2 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 can define a regular language to be a language represented by a regular expression, we can define a context-free language in terms of a context-free grammar.

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

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

The class of context-free languages is remarkably less restrictive than the class of regular languages and, as we've seen, context-freeness allows us to perform certain simple actions like counting or matching symbols. Let's now consider a couple of other examples of context-free languages and their grammars.

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:

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

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:

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

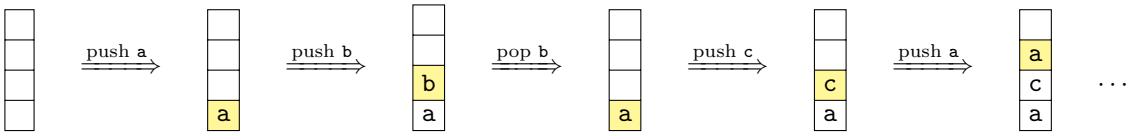
3 Pushdown Automata

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.[3] 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*.[4]

3.1 Definition

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 mixture of input and stack actions, 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 ϵ in place of either the input word action (i.e., when we don't read a symbol of the input word) or the stack action (i.e., when we don't push to/pop 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

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

[4] 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*.

any potential issues, we often incorporate a special "bottom of stack" symbol $\perp$ into the transitions of a 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.[5]

Having established all of the technical details, we can now formulate the 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*.

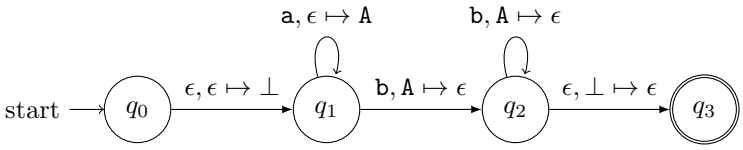
You may have noticed in our definition that the transition function maps to the power set of state/stack symbol pairs, which makes the pushdown automaton nondeterministic. This was not done by mistake. Unlike finite automata, where the deterministic and nondeterministic models are equivalent in terms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

[5] 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 $n \geq 1$.

3.2 Computations and Accepting Computations

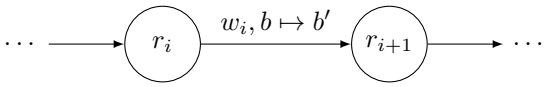
Let us now consider precisely what it means for a pushdown automaton to accept an input word. As we had with finite automata, one of the main conditions for acceptance is that there exists some sequence of states through the automaton where it begins reading its input word in an initial state and finishes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

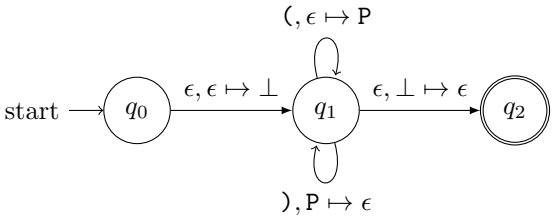
Indeed, the second condition corresponds exactly to having the following transition in the pushdown automaton:



3.3 Language of a Pushdown Automaton

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.