1.3 Normal Forms of Context-Free Grammars

Up to now, we've imposed no restrictions on the form of each rule in our context-free grammars. As long as each rule of our context-free grammar looked like $A \to \alpha$, where A is a nonterminal symbol and α is a combination of terminal and nonterminal symbols, we were happy.

However, computers (and, by extension, the people who program computers) like having structure. For instance, a compiler for a programming language usually incorporates a context-free grammar into its workflow at some point during the compilation of a program, and having a highly-structured grammar makes the compiler's job of determining which rule to apply both easier and faster.

Therefore, in some cases, we might like to transform a context-free grammar into a *normal form*; that is, to modify the grammar in such a way that each rule follows a canonical form or template.

The normal form we will focus on, the *Chomsky normal form*, was (as the name suggests) first studied by Noam Chomsky in the late 1950s as he attempted to develop a model of natural language using grammars. A grammar in Chomsky normal form is such that each rule either has two nonterminal symbols or one terminal symbol on the right-hand side.

Definition 8 (Chomsky normal form)**.** A context-free grammar is in Chomsky normal form if every rule in the grammar is of one of the two following forms:

1. $A \to BC$ for $A, B, C \in V$ with $B, C \neq S$; or
2. $A \to a$ for $A \in V$ and $a \in \Sigma$.

Additionally, we may allow the rule $S \to \epsilon$.

The main benefit of converting a grammar into Chomsky normal form comes in how we can represent and store derivations of words in memory. Since each rule derives either two nonterminal symbols or one terminal symbol, every parse tree will have a branching factor of either 2 or 1. This fact allows us to use efficient data structures for representing binary trees in memory, as well as to apply efficient algorithms to process parse trees and derivations. Moreover, the number of steps in a derivation using a grammar in Chomsky normal form is easy to bound: if the grammar generates a word w, then the derivation of w will contain $|w| - 1$ applications of a rule of the first form and $|w|$ applications of a rule of the second form.

Example 9. Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, and consider the following two grammars. Each grammar generates words consisting of one `b` surrounded on either side by zero or more `a`s. The grammar on the left is not in Chomsky normal form. The grammar on the right is in Chomsky normal form, and it is equivalent to the grammar on the left.

$$\begin{aligned} S &\to A\mathtt{b}A \\ A &\to A\mathtt{a} \mid \epsilon \end{aligned} \qquad\qquad \begin{aligned} S_0 &\to TA \mid BA \mid AB \mid \mathtt{b} \\ T &\to AB \\ A &\to AC \mid \mathtt{a} \\ B &\to \mathtt{b} \\ C &\to \mathtt{a} \end{aligned}$$

Every context-free grammar can be converted into a context-free grammar in Chomsky normal form, and the conversion process consists of four steps:

1. **START**: Replace the start nonterminal.

   Add a new start nonterminal S_0 together with a new rule $S_0 \to S$, where S is the start nonterminal of the original grammar. This guarantees that S_0 will not occur on the right-hand side of any rule.

2. **EPS**: Remove epsilon rules.

   Remove all rules of the form $A \to \epsilon$, where $A \neq S$. For each occurrence of A on the right-hand side of a rule in the original grammar, add a new rule with that occurrence of A removed from the right-hand side; that is, convert all rules of the form $X \to \alpha A \beta$ to $X \to \alpha\beta$, where α and β are combinations

of nonterminal and terminal symbols. Note that this applies to each *occurrence* of A, so a rule of the form $X \to \alpha A \beta A \gamma$ would contribute three new rules: $X \to \alpha \beta A \gamma$, $X \to \alpha A \beta \gamma$, and $X \to \alpha \beta \gamma$.

If there exists a rule of the form $X \to A$, then add a new rule of the form $X \to \epsilon$ unless we previously removed a rule of that form.

Repeat until all epsilon rules not involving the original start nonterminal are removed.

3. **UNIT**: Remove unit rules.

   Unit rules are rules of the form $A \to B$, where A and B are nonterminal symbols. Remove all rules of this form.

   If there exists a rule of the form $B \to \alpha$, where α is a combination of nonterminal and terminal symbols, then add a new rule $A \to \alpha$ unless we previously removed a unit rule of that form.

   Repeat until all unit rules are removed.

4. **BIN**: Remove rules with more than two nonterminal or terminal symbols on the right-hand side.

   Replace each rule of the form $A \to \alpha_1 \alpha_2 \dots \alpha_k$, where $k \geq 3$ and each α_i is either a nonterminal or terminal symbol, with a series of new rules $A \to \alpha_1 A_1$, $A_1 \to \alpha_2 A_2$, $\dots$, $A_{k-2} \to \alpha_{k-1} \alpha_k$. Each A_i is a new nonterminal symbol.

   If α_i is a terminal symbol in one of our new rules $A_{i-1} \to \alpha_i A_i$, then remove this rule, replace α_i with a new nonterminal symbol B_i, and add two new rules $B_i \to \alpha_i$ and $A_{i-1} \to B_i A_i$.

Example 10. Consider the following grammar not in Chomsky normal form:

$$\begin{aligned} S &\to ASB \\ A &\to \mathtt{a}AS \mid \mathtt{a} \mid \epsilon \\ B &\to S\mathtt{b}S \mid A \mid \mathtt{bb} \end{aligned}$$

We will convert this grammar to an equivalent grammar in Chomsky normal form.

1. **START**: Replace the start nonterminal.

   Adding a new start nonterminal and the rule $S_0 \to S$ gives us the following:

$$\begin{aligned} S_0 &\to S \\ S &\to ASB \\ A &\to \mathtt{a}AS \mid \mathtt{a} \mid \epsilon \\ B &\to S\mathtt{b}S \mid A \mid \mathtt{bb} \end{aligned}$$

2. **EPS**: Remove epsilon rules.

   The first epsilon rule we will remove is $A \to \epsilon$. Since A occurs on the right-hand side of two other rules ($S \to ASB$ and $A \to \mathtt{a}AS$), removing this rule means we must add two new rules $S \to SB$ and $A \to \mathtt{a}S$. Additionally, since we have a rule $B \to A$, we must add a new rule $B \to \epsilon$. This gives the following:

$$\begin{aligned} S_0 &\to S \\ S &\to ASB \mid SB \\ A &\to \mathtt{a}AS \mid \mathtt{a} \mid \mathtt{a}S \\ B &\to S\mathtt{b}S \mid A \mid \mathtt{bb} \mid \epsilon \end{aligned}$$

Next, we remove the epsilon rule $B \to \epsilon$ that we just added. Since B occurs on the right-hand side of two other rules ($S \to ASB$ and $S \to SB$), removing this rule means we must add two new rules $S \to AS$ and $S \to S$. This gives the following:

$$\begin{aligned} S_0 &\to S \\ S &\to ASB \mid SB \mid AS \mid S \\ A &\to \mathtt{a}AS \mid \mathtt{a} \mid \mathtt{a}S \\ B &\to S\mathtt{b}S \mid A \mid \mathtt{bb} \end{aligned}$$

3. **UNIT**: Remove unit rules.

   We first remove the unit rule $B \to A$. We do so by replacing A on the right-hand side of the rule with everything that can be produced by a rule of the form $A \to \alpha$:

$$\begin{aligned} S_0 &\to S \\ S &\to ASB \mid SB \mid AS \mid S \\ A &\to \mathtt{a}AS \mid \mathtt{a} \mid \mathtt{a}S \\ B &\to S\mathtt{b}S \mid \mathtt{bb} \mid \mathtt{a}AS \mid \mathtt{a} \mid \mathtt{a}S \end{aligned}$$

   Next, we remove the unit rule $S \to S$. This change is more straightforward, as we don't need to modify the S rule in any other way:

$$\begin{aligned} S_0 &\to S \\ S &\to ASB \mid SB \mid AS \\ A &\to \mathtt{a}AS \mid \mathtt{a} \mid \mathtt{a}S \\ B &\to S\mathtt{b}S \mid \mathtt{bb} \mid \mathtt{a}AS \mid \mathtt{a} \mid \mathtt{a}S \end{aligned}$$

   Lastly, we remove the unit rule $S_0 \to S$. Again, we replace S on the right-hand side of the rule with everything that can be produced by a rule of the form $S \to \alpha$:

$$\begin{aligned} S_0 &\to ASB \mid SB \mid AS \\ S &\to ASB \mid SB \mid AS \\ A &\to \mathtt{a}AS \mid \mathtt{a} \mid \mathtt{a}S \\ B &\to S\mathtt{b}S \mid \mathtt{bb} \mid \mathtt{a}AS \mid \mathtt{a} \mid \mathtt{a}S \end{aligned}$$

4. **BIN**: Remove rules with more than two nonterminal or terminal symbols on the right-hand side.

   Starting from the top, we replace the rule $S_0 \to ASB$ with the rules $S_0 \to AU_1$ and $U_1 \to SB$:

$$\begin{aligned} S_0 &\to AU_1 \mid SB \mid AS \\ S &\to ASB \mid SB \mid AS \\ A &\to \mathtt{a}AS \mid \mathtt{a} \mid \mathtt{a}S \\ B &\to S\mathtt{b}S \mid \mathtt{bb} \mid \mathtt{a}AS \mid \mathtt{a} \mid \mathtt{a}S \\ U_1 &\to SB \end{aligned}$$

Similarly, we replace the rule $S \to ASB$ with the rules $S \to AU_2$ and $U_2 \to SB$:

$$\begin{aligned}
S_0 &\to AU_1 \mid SB \mid AS \\
S &\to AU_2 \mid SB \mid AS \\
A &\to \mathtt{a}AS \mid \mathtt{a} \mid \mathtt{a}S \\
B &\to S\mathtt{b}S \mid \mathtt{bb} \mid \mathtt{a}AS \mid \mathtt{a} \mid \mathtt{a}S \\
U_1 &\to SB \\
U_2 &\to SB
\end{aligned}$$

Next, we replace the rule $A \to \mathtt{a}AS$ with the rules $A \to \mathtt{a}U_3$ and $U_3 \to AS$:

$$\begin{aligned}
S_0 &\to AU_1 \mid SB \mid AS \\
S &\to AU_2 \mid SB \mid AS \\
A &\to \mathtt{a}U_3 \mid \mathtt{a} \mid \mathtt{a}S \\
B &\to S\mathtt{b}S \mid \mathtt{bb} \mid \mathtt{a}AS \mid \mathtt{a} \mid \mathtt{a}S \\
U_1 &\to SB \\
U_2 &\to SB \\
U_3 &\to AS
\end{aligned}$$

Moving along, we replace the rules $B \to S\mathtt{b}S$ and $B \to \mathtt{a}AS$ with the set of rules $B \to SU_4$, $B \to \mathtt{a}U_5$, $U_4 \to \mathtt{b}S$, and $U_5 \to AS$:

$$\begin{aligned}
S_0 &\to AU_1 \mid SB \mid AS \\
S &\to AU_2 \mid SB \mid AS \\
A &\to \mathtt{a}U_3 \mid \mathtt{a} \mid \mathtt{a}S \\
B &\to SU_4 \mid \mathtt{bb} \mid \mathtt{a}U_5 \mid \mathtt{a} \mid \mathtt{a}S \\
U_1 &\to SB \\
U_2 &\to SB \\
U_3 &\to AS \\
U_4 &\to \mathtt{b}S \\
U_5 &\to AS
\end{aligned}$$

Lastly, we must replace all rules containing one terminal and one nonterminal symbol on the right-hand side. We introduce two new rules $V_1 \to \mathtt{a}$ and $V_2 \to \mathtt{b}$ and make the appropriate changes to other rules:

$$\begin{aligned}
S_0 &\to AU_1 \mid SB \mid AS \\
S &\to AU_2 \mid SB \mid AS \\
A &\to V_1U_3 \mid \mathtt{a} \mid V_1S \\
B &\to SU_4 \mid V_2V_2 \mid V_1U_5 \mid \mathtt{a} \mid V_1S \\
U_1 &\to SB \\
U_2 &\to SB \\
U_3 &\to AS \\
U_4 &\to V_2S \qquad\qquad V_1 \to \mathtt{a} \\
U_5 &\to AS \qquad\qquad V_2 \to \mathtt{b}
\end{aligned}$$

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 $\mathtt{a}$s and $\mathtt{b}$s are related, while the number of $\mathtt{c}$s is independent of the number of $\mathtt{a}$s and $\mathtt{b}$s.

Let's construct a context-free grammar generating words in L. Evidently, we will need two kinds of rules: one rule will generate the $\mathtt{a}$s and $\mathtt{b}$s together, while the other rule will generate the $\mathtt{c}$s. 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 $\mathtt{a}$s and $\mathtt{b}$s occur before the $\mathtt{c}$s in the generated word. The second rule, U, either recursively produces two $\mathtt{a}$s and one $\mathtt{b}$ or produces the empty word. This ensures that we maintain the correct count of $2i$ $\mathtt{a}$s and i $\mathtt{b}$s. Finally, the third rule, V, either recursively produces one $\mathtt{c}$ or produces the symbols $\mathtt{cc}$. This ensures that we have exactly $j + 2$ $\mathtt{c}$s 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 $\mathtt{a}$s and $\mathtt{b}$s no longer matters; we just need the same count of $\mathtt{a}$s and $\mathtt{b}$s. 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 $\mathtt{a}$s and $\mathtt{b}$s 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 $\mathtt{a}$s and $\mathtt{b}$s; for instance, to place two $\mathtt{a}$s or two $\mathtt{b}$s 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:

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

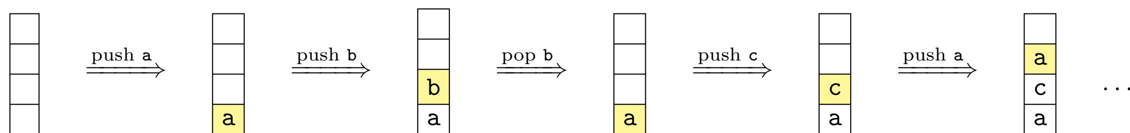
2 Pushdown Automata

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*.