pushdown automaton in such a way that $\perp$ is both the first symbol pushed to the stack and the last symbol popped from the stack.[6]

Having established all of the technical details, we can now present the formal definition of a pushdown automaton.

Definition 14 (Pushdown automaton)**.** A pushdown automaton is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where

- Q is a finite set of *states*;
- Σ is the *input alphabet*;
- Γ is the *stack alphabet*;
- $\delta\colon Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \to \mathcal{P}(Q \times (\Gamma \cup \{\epsilon\}))$ is the *transition function*;
- $q_0 \in Q$ is the *initial* or *start state*; and
- $F \subseteq Q$ is the set of *final* or *accepting states*.

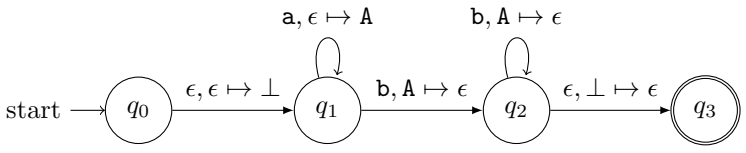
You may have noticed in our definition that the transition function maps to the power set of state/stack symbol pairs, which makes the pushdown automaton nondeterministic. This was not done by mistake. Unlike finite automata, where the deterministic and nondeterministic models are equivalent in terms of recognition power, deterministic pushdown automata recognize *fewer* languages than nondeterministic pushdown automata. In the interest of full generality, then, we will take all of our pushdown automata to be nondeterministic.

Example 15. Consider a pushdown automaton $\mathcal{M}$ where $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $\Gamma = \{\perp, \mathtt{A}\}$, $q_0 = q_0$, $F = \{q_3\}$, and δ is specified by the following table:

Σ:	$\mathtt{a}$			$\mathtt{b}$			ϵ		
Γ:	$\perp$	$\mathtt{A}$	ϵ	$\perp$	$\mathtt{A}$	ϵ	$\perp$	$\mathtt{A}$	ϵ
q_0	—	—	—	—	—	—	—	—	$\{(q_1, \perp)\}$
q_1	—	—	$\{(q_1, \mathtt{A})\}$	—	$\{(q_2, \epsilon)\}$	—	—	—	—
q_2	—	—	—	—	$\{(q_2, \epsilon)\}$	—	$\{(q_3, \epsilon)\}$	—	—
q_3	—	—	—	—	—	—	—	—	—

In the transition function table, the top row indicates the input symbol being read and the second-from-top row indicates the symbol to be popped from the stack. Each entry of the table is an ordered pair where the first element is the state being transitioned to and the second element is the symbol being pushed to the stack.

The pushdown automaton $\mathcal{M}$ can be represented visually as follows:



Notice that each transition has a label of the form $a, B \mapsto C$; this means that, upon reading an input symbol a and popping a symbol B from the stack, the pushdown automaton pushes a symbol C to the stack.

Between states q_0 and q_1, the pushdown automaton pushes the symbol $\perp$ to the stack to act as the "bottom of stack" symbol. In state q_1, the pushdown automaton reads some number of $\mathtt{a}$s and pushes the same number of $\mathtt{A}$s to the stack. Between states q_1 and q_2, as well as in state q_2, the pushdown automaton reads some number of $\mathtt{b}$s and pops the same number of $\mathtt{A}$s from the stack. Finally, between states q_2 and q_3, the pushdown automaton pops the "bottom of stack" symbol $\perp$ from the stack.

[6] Strictly speaking, we do not require a special symbol to mark the bottom of the stack. Pushdown automata can accept either by final state or by empty stack, and as it turns out, the two methods of acceptance are equivalent. Here, we will follow the "accept by final state" convention.

After some observation, we can see that our pushdown automaton accepts all input words of the form $\mathtt{a}^n\mathtt{b}^n$ where $n \geq 1$.

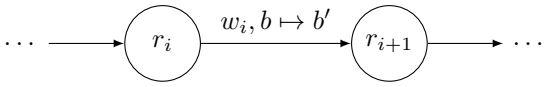
Let us now consider precisely what it means for a pushdown automaton to accept an input word. As we had with finite automata, one of the main conditions for acceptance is that there exists some sequence of states through the automaton where it begins reading its input word in an initial state and finishes reading in an accepting state. Since pushdown automata also come with a stack, though, we must account for the contents of the stack over the course of the computation. Specifically, we assume that the stack is empty at the beginning of the computation and, on each transition, the pushdown automaton can modify the top symbol of its stack appropriately.

Definition 16 (Accepting computation of a pushdown automaton)**.** Let $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a pushdown automaton, and let $w = w_0 w_1 \dots w_{n-1}$ be an input word of length n where $w_0, w_1, \dots, w_{n-1} \in \Sigma$. The pushdown automaton $\mathcal{M}$ accepts the input word w if there exists a sequence of states $r_0, r_1, \dots, r_n \in Q$ and a sequence of stack contents $s_0, s_1, \dots, s_n \in \Gamma^*$ satisfying the following conditions:

1. $r_0 = q_0$ and $s_0 = \epsilon$;
2. $(r_{i+1}, b') \in \delta(r_i, w_i, b)$ for all $0 \leq i \leq (n-1)$, where $s_i = bt$ and $s_{i+1} = b't$ for some $b, b' \in \Gamma \cup \{\epsilon\}$ and $t \in \Gamma^*$; and
3. $r_n \in F$.

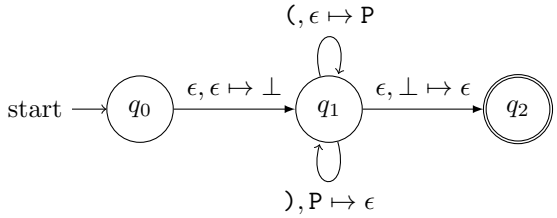
The second condition is rather notation-heavy, but the underlying idea describes exactly how a pushdown automaton transitions between states: starting in a state r_i with a symbol b at the top of the stack, the pushdown automaton reads an input symbol w_i and pops the symbol b from the stack. The transition function then sends the pushdown automaton to a state r_{i+1} and pushes the symbol b' to the stack.

Indeed, the second condition corresponds exactly to having the following transition in the pushdown automaton:



Lastly, pushdown automata recognize languages just as finite automata do, and the set of all input words accepted by a pushdown automaton is referred to as the language of that automaton. We denote the class of languages recognized by a pushdown automaton by PDA.

Example 17. Consider $L_{()}$, our language of balanced parentheses from earlier. Suppose $\Sigma = \{(,)\}$ and $\Gamma = \{\perp, \mathtt{P}\}$. A pushdown automaton recognizing this language is as follows:

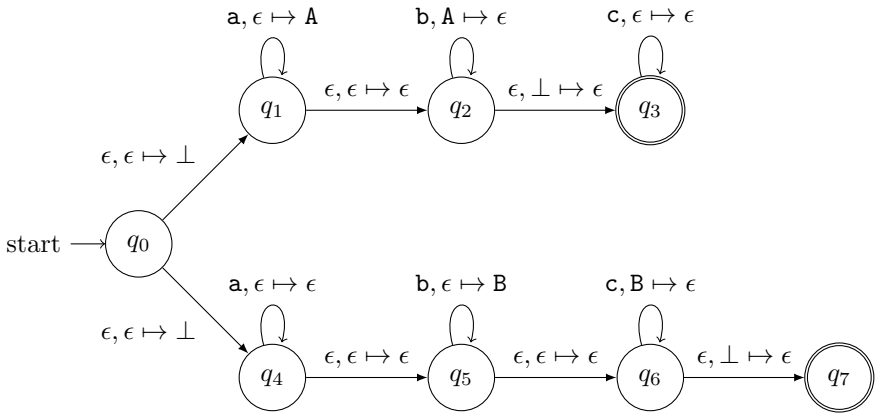


As the transitions show, after pushing the "bottom of stack" symbol $\perp$ to the stack, the pushdown automaton reads left and right parentheses. Every time a left parenthesis `(` is read, the pushdown automaton pushes a symbol `P` to the stack. Likewise, every time a right parenthesis `)` is read, the pushdown automaton pops a symbol `P` from the stack to account for some left parenthesis being matched.

Note that, if the input word contains more right parentheses than left parentheses, then the pushdown automaton will not be able to pop a symbol `P` from the stack. Similarly, if the input word contains more left parentheses than right parentheses, then it will not be able to pop the "bottom of stack" symbol $\perp$ from the stack. In either case, it becomes stuck in state q_1 and unable to accept the input word.

Example 18. Consider the language $L_{\text{twoequal}} = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } j = k\}$ over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. A pushdown automaton recognizing this language must have two "branches": one branch to handle the case where $i = j$, and one branch to handle the case where $j = k$. Since we don't know in advance which branch we will need to take, we can use the nondeterminism inherent in the pushdown automaton model.

A pushdown automaton recognizing this language would therefore look like the following, where the upper branch handles the case $i = j$ and the lower branch handles the case $j = k$:



2.1 PDA = CFG

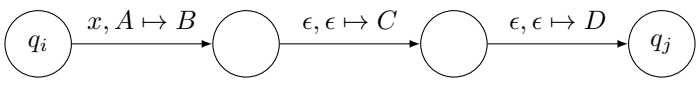
You may recall from our discussion of regular languages that we proved a couple of exciting results: a language L is regular if and only if there exists a finite automaton recognizing L, and a language L is regular if and only if there exists a regular expression matching words in L. These two results allowed us to establish Kleene's theorem, which brought together all of our different representations of regular languages.

Now that we're focusing on context-free languages, and now that we have two ways of representing context-free languages—namely, context-free grammars and pushdown automata—it would be nice to establish a connection between the two representations. This brings us to yet another exciting result, which will be the focus of this section. Since the overall proof is quite lengthy, we will split the proof of the main result into two parts.

Lemma 19. *Given a context-free grammar G generating a language $L(G)$, there exists a pushdown automaton $\mathcal{M}$ such that $L(\mathcal{M}) = L(G)$.*

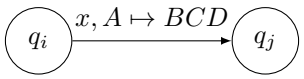
Proof. Suppose we are given a context-free grammar $G = (V, \Sigma_G, R, S)$. We will construct a pushdown automaton $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, F)$ that recognizes the language generated by G. Our pushdown automaton $\mathcal{M}$ will function as a *top-down parser*[7] of the input word w; that is, beginning with the start nonterminal S, $\mathcal{M}$ will repeatedly apply rules from R to check whether w can be generated via a leftmost derivation. If so, then $\mathcal{M}$ will accept w.

Note that, for the purposes of this proof, we will "condense" multiple transitions of our pushdown automaton into one transition; that is, if we have some sequence of transitions



[7] We could alternatively construct $\mathcal{M}$ to act as a *bottom-up parser*, where it applies rules backward starting from the input word w to see if the start nonterminal S can be reached. However, we will not discuss that construction here.

then we will depict this sequence of transitions as one single transition of the form



and we replace the symbol A on the stack with the symbols BCD, in that order from bottom to top.

We construct $\mathcal{M}$ in the following way:

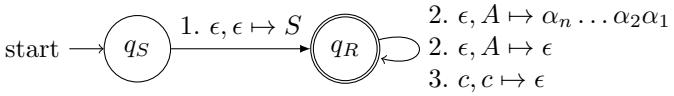
- The set of states is $Q = \{q_S, q_R\}$. The first state, q_S, corresponds to the point during the computation at which the context-free grammar G begins to generate the word. The second state, q_R, corresponds to the remainder of the computation where G applies rules from its rule set.
- The input alphabet is $\Sigma = \Sigma_G$. If $\mathcal{M}$ accepts its input word, then the word could be generated by G, and therefore it must consist of terminal symbols.
- The stack alphabet is $\Gamma = V \cup \Sigma_G$. We will use the stack of $\mathcal{M}$ to keep track of where we are in the leftmost derivation of the word.
- The initial state is $q_0 = q_S$.
- The final state is $F = \{q_R\}$.
- The transition function δ consists of three types of transitions:
  1. **Initial transition**: $\delta(q_S, \epsilon, \epsilon) = \{(q_R, S)\}$. This transition initializes the stack by pushing to it the start nonterminal S, and then moves to the state q_R for the remainder of the computation.
  2. **Nonterminal transition**: $\delta(q_R, \epsilon, A) = \{(q_R, \alpha_n \dots \alpha_2 \alpha_1)\}$ for each rule of the form $A \to \alpha_1 \alpha_2 \dots \alpha_n$, where $A \in V$ and $\alpha_i \in V \cup \Sigma_G$ for all i. Transitions of this form simulate the application of a given rule by popping the left-hand side (A) from the stack and pushing the right-hand side ($\alpha_1 \alpha_2 \dots \alpha_n$) to the stack in its place in reverse order. Pushing the symbols in reverse ensures that the next symbol we need to read (α_1) is at the top of the stack.

     Note that if $n = 0$, then the transition will be of the form $\delta(q_R, \epsilon, A) = \{(q_R, \epsilon)\}$.
  3. **Terminal transition**: $\delta(q_R, c, c) = \{(q_R, \epsilon)\}$ for each terminal symbol $c \in \Sigma_G$. Transitions of this form compare a terminal symbol on the stack to the current input word symbol. If the two symbols match, then the computation continues.

During the computation, after the initial transition is followed, $\mathcal{M}$ follows either nonterminal transitions or terminal transitions until its stack is empty or it runs out of input word symbols. If a nonterminal symbol A is at the top of the stack, $\mathcal{M}$ nondeterministically chooses one of the rules for A and follows the corresponding transition. If a terminal symbol c is at the top of the stack, $\mathcal{M}$ performs the comparison between input and stack symbol as described earlier.

By this construction, we can see that $\mathcal{M}$ finishes its computation with an empty stack and no input word symbols of w left to read whenever $S \Rightarrow^* w$, and so $\mathcal{M}$ accepts the input word w if w can be generated by the context-free grammar G. Therefore, $L(\mathcal{M}) = L(G)$ as desired. □

The pushdown automaton constructed in the proof of Lemma 19 can be illustrated as follows, where the number of each transition corresponds to its type:



Note that we don't require a "bottom of stack" symbol $\perp$ for this pushdown automaton, since we're only using the stack to keep track of where we are in the grammar's derivation.