introducing several new states ($m-1$ states if v_i has m symbols). Call these states $q_i^1 \rightarrow q_i^{m-1}$. Now, to implement a transition from (q_i, a, s) to (q_j, xyz) put symbols of xyz onto stack one by one, starting from the last one. That is, the corresponding transitions to put a string xyz on the stack are $\delta(q_i, a, s)) \rightarrow (q_{i,j}^1, z)$, $\delta(q_{i,j}^1, \epsilon, \epsilon)) \rightarrow (q_{i,j}^2, y)$, and $\delta(q_{i,j}^2, \epsilon, \epsilon)) \rightarrow (q_j, x)$. Thus, every "transition" for a rule described above can be done on a pushdown automaton by a sequence of states; note that you need a separate sequence of states for every new transition, even if the string is the same.

Finally, describe the resulting pushdown automaton as follows: $Q = \{\{q_0, q_{loop}, q_{accept}\} \cup E\}$ where E is the set of auxiliary states implementing pushing strings on the stack. Σ, the input alphabet, remains the same. $\Gamma = \Sigma \cup V \cup \{\$\}$ (that is, any terminal or variable can be put on the stack). The start state is q_0, and the transition function is described above. $\square$

Example 1. Consider the following grammar: $G = (\{A\}, \{0, 1, \#\}, \{A \to 0A1, A \to \#\}, A)$. The following automation accepts exactly strings generated by this grammar.


q0
q1
q2
q3
q_loop
q_accept
ε, ε −>$
ε, ε −>A
ε,A−>1
ε, ε −>A
ε, ε −> 0
ε,A−>#
0,0 −>ε
1,1 −>ε
#,# −>ε
ε, $−> ε