5

Production Grammars

A *production grammar* (also known as a generative grammar, and which I will call simply grammar from now on) is a rewrite system that describes how to *generate* strings using a set of production rules.

Example 1: Here is a simple grammar given by two rules:

$$\begin{aligned} S &\to \mathtt{a}S\mathtt{b} \\ S &\to \epsilon \end{aligned} \tag{5.1}$$

These rules say that you can rewrite symbol S into $\mathtt{a}S\mathtt{b}$, or into the empty string. Here is a sequence of rewrites showing that these rules, starting with symbol S, can generate the string $\mathtt{aaabbb}$:

$$\begin{aligned} \underline{S} &\Longrightarrow \mathtt{a}\underline{S}\mathtt{b} \\ &\Longrightarrow \mathtt{aa}\underline{S}\mathtt{bb} \\ &\Longrightarrow \mathtt{aaa}\underline{S}\mathtt{bbb} \\ &\Longrightarrow \mathtt{aaabbb} \end{aligned}$$

(At every step, I indicated which symbol gets rewritten by underlining it.) It's not too difficult to see that such a grammar can generate all strings of the form $\mathtt{a}^n\mathtt{b}^n$ for any $n \geq 0$.

Example 2: Here is a slightly more complicated grammar, given by five rules:

$$\begin{aligned} S &\to TB \\ T &\to \mathtt{a}T\mathtt{b} \\ T &\to \epsilon \\ B &\to \mathtt{b}B \\ B &\to \epsilon \end{aligned} \tag{5.2}$$

Here is a sequence of rewrites showing that these rules, starting with symbol S, can generate the string aabbb:

$$\begin{aligned} \underline{S} &\Longrightarrow \underline{T}B \\ &\Longrightarrow \mathtt{a}\underline{T}\mathtt{b}B \\ &\Longrightarrow \mathtt{aa}\underline{T}\mathtt{bb}B \\ &\Longrightarrow \mathtt{aabb}\underline{B} \\ &\Longrightarrow \mathtt{aabbb}\underline{B} \\ &\Longrightarrow \mathtt{aabbb} \end{aligned}$$

Symbols S, T, B are intermediate (or nonterminal) symbols used during the rewrites, as opposed to a and b which are symbols in the strings that we care about generating. Again, it is not difficult to see that this grammar generates strings of the form $\mathtt{a}^n\mathtt{b}^m$ where $m \geq n \geq 0$.

All of the above grammars have the characteristic that the left-hand side of every rule consists of a single nonterminal symbol, that is, every rule is of the form $A \to \ldots$ for some nonterminal A. We call grammars made up of such rules *context-free grammars*, and they are an important class of grammars.

Example 3: Here is a grammar that is *not* context-free (also called unrestricted):

$$\begin{aligned} S &\to AB \\ B &\to XbB\mathtt{c} \\ B &\to \epsilon \\ \mathtt{b}X &\to X\mathtt{b} \\ A &\to AA \\ A &\to \epsilon \\ AX &\to \mathtt{a} \\ \mathtt{a}X &\to X\mathtt{a} \end{aligned} \tag{5.3}$$

Intuitively, these rules let us expand the initial A into a sequence of As, and the initial B into a sequence of b along with the same number of Xs on the left of the bs and cs on the right. Rules allow the Xs to "migrate" to the nearest As and interact with them to produce as.

Here is a sequence of rewrites showing how to generate aabbcc:

$$
\begin{aligned}
\underline{S} &\Longrightarrow A\underline{B} \\
&\Longrightarrow AX\texttt{b}\underline{B}\texttt{c} \\
&\Longrightarrow AX\texttt{b}X\texttt{b}\underline{B}\texttt{cc} \\
&\Longrightarrow AX\underline{\texttt{b}X}\texttt{bcc} \\
&\Longrightarrow \underline{A}XX\texttt{bbcc} \\
&\Longrightarrow A\underline{AX}X\texttt{bbcc} \\
&\Longrightarrow A\underline{\texttt{a}X}\texttt{bbcc} \\
&\Longrightarrow \underline{AX}\texttt{abbcc} \\
&\Longrightarrow \texttt{aabbcc}
\end{aligned}
$$

This grammar generates all strings of the form $\texttt{a}^n\texttt{b}^n\texttt{c}^n$ for $n \geq 0$.

Formal Definitions

Definition: A production grammar is a tuple $G = (N, \Sigma, R, S)$ where

- N is a finite set of nonterminal symbols;
- Σ is a finite set of terminal symbols;
- R is a finite set of rules, each of the form $w_1 \rightarrow w_2$ where $w_1, w_2 \in (N \cup \Sigma)^*$ and w_1 has at least one nonterminal symbol;
- $S \in N$ is a nonterminal symbol called the start symbol.

Rewriting using a grammar G is defined using a relation $\Longrightarrow_G$:

$$uw_1v \Longrightarrow_G uw_2v \text{ if } w_1 \rightarrow w_2 \in R$$

and generalizing to the multi-step rewrite relation $\Longrightarrow_G^*$ defined by taking $w_1 \Longrightarrow_G^* w_2$ if $w_1 = w_2$ or $\exists u_1, \ldots, u_k$ such that $w_1 \Longrightarrow_G u_1$, $u_1 \Longrightarrow_G u_2, \ldots,$ $u_{k-1} \Longrightarrow_G u_k$, and $u_k \Longrightarrow_G w_2$. We usually drop the G when it's clear from context.

The language $L(G)$ of grammar G is the set of all strings of terminals that can be generated from the start symbol of the grammar:

$$L(G) = \{w \in \Sigma^* \mid S \Longrightarrow_G^* w\}$$

Context-Free Languages

An important class of languages is the *context-free languages*. A language is *context-free* if there exists a context-free grammar that can generate it.

It is easy to see that regular languages are context-free. To show that, it suffices to show that to every deterministic finite automaton there exists a context-free grammar that generates the language accepted by the automaton.

Let $M = (Q, \Sigma, \delta, s, F)$ be a deterministic finite automaton. Construct the grammar $G_M = (N, \Sigma, R, S)$ by taking $N = Q$ and $S = s$, and by having one rule in R of the form

$$p \to aq$$

for every transition in M of the form $\delta(p, a) = q$, and one rule in R of the form

$$p \to \epsilon$$

for every $p \in F$.

Since $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$ is context-free by grammar (1) above but not regular, the class of context-free languages is a strictly larger class of languages than the regular languages.

It is a bit more painful to show that every context-free language is computable. (One way to show it is to show that every context-free language can be rewritten into an equivalent context-free grammar — where by equivalent we mean that it can generate the same language — in which all rules have the form $A \to \mathtt{a}$ or $A \to BC$. In other words, strings only grow during a derivation. We can now show that the language of the grammar is computable by exhibiting a nondeterministic Turing machine that repeatedly and nondeterministically expands all the nonterminals starting from the initial symbol, until either the desired string is produced — in which case it accepts — or until the length of the string is longer than the desired string — in which case it rejects.).

Not every computable language is context-free though. The language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$ is clearly computable — we can easily create a Turing machine using the scan-and-cross-out trick used in showing that $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$ is computable. It is not context-free, however, using a version of the Pumping Lemma for context-free languages. (Note that grammar (3) above was not context-free.)

Unrestricted Grammars and Turing Machines

What about unrestricted grammars? They turn out to be as expressive as Turing machines. More precisely, we can show that for every Turing-enumerable language, there is an unrestricted grammar that can generate it.[1]

The idea of the proof is simple: given a Turing machine, we construct an unrestricted grammar that can generate the strings accepted by the Turing machine by si mulating, through rewriting, the sequence of configurations the Turing machine goes through.

Let $M = (Q, \Gamma, \Sigma, \sqcup, \vdash, \delta, s, acc, rej)$ be a Turing machine.

Construct the grammar $G_M = (N, \Sigma, R, A_1)$ by taking

$$N = \{A, B, C\} \cup Q \cup \{\begin{bmatrix} a \\ X \end{bmatrix} \mid a \in \Sigma \cup \{\epsilon\}, X \in \Gamma\}$$

and the following rules:

$$\begin{aligned}
&A \to s \begin{bmatrix} \epsilon \\ \vdash \end{bmatrix} B \\
&B \to \begin{bmatrix} a \\ a \end{bmatrix} B \quad \text{(one such for every } a \in \Sigma) \\
&B \to C \\
&C \to \begin{bmatrix} \epsilon \\ \sqcup \end{bmatrix} C \\
&C \to \epsilon
\end{aligned}$$

as well as rules of the form

$$q \begin{bmatrix} a \\ X \end{bmatrix} \to \begin{bmatrix} a \\ Y \end{bmatrix} p$$

[1]The other direction, that any language generated by a grammar can be accepted by a Turing machine we can derive either from the Church-Turing thesis, or from the observation that we can find a nondeterministic Turing machine that simulates the generation of strings via the rewrite rules of the grammar.

for every $a \in \Sigma \cup \{\epsilon\}$ and every q, X, Y, p such that $\delta(q, X) = (p, Y, R)$, rules of the form

$$\begin{bmatrix} b \\ Z \end{bmatrix} q \begin{bmatrix} a \\ X \end{bmatrix} \to p \begin{bmatrix} b \\ Z \end{bmatrix} \begin{bmatrix} a \\ Y \end{bmatrix}$$

for every $a, b \in \Sigma \cup \{\epsilon\}$, every $Z \in \Gamma$, and every q, X, Y, p such that $\delta(q, X) = (p, Y, L)$, and rules of the form

$$\begin{aligned}
&\begin{bmatrix} a \\ X \end{bmatrix} acc \to acc\ a\ acc \\
&acc \begin{bmatrix} a \\ X \end{bmatrix} \to acc\ a\ acc \\
&\begin{bmatrix} \epsilon \\ X \end{bmatrix} acc \to acc \\
&acc \begin{bmatrix} \epsilon \\ X \end{bmatrix} \to acc \\
&acc \to \epsilon
\end{aligned}$$

for every $a \in \Sigma$ and $X \in \Gamma$.

Here's an example that shows how the grammar works. Let M be the Turing machine with states $Q = \{s, q, acc, rej\}$ that accepts all strings starting with an a. The transitions that are relevant are $\delta(s, \vdash) = (q, \vdash, R)$ and $\delta(q, \mathsf{a}) = (acc, \sqcup, R)$. (That last transition erases the first a to show what happens when the tape is changed during execution.) Every other transition goes to the reject state *rej*. The following sequence of rewrites for G_M shows how G_M can generate ab:

$$\begin{aligned}
A &\Longrightarrow s \begin{bmatrix} \epsilon \\ \vdash \end{bmatrix} B \\
&\Longrightarrow s \begin{bmatrix} \epsilon \\ \vdash \end{bmatrix} \begin{bmatrix} \mathsf{a} \\ \mathsf{a} \end{bmatrix} B \\
&\Longrightarrow s \begin{bmatrix} \epsilon \\ \vdash \end{bmatrix} \begin{bmatrix} \mathsf{a} \\ \mathsf{a} \end{bmatrix} \begin{bmatrix} \mathsf{b} \\ \mathsf{b} \end{bmatrix} B \\
&\Longrightarrow s \begin{bmatrix} \epsilon \\ \vdash \end{bmatrix} \begin{bmatrix} \mathsf{a} \\ \mathsf{a} \end{bmatrix} \begin{bmatrix} \mathsf{b} \\ \mathsf{b} \end{bmatrix} C \\
&\Longrightarrow s \begin{bmatrix} \epsilon \\ \vdash \end{bmatrix} \begin{bmatrix} \mathsf{a} \\ \mathsf{a} \end{bmatrix} \begin{bmatrix} \mathsf{b} \\ \mathsf{b} \end{bmatrix}
\end{aligned}$$

$$
\begin{aligned}
&\Longrightarrow \begin{bmatrix}\epsilon\\ \vdash\end{bmatrix} q \begin{bmatrix}\mathsf{a}\\ \mathsf{a}\end{bmatrix}\begin{bmatrix}\mathsf{b}\\ \mathsf{b}\end{bmatrix}\\
&\Longrightarrow \begin{bmatrix}\epsilon\\ \vdash\end{bmatrix}\begin{bmatrix}\mathsf{a}\\ \sqcup\end{bmatrix} acc \begin{bmatrix}\mathsf{b}\\ \mathsf{b}\end{bmatrix}\\
&\Longrightarrow \begin{bmatrix}\epsilon\\ \vdash\end{bmatrix} acc\ \mathsf{a}\ acc \begin{bmatrix}\mathsf{b}\\ \mathsf{b}\end{bmatrix}\\
&\Longrightarrow acc\ \mathsf{a}\ acc \begin{bmatrix}\mathsf{b}\\ \mathsf{b}\end{bmatrix}\\
&\Longrightarrow \mathsf{a}\ acc \begin{bmatrix}\mathsf{b}\\ \mathsf{b}\end{bmatrix}\\
&\Longrightarrow \mathsf{a}\ acc\ \mathsf{b}\ acc\\
&\Longrightarrow \mathsf{a}\ \mathsf{b}\ acc\\
&\Longrightarrow \mathsf{a}\ \mathsf{b}
\end{aligned}
$$

It's pretty easy to show that if M accepts w, then G_M can generate w. It's a bit tricker to show that if M cannot accept w, then G_M cannot generate w.

One consequence of this result is that it is noncomputable to determine if a grammar can generate a given string. If it were possible, then we could use that to solve the halting problem, as follows. Let M be a Turing machine and w an input. To decide if M halts on input w, first construct a Turing machine M' that accepts a string exactly when M halts on that string, by simulation. Then construct grammar $G_{M'}$, and ask whether G_M can generate w. If it can, then M' accepts w, and thus M halts on w. If it can't, then M' does not accept w, and M does not halt on w. Since this process can decide the halting problem, and we know that the halting problem is noncomputable, there cannot be a way to determine whether G_M can generate w.