1.4 Context-Free Languages

With our notion of a context-free grammar, it's easy for us to define a context-free language. Just like we defined regular languages in terms of regular operations or regular expressions, we have the following definition for context-free languages in terms of context-free grammars.

Definition 11 (Context-free language)**.** If some language L is generated by a context-free grammar, then L is context-free.

Thus, each of the languages in this lecture for which we constructed context-free grammars—the language of words $\mathtt{a}^n\mathtt{b}^n$, the language of balanced parentheses, the language of words $\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k$ where $i = j$ or $j = k$—are context-free languages. Our definition also tells us that a language is context-free if we can construct a context-free grammar generating words in that language. However, be careful: any such context-free grammar must generate *all* and *only all* the words in the given language.

As a shorthand, we denote the class of languages generated by a context-free grammar by CFG.

Example 12. Consider the language $L = \{\mathtt{a}^{2i}\mathtt{b}^i\mathtt{c}^{j+2} \mid i, j \geq 0\}$ over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Here, we can see that the counts of as and bs are related, while the number of cs is independent of the number of as and bs.

Let's construct a context-free grammar generating words in L. Evidently, we will need two kinds of rules: one rule will generate the as and bs together, while the other rule will generate the cs. We can use the start nonterminal to apply these rules in the correct order. Our context-free grammar will therefore look like the following:

$$\begin{aligned} S &\to UV \\ U &\to \mathtt{a}\mathtt{a}U\mathtt{b} \mid \epsilon \\ V &\to \mathtt{c}V \mid \mathtt{c}\mathtt{c} \end{aligned}$$

Let's now take a look at each rule in turn. The first rule, S, ensures that we apply the U rule before the V rule. This in turn ensures that all as and bs occur before the cs in the generated word. The second rule, U, either recursively produces two as and one b or produces the empty word. This ensures that we maintain the correct count of $2i$ as and i bs. Finally, the third rule, V, either recursively produces one c or produces the symbols cc. This ensures that we have exactly $j + 2$ cs in our generated word.

Example 13. Recall our context-free language $L_{\mathtt{a}=\mathtt{b}} = \{\mathtt{a}^n\mathtt{b}^n \mid n \geq 1\}$. Here, let's consider a more general language: $L_{\text{mixed}\mathtt{a}=\mathtt{b}} = \{w \in \{\mathtt{a}, \mathtt{b}\}^* \mid |w|_{\mathtt{a}} = |w|_{\mathtt{b}}\}$. Observe that the main difference with this language is that the order of as and bs no longer matters; we just need the same count of as and bs. Can we construct a context-free grammar for $L_{\text{mixed}\mathtt{a}=\mathtt{b}}$?

Since order no longer matters, we just need our context-free grammar to generate a pair of as and bs each time we add terminal symbols. For this, we can use essentially the same rule as we used in our context-free grammar for $L_{\mathtt{a}=\mathtt{b}}$: $S \to \mathtt{a}S\mathtt{b} \mid \mathtt{b}S\mathtt{a}$. We also need a rule that allows us to mix the order of as and bs; for instance, to place two as or two bs next to each other instead of strictly nesting pairs. For this, we can use a rule similar to one we included in our context-free grammar for $L_{()}$: $S \to SS$.

Thus, our context-free grammar will look like the following:

$$S \to SS \mid \mathtt{a}S\mathtt{b} \mid \mathtt{b}S\mathtt{a} \mid \epsilon$$

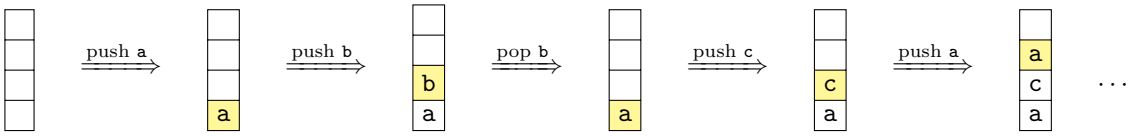
2 Pushdown Automata

When we first introduced finite automata as a computational model for regular languages, we emphasized the facts that finite automata have no memory, no storage, and no ability to return to a previously-read symbol. Naturally, these restrictions limited the kinds of languages the model is able to recognize, and we showed that such restrictions resulted in the model recognizing exactly the class of regular languages.

At the end of the previous lecture, we saw that there exist languages that are not regular, and therefore are not recognized by finite automata. We know now that the next "step" of our language hierarchy is the class of context-free languages. Thus, a new question arises: what kind of computational model is capable of recognizing context-free languages?

Since every context-free language is generated by a context-free grammar, and since we know that context-free grammars must "remember" which nonterminal and terminal symbols are being manipulated over the course of a derivation, any model of computation recognizing context-free languages must include a form of memory. What is the best form of memory to use in this situation? If we view a derivation as a parse tree, then the derivation progresses as we go deeper into the parse tree, and we can easily model the depth of a derivation using *stack* memory.

As a brief review, a stack is a data structure with two operations that manipulate data: *push* and *pop*. Pushing a symbol to a stack adds it to the top of the stack and moves all other symbols one position down in the stack. Conversely, popping a symbol from a stack removes it from the top and moves all other symbols one position up. As a result, a stack provides last-in-first-out (LIFO) storage.[4] We can view the symbol at the top of the stack at any time during a computation, but we cannot view any other symbols in the stack unless we pop the symbol currently at the top of the stack.



Since we're dealing with an abstract model of computation and not a real-world computer, we can make the assumption that our stack size is *unbounded*; that is, we can push as many symbols to the stack as we want without worrying about running out of space.

Now that we have our form of storage established, we can define our model of computation. At its core, this model is a finite automaton with a stack added to it. Since the automaton is now able to push symbols to a stack, we give it an appropriate name: a *pushdown automaton*.[5]

In addition to reading a symbol of its input word on a transition, a pushdown automaton can read from and write to the stack on the same transition. In order to perform this double duty, we specify two alphabets for a pushdown automaton: the *input alphabet*, which contains symbols used in the input word, and the *stack alphabet*, which contains symbols the pushdown automaton can use in its stack. This allows us to combine actions on the input word and actions on the stack in a single transition, without risking confusion over the meaning of any particular alphabet symbol. The transitions of a pushdown automaton may additionally use epsilon for either the input word action (i.e., not reading a symbol of the input word) or the stack action (i.e., not pushing to/popping from the stack). Just like we denoted a finite automaton's alphabet by Σ, we will use Σ to denote a pushdown automaton's input alphabet. Likewise, we will use Γ to denote the stack alphabet.

In order for our model of computation to use two alphabets at once, we must modify its transition function accordingly. Recall that a finite automaton (with epsilon transitions) transitions on a pair (q, a), where $q \in Q$ and $a \in \Sigma \cup \{\epsilon\}$. By comparison, a pushdown automaton transitions on a tuple (q, a, b), where $q \in Q$, $a \in \Sigma \cup \{\epsilon\}$, and $b \in \Gamma \cup \{\epsilon\}$. Thus, a pushdown automaton uses both the current symbol of its input word (or epsilon) as well as the top symbol of its stack (or epsilon) to determine to which state it will transition. After transitioning, the pushdown automaton will be in a possibly-different state, and it will have a possibly-different symbol at the top of its stack.

Lastly, a pushdown automaton has no mechanism for detecting whether or not its stack is empty. To avoid any potential issues, we often incorporate a special "bottom of stack" symbol $\perp$ into the transitions of a

[4] By comparison, a data structure like a queue would provide first-in-first-out (FIFO) storage.

[5] The name "pushdown automaton" doesn't specifically come from its ability to push symbols, but rather from an older term for a stack: a *pushdown store*.

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*.

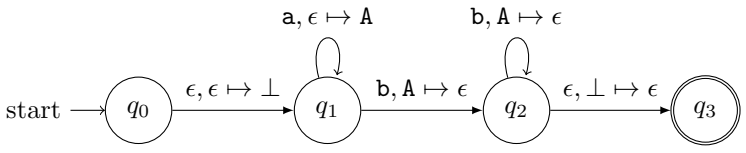
You may have noticed in our definition that the transition function maps to the power set of state/stack symbol pairs, which makes the pushdown automaton nondeterministic. This was not done by mistake. Unlike finite automata, where the deterministic and nondeterministic models are equivalent in terms 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 $n \geq 1$.

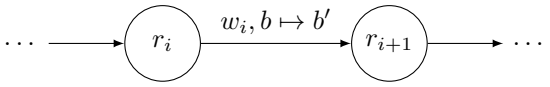
Let us now consider precisely what it means for a pushdown automaton to accept an input word. As we had with finite automata, one of the main conditions for acceptance is that there exists some sequence of states through the automaton where it begins reading its input word in an initial state and finishes 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. $r_n \in F$.

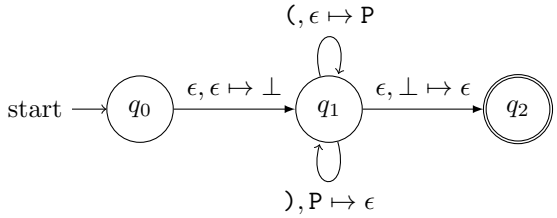
The second condition is rather notation-heavy, but the underlying idea describes exactly how a pushdown automaton transitions between states: starting in a state r_i with a symbol b at the top of the stack, the pushdown automaton reads an input symbol w_i and pops the symbol b from the stack. The transition 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 $\Sigma = \{\mathtt{(}, \mathtt{)}\}$ and $\Gamma = \{\perp, \mathtt{P}\}$. A pushdown automaton recognizing this language is as follows:

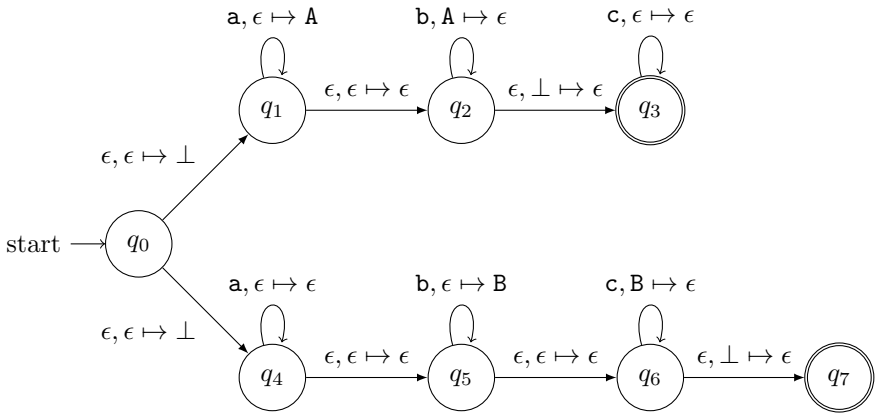


As the transitions show, after pushing the "bottom of stack" symbol $\perp$ to the stack, the pushdown automaton reads left and right parentheses. Every time a left parenthesis `(` is read, the pushdown automaton pushes a symbol `P` to the stack. Likewise, every time a right parenthesis `)` is read,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

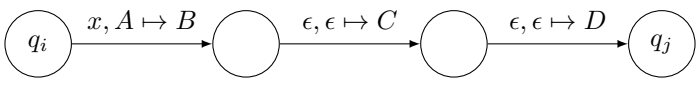
You may recall from our discussion of regular languages that we proved a couple of exciting results: a language L is regular if and only if there exists a finite automaton recognizing L, and a language L is regular if and only if there exists a regular expression matching words in L. These two results 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 $L(\mathcal{M}) = L(G)$.*

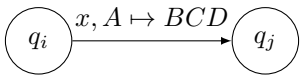
Proof. Suppose we are given a context-free grammar $G = (V, \Sigma_G, R, S)$. We will construct a pushdown automaton $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, F)$ that recognizes the language generated by G. Our pushdown automaton $\mathcal{M}$ will function as a *top-down parser*[7] of the input 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:

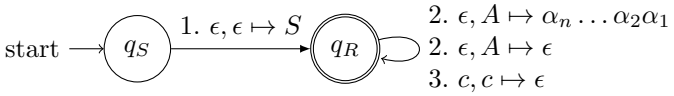
- The set of states is $Q = \{q_S, q_R\}$. The first state, q_S, corresponds to the point during the computation at which the context-free grammar G begins to generate the word. The second state, q_R, corresponds to the remainder of the computation where G applies rules from its rule set.
- The input 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:

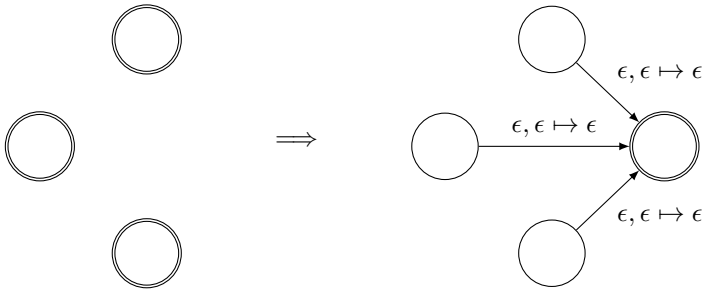


Note that we don't require a "bottom of stack" symbol $\perp$ for this pushdown automaton, since we're only using the stack to keep track of where we are in the grammar's derivation.

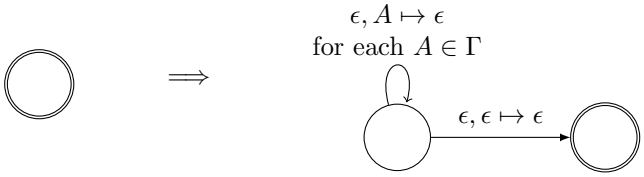
Now, we consider the other direction. In order to convert a pushdown automaton to a context-free grammar, we must first ensure the pushdown automaton has certain properties: namely, the pushdown automaton must have a single accepting state, it must empty its stack before accepting, and each transition of the pushdown automaton must either push to or pop from the stack, but not both simultaneously. Let us refer to a pushdown automaton with these properties as a *simplified pushdown automaton*.

Fortunately, it's easy to convert from a pushdown automaton to a simplified pushdown automaton.

- To ensure the pushdown automaton has a single accepting state, we make each original accepting state non-accepting and add epsilon transitions from those states to a new single accepting state.



- To ensure the pushdown automaton empties its stack before accepting, we add a state immediately before the accepting state that removes all symbols from the stack.



- To ensure that each transition of the pushdown automaton either pushes to or pops from the stack, but not both, we split each transition that both pushes and pops into two separate transitions.

Additionally, if we have an epsilon transition that neither pushes nor pops, then we replace it with two "dummy" transitions that push and then immediately pop the same stack symbol.

With a simplified pushdown automaton, we can now perform the conversion to a context-free grammar.

Lemma 20. *Given a simplified pushdown automaton $\mathcal{M}$ recognizing a language $L(\mathcal{M})$, there exists a context-free grammar G such that $L(G) = L(\mathcal{M})$.*

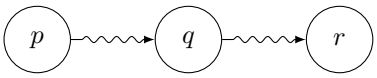
Proof. Suppose we are given a simplified pushdown automaton $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}})$. We will construct a context-free grammar $G = (V, \Sigma_G, R, S)$ that generates the language recognized by $\mathcal{M}$. For each pair of states p and q in $\mathcal{M}$, our grammar will include a rule A_{pq} that simulates the computation of $\mathcal{M}$ starting in state p with some stack contents and ending in state q with the same stack contents. (Note that the stack may be manipulated during this computation; we just ensure that the contents of the stack are the same at the beginning and the end.)

We construct G in the following way:

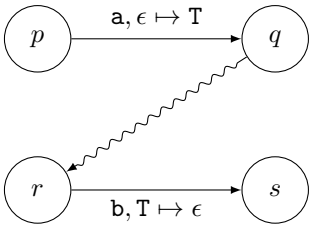
- The set of nonterminal symbols is $V = \{A_{pq} \mid p, q, \in Q\}$.
- The set of terminal symbols is $\Sigma_G = \Sigma$.

- The start nonterminal is $S = A_{q_0 q_{\text{accept}}}$ (i.e., the rule corresponding to the computation starting in state q_0 and ending in state q_{accept}).
- The set of rules, R, contains the following:
  - For each state $q \in Q$, add the rule $A_{qq} \to \epsilon$ to R.

  - For each triplet of states $p, q, r \in Q$, add the rule $A_{pr} \to A_{pq}A_{qr}$ to R.



  - For each quadruplet of states $p, q, r, s \in Q$, input symbols $\mathtt{a}, \mathtt{b} \in \Sigma \cup \{\epsilon\}$, and stack symbol $\mathtt{T} \in \Gamma$, if $(q, \mathtt{T}) \in \delta(p, \mathtt{a}, \epsilon)$ and $(s, \epsilon) \in \delta(r, \mathtt{b}, \mathtt{T})$, then add the rule $A_{ps} \to \mathtt{a}A_{qr}\mathtt{b}$ to R.



The first type of rule is a "dummy" rule that essentially corresponds to staying in the state q and adding nothing to the derivation. The second type of rule breaks down the overall computation into smaller components, taking into account intermediate states. Finally, the third type of rule adds terminal symbols to the derivation depending on the components of the overall computation.

With these rules, we can establish that the rule $A_{q_0 q_{\text{accept}}}$ generates a word w if and only if, starting in the state q_0 with an empty stack, the computation of $\mathcal{M}$ on w ends in the state q_{accept} also with an empty stack. Therefore, w is generated by the context-free grammar G if $\mathcal{M}$ accepts w, and $L(G) = L(\mathcal{M})$ as desired. □

Since we know by Definition 11 that a language is context-free if there exists a context-free grammar generating the language, we can combine the previous two lemmas to get the main result of this section.

Theorem 21. *A language A is context-free if and only if there exists a pushdown automaton $\mathcal{M}$ such that $L(\mathcal{M}) = A$.*

Proof. ($\Rightarrow$): Follows from Lemma 19.

($\Leftarrow$): Follows from Lemma 20. □

Let's now consider some examples of converting from a context-free grammar to a pushdown automaton and vice versa.

Example 22. Consider the following context-free grammar G, where $V = \{S, A\}$ and $\Sigma_G = \{\mathtt{0}, \mathtt{1}, \mathtt{\#}\}$:

$$S \to \mathtt{0}S\mathtt{1} \mid A$$
$$A \to \mathtt{\#}$$

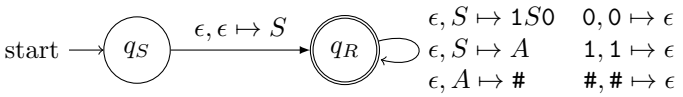
This grammar generates words of the form $\mathtt{0}^n\mathtt{\#}\mathtt{1}^n$, where $n \geq 0$.

We convert the context-free grammar G to a pushdown automaton $\mathcal{M}$. Take $Q = \{q_S, q_R\}$, $\Sigma = \Sigma_G$, $\Gamma = V \cup \Sigma_G$, $q_0 = q_S$, and $F = \{q_R\}$. Finally, add the following transitions to δ:

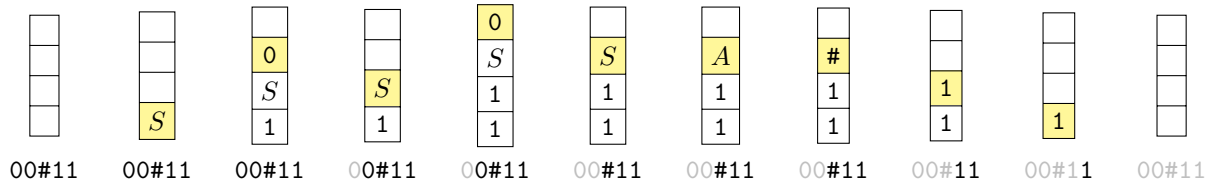
- $\delta(q_S, \epsilon, \epsilon) = \{(q_R, S)\}$. This initial transition pushes the start nonterminal S to the stack.

- $\delta(q_R, \epsilon, S) = \{(q_R, \texttt{1}S\texttt{0}), (q_R, A)\}$. These nonterminal transitions account for the S rules.
- $\delta(q_R, \epsilon, A) = \{(q_R, \texttt{\#})\}$. This nonterminal transition accounts for the A rule.
- $\delta(q_R, \texttt{0}, \texttt{0}) = \{(q_R, \epsilon)\}$, $\delta(q_R, \texttt{1}, \texttt{1}) = \{(q_R, \epsilon)\}$, and $\delta(q_R, \texttt{\#}, \texttt{\#}) = \{(q_R, \epsilon)\}$. These terminal transitions match the terminal symbols on the stack to the input word symbols.

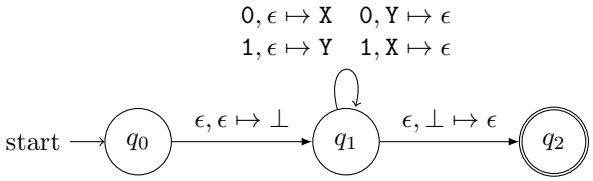
Diagrammatically, the pushdown automaton $\mathcal{M}$ looks like the following:



As an illustration of the computation of $\mathcal{M}$, let's look at the stack as $\mathcal{M}$ reads an example input word 00#11. We can see that G generates this word by the derivation $S \Rightarrow \texttt{0}S\texttt{1} \Rightarrow \texttt{00}S\texttt{11} \Rightarrow \texttt{00}A\texttt{11} \Rightarrow \texttt{00\#11}$.



Example 23. Consider the following pushdown automaton $\mathcal{M}$, where $\Sigma = \{\texttt{0}, \texttt{1}\}$ and $\Gamma = \{\texttt{X}, \texttt{Y}\}$:



This pushdown automaton recognizes words of the form $w\overline{w}$, where $\overline{w}$ is equal to w with 0s and 1s swapped.

We convert the pushdown automaton $\mathcal{M}$ to a context-free grammar G. Let $V = \{A_{00}, A_{01}, A_{02}, A_{11}, A_{12}, A_{22}\}$ and take $\Sigma_G = \Sigma$. We also take $S = A_{02}$, since q_0 is the initial state and q_2 is the accepting state of $\mathcal{M}$. Finally, we add the following rules to the rule set R:

- $A_{00} \to \epsilon$, $A_{11} \to \epsilon$, and $A_{22} \to \epsilon$. These rules are of the first type.
- $A_{01} \to A_{01}A_{11}$, $A_{02} \to A_{01}A_{12}$, $A_{11} \to A_{11}A_{11}$, and $A_{12} \to A_{11}A_{12}$. These rules are of the second type.
- $A_{11} \to \texttt{0}A_{11}\texttt{1}$, $A_{11} \to \texttt{1}A_{11}\texttt{0}$, and $A_{02} \to A_{11}$. These rules are of the third type.

As an illustration, let's see how G derives an example input word 001110. Beginning from the start nonterminal A_{02}, the derivation proceeds in the following way:

$$A_{02} \Rightarrow A_{11} \Rightarrow A_{11}A_{11} \Rightarrow \texttt{0}A_{11}\texttt{1}\,A_{11} \Rightarrow \texttt{0}\,\texttt{0}A_{11}\texttt{1}\,\texttt{1}A_{11} \Rightarrow \texttt{00}\,\epsilon\,\texttt{11}A_{11} \Rightarrow \texttt{0011}\,\texttt{1}A_{11}\texttt{0} \Rightarrow \texttt{00111}\,\epsilon\,\texttt{0} = \texttt{001110}.$$

Note that, as a consequence of the equivalence between context-free grammars and pushdown automata, we get an important corollary relating the class of context-free languages to our familiar class of regular languages.

Corollary 24. *Every regular language is also a context-free language.*

Proof. Every regular language is recognized by some finite automaton. Since a finite automaton is a pushdown automaton that does not use the stack, every regular language is also accepted by some pushdown automaton. Therefore, every regular language is context-free. □