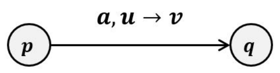
Pushdown Automata (PDAs) and Parsing for CFLs

Pushdown Automata: A *pushdown automaton (PDA)* M is a generalization of an NFA that also includes a stack. That is, $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ where Γ, the alphabet of symbols allowed on the stack and δ is a finite collection of transition rules that say how the stack and state can change based on the current scanned symbol and what is on the top of the stack. (PDAs are the machine model that exactly recognizes CFLs and they need to be nondeterministic to do so. The deterministic version only recognizes a limited subset of CFLs.)

While the formal definition as given in Chapter 2 of Sipser's text only allows the PDA to examine the top symbol on the stack, we allow PDA rules to depend on whether longer strings are on the stack; as noted in Chapter 2, we can simulate such rules with a PDA having extra states. Therefore our PDA diagrams have transitions of the following form for any $a \in \Sigma \cup \{\varepsilon\}$, and any $u, v \in \Gamma^*$,

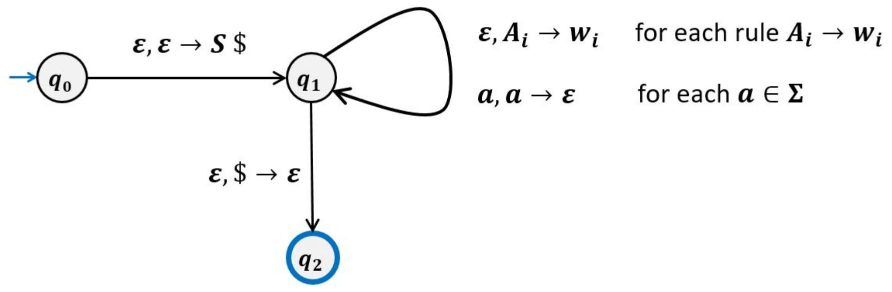


which means that if PDA is in state p, the next scanned symbol is a, and the top portion of the stack matches the string u, then the PDA can read a, replace u on the top of the stack with v and go to state q.

We will show that, via two different methods called Top-Down Parsing and Bottom-Up Parsing, given any CFG G there is a PDA M such that $L(M) = L(G)$. The other direction, showing that for every PDA there is a CFG G such that $L(G) = L(M)$ is much trickier. The textbook gives a very nice proof of that fact, but since PDAs are not a main focus of this course we will only discuss the parsing side since it is related to practical application.

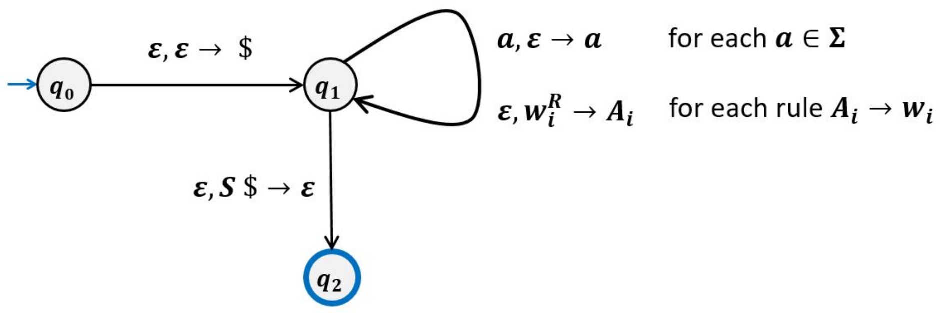
Let G be a context-free grammar with start symbol S and rules $A_1 \to w_1$, $A_2 \to w_2$, ..., $A_r \to w_r$. (Of course, S will typically be at least one of the A_i.)

Top-Down Parsing In top-down parsing, the algorithm marks the bottom of the stack with a special \$ symbol (which is not a symbol in G) and puts S on the stack. The stack is used to store (the unmatched portion) of a string generated by the grammar that it matches with the input. The top-down parser nondeterministically guesses which rules to apply without even looking at the input. When the top of the stack is an input symbol a, it must match with the next symbol of the input. When it is a variable A, the algorithm ignores the input and replaces A with the RHS of some rule it is associated with. The parser accepts if the input is fully matched when the stack only has the bottom-of-stack symbol on it. Here it is in PDA diagram form:



$\varepsilon, \varepsilon \to S\,\$$
q_0
q_1
q_2
$\varepsilon, A_i \to w_i$ for each rule $A_i \to w_i$
$a, a \to \varepsilon$ for each $a \in \Sigma$
$\varepsilon, \$ \to \varepsilon$

Bottom-Up Parsing In bottom-up parsing, the algorithm marks the bottom of the stack with \$ and works from the input string w, pushing symbols of w onto the stack and applying rules backwards (from right to left) on the stack, eventually trying to produce a stack that only has the start symbol S above the bottom-of-stack symbol. Since pushing the symbols of w onto the stack one-by-one means that when we pop them, they will come out in reverse order. Therefore, when we apply rules from right to left, we need to begin with the reverse of the right-side. Bottom-up parsing is also nondeterministic since the reverses of more than one rule may be possible at a given step and one may also choose to transfer an input symbol to the stack. In practice, programming language grammars are designed so that in bottom-up parsing, it is clear what to do next. Here it is in PDA diagram form:


q_0
$\varepsilon, \varepsilon \to \$$
q_1
$a, \varepsilon \to a$ for each $a \in \Sigma$
$\varepsilon, w_i^R \to A_i$ for each rule $A_i \to w_i$
$\varepsilon, S\,\$ \to \varepsilon$
q_2

An Example: Consider the grammar that generates matched parentheses: $S \to (S)$, $S \to SS$, $S \to \varepsilon$. We won't draw the diagrams, but note that the reverse of (S) is $)S($ since the individual symbols in the reverse are not flipped.

Here the parses for the string $(()())$:

Top-Down Parse			Bottom-Up Parse		
State	Input	Stack	State	Input	Stack
q_0	$(()())$	ε	q_0	$(()())$	ε
q_1	$(()())$	$S\$$	q_1	$(()())$	$\$$
q_1	$(()())$	$(S)\$$	q_1	$()())$	$(\$$
q_1	$()())$	$S)\$$	q_1	$)())$	$((\$$
q_1	$()())$	$SS)\$$	q_1	$)())$	$S((\$$
q_1	$()())$	$(S)S)\$$	q_1	$())$	$)S((\$$
q_1	$)())$	$S)S)\$$	q_1	$())$	$S(\$$
q_1	$)())$	$)S)\$$	q_1	$))$	$(S(\$$
q_1	$())$	$S)\$$	q_1	$))$	$S(S(\$$
q_1	$())$	$(S))\$$	q_1	$)$	$)S(S(\$$
q_1	$))$	$S))\$$	q_1	$)$	$SS(\$$
q_1	$))$	$))\$$	q_1	$)$	$S(\$$
q_1	$)$	$)\$$	q_1	ε	$)S(\$$
q_1	ε	$\$$	q_1	ε	$S\$$
q_2	ε	ε	q_2	ε	ε