Notes on Sipser: §2.2, “Pushdown Automata”
CSC 341, “Automata, Formal Languages, and Computational Complexity”
Department of Computer Science · Grinnell College
August 31, 2020

By definition, the stack in a pushdown automaton conforms strictly to the invariants of an extremely limited stack data type (without size, without clear, without even an `empty-stack?` procedure). For example, if a pushdown automaton needs to recover the second element of its stack, the one below the top element, it must first pop the top element in order to get the second element to the top. If, after popping and using that second element, the pushdown automaton needs to push the original top element back onto the stack, it must have the identity of that element encoded in its current state. Pushdown automata do not have additional memory caches in which to place such values, even temporarily—all information has to be stored in states or in the stack.

In the construction in Lemma 2.21, in the design of the part of the pushdown automaton $P$ that implements a rule in the context-free grammar $G$ with a string of two or more symbols on its right-hand side, the automaton pushes those symbols onto the stack in reverse order, starting with the rightmost of them and ending with the leftmost. This arrangement strikes some readers, at first glance, as backwards. There’s a good reason for it, though: When the automaton pushes symbols on the right-hand side of the rule onto the stack, they haven’t yet been matched against the input. As the pushdown automaton consumes the input (from left to right), it will need to match the individual terminal symbols to identical terminals on the stack, which can happen only if the next terminal to be matched is at the top of the stack. The symbols from the right-hand side of the rule are pushed onto the stack in the reverse order so as to arrange for the one that will be matched to the input first to wind up on top of the stack, with the ones that will be matched to subsequent input below it.

In the first of the three parts of the construction (at the bottom of page 119) of the transition function $\delta$ for the pushdown automaton $P$, the specification for $\delta(q_{\text{loop}}, \varepsilon, A)$ should be

$$\{(q_{\text{loop}}, w) \mid \text{where } A \rightarrow w \text{ is a rule in } G\}$$

(not “a rule in $R$”). Sipser includes this correction on the Errata page on the Web.

Given a pushdown automaton with $m$ states, the construction in Lemma 2.27 produces a context-free grammar with $m^2$ variables. As in the proof of Theorem 1.39, this is apt to strike practical-minded readers as excessive, and indeed it is usually possible to find a simpler grammar that generates exactly those strings that the pushdown automaton accepts. From the point of view of the automata theorist, however, the objective is not to provide a construction that is easy to implement or well optimized, but rather one that delivers a provably correct result for any given pushdown automaton, regardless of its size or complexity. Resource use is irrelevant, since we don’t actually have to use the constructed grammar or implement it on real equipment; we only need to establish that there exists such a grammar.
As in the proof of Corollary 1.40, there is a little bit of handwaving in Sipser’s observation that “every finite automaton is automatically a pushdown automaton that simply ignores its stack” (page 124), but once again the mapping from deterministic finite automata to pushdown automata that recognize the same languages is straightforward: Use $\emptyset$ for the stack alphabet and use the transition function $\delta$ from the finite automaton to define the pushdown-automaton transition function $\delta'$ as

$$
\delta'(q, a, \varepsilon) = \begin{cases} 
\{(\delta(q, a), \varepsilon)\} & \text{for all } a \in \Sigma, \\
\emptyset & \text{if } a = \varepsilon.
\end{cases}
$$

Now every transition that the deterministic finite automaton makes in processing any input string is exactly duplicated in the pushdown automaton, which therefore ends in the same state and accepts exactly the same strings as the deterministic finite automaton.