CS 3719 (Theory of Computation and Algorithms) – Lecture 11

Antonina Kolokolova*

January 28, 2011

0.1 Regular languages vs. context-free languages

Last class we have seen an example of a nonregular language that is context-free. But are there regular languages that are not context-free? Here, we will show that indeed the class of regular languages is a (strict) subset of the class of context-free languages, by showing that every regular language is context-free.

Note that once we show that pushdown automata accept exactly context-free languages the proof becomes easy: just construct a pushdown automaton out of a NFA which would ignore the stack; you can see that this PDA would accept the same language as the NFA. But as a warm-up for showing the equivalence between context-free languages and pushdown automata, let us show directly that every regular language can be generated by a context-free grammar.

As an example of the ideas involved in the proof, let us show that context-free languages are closed under the star operation. Suppose that G is a context-free grammar. We would like to create a grammar for $\mathcal{L}(G)^*$, where each element is a finite string consisting of 0 or more concatenated strings from $\mathcal{L}(G)$. Let S be the start symbol of G. Now, add a rule $S \to SS|\epsilon$ to the grammar. Suppose a string w consists of k occurrences of strings from $\mathcal{L}(G)$. If $k = 0$, then G generates w by the $S \to \epsilon$ part of the rule. Otherwise, if $k > 0$, then apply the first part of the rule $k - 1$ times, and then use the rest of the rules of the grammar. This shows that every string in $\mathcal{L}(G)^*$ is generated by the new grammar. But how can we make sure that no spurious strings are generated? A trick we can use here is to modify the original grammar first so that there are no more rules using S, other than the start rule. For that, just introduce a new start symbol S_0 and add a rule $S_0 \to S$. Now, think of a parse tree for this grammar. Once a symbol other than S_0 appears on a path from the root down, S_0 cannot appear anywhere below it. So the whole subtree will be generated according to the rules of the original grammar, resulting in a string from G. Since

*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.

this applies to every subtree (except for ones with ϵ-leaves which can be ignored as part of the string), the resulting string will be a concatenation of strings in $\mathcal{L}(G)$.

Theorem 7. *Every regular language is context-free.*

Proof. Let A be a regular language. Then there exists a DFA $N = (Q, \Sigma, \delta, q_0, F)$ such that $\mathcal{L}(N) = A$. Build a context-free grammar $G = (V, \Sigma, R, S)$ as follows. Set $V = \{R_i | q_i \in Q\}$ (that is, G has a variable for every state of N). Now, for every transition $\delta(q_i, a) = q_j$ add a rule $R_i \to aR_j$. For every accepting state $q_i \in F$ add a rule $R_i \to \epsilon$. Finally, make the start variable $S = R_0$. □

Example 1. Consider a deterministic automaton with 3 states accepting all strings containing ab which has transitions $\delta(q_0, a) = q_1$ and $\delta(q_1, b) = q_2$, among others. Here, q_0 is a start state and $F = \{q_2\}$. Add rules $R_0 \to aR_1$, $R_1 \to bR_2$, $R_2 \to \epsilon$ (and similarly for the rest of transitions). Make $S = R_0$. You can check that the resulting grammar generates all strings with a substring ab.

0.2 Arithmetic expressions

A canonical example of use of context-grammars is in parsing. In particular, here we will see how to parse an arithmetic expression using context-free grammars.

Example 2. Consider the following grammar G_1:

$EXPR \to EXPR + EXPR | EXPR * EXPR | (EXPR) | x | y | z | 0 | 1 | 2 | 3$

This grammar generates arithmetic expressions such as $x + 2 * y$. However, there is a problem: it generates it ambiguously, ignoring precedence rules. So evaluating the expression according to the tree might give different answers, depending whether the first rule applied was multiplication or addition.

How would we modify the grammar to make it respect precedence rules? One way of doing it is to give different names to parts of a sum vs. parts of a product and treat them differently.

Example 3. Consider the following grammar G_1, with EXPR the start symbol.

$EXPR \to EXPR + TERM | TERM$
$TERM \to TERM * FACTOR | FACTOR$
$FACTOR \to (EXPR) | x | y | z | 0 | 1 | 2 | 3$

Note that in this case for the arithmetic expression there is only one possible parse tree: