Notes on Grammars

Foundations of Computer Science

March 24, 2016

A *generative grammar* (which I'll call simply a grammar) is a rewrite system that describes how to *generate* strings using a set of rules.

Here is a simple grammar given by two rules:

$$\begin{aligned} S &\to \mathtt{a}S\mathtt{b} \\ S &\to \epsilon \end{aligned} \tag{1}$$

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 $\mathtt{aaabbb}$:

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

(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 $n \geq 0$.

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

$$\begin{aligned} S &\to TB \\ T &\to \mathtt{a}T\mathtt{b} \\ T &\to \epsilon \\ B &\to \mathtt{b}B \\ B &\to \epsilon \end{aligned} \tag{2}$$

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

$$\begin{aligned} \underline{S} &\to \underline{T}B \\ &\to \mathtt{a}\underline{T}\mathtt{b}B \end{aligned}$$

$$\begin{aligned}
&\to \mathtt{aa}\underline{T}\mathtt{bb}B \\
&\to \mathtt{aabb}\underline{B} \\
&\to \mathtt{aabbb}\underline{B} \\
&\to \mathtt{aabbb}
\end{aligned}$$

Symbols S, T, B are intermediate (or nonterminal) symbols used during the rewrites, as opposed to $\mathtt{a}$ and $\mathtt{b}$ 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 each rule has a single nonterminal in it. We call grammars made up of such rules *context-free grammars*, and they are an important class of grammars.

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

$$\begin{aligned}
&S \to ABC \\
&B \to XbBX \\
&B \to \epsilon \\
&\mathtt{b}X \to X\mathtt{b} \\
&A \to AA \\
&A \to \epsilon \\
&AX \to \mathtt{a} \\
&\mathtt{a}X \to X\mathtt{a} \\
&C \to CC \\
&C \to \epsilon \\
&XC \to \mathtt{c} \\
&X\mathtt{c} \to \mathtt{c}X
\end{aligned} \tag{3}$$

Here is a sequence of rewrites showing how to generate $\mathtt{aabbcc}$:

$$\begin{aligned}
\underline{S} &\to A\underline{B}C \\
&\to AX\mathtt{b}\underline{B}XC \\
&\to AX\mathtt{b}X\mathtt{b}\underline{B}XXC \\
&\to AX\underline{\mathtt{b}X}\mathtt{b}XXC \\
&\to \underline{A}XX\mathtt{bb}XXC \\
&\to A\underline{AX}X\mathtt{bb}XXC \\
&\to A\underline{\mathtt{a}X}\mathtt{bb}XXC \\
&\to \underline{AX}\mathtt{abb}XXC \\
&\to \mathtt{aabb}XX\underline{C} \\
&\to \mathtt{aabb}X\underline{XC}C \\
&\to \mathtt{aabb}\underline{X\mathtt{c}}C
\end{aligned}$$

$$\rightarrow \texttt{aabbc}\underline{XC}$$
$$\rightarrow \texttt{aabbcc}$$

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

Definition: A generative 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 $\longrightarrow^1_G$:

$$uw_1v \longrightarrow^1_G uw_2v \text{ if } w_1 \rightarrow w_2 \in R$$

and generalizing to the multi-step rewrite relation $\longrightarrow^*_G$:

$$w_1 \longrightarrow^*_G w_2 \text{ if } w_1 = w_2 \text{ or } \exists u \text{ such that } w_1 \longrightarrow^1_G u \text{ and } u \longrightarrow^*_G w_2$$

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:

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

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

$$p \rightarrow aq$$

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

$$p \rightarrow \epsilon$$

for every $p \in F$.

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

It is a bit more painful to show that every context-free language is decidable, but it is true. The language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n\}$ is not context-free (we gave an unrestricted grammar for it, and we can show that there is no context-free grammar that can generate) but decidable, showing that context-free language is strictly contained within the decidable languages.

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 simulating, 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 $N = \{A_1, A_2, A_3\} \cup Q \cup ((\Sigma \cup \{\epsilon\}) \times \Gamma)$, and the following rules:

$$\begin{aligned}
&A_1 \rightarrow sA_2 \\
&A_2 \rightarrow [a, a]\ A_2 \quad \text{(for each } a \in \Sigma) \\
&A_2 \rightarrow A_3 \\
&A_3 \rightarrow [\epsilon, \sqcup]\ A_3 \\
&A_3 \rightarrow \epsilon
\end{aligned}$$

as well as rules

$$q\ [a, X] \rightarrow [a, Y]\ p$$

for every q, a, X, Y, p such that $\delta(q, X) = (p, Y, R)$,

$$[b, Z]\ q\ [a, X] \rightarrow p\ [b, Z]\ [a, Y]$$

for every q, a, b, X, Y, Z, p such that $\delta(q, X) = (p, Y, L)$, and rules

$$\begin{aligned}
&[a, X]\ acc \rightarrow acc\ a\ acc \\
&acc\ [a, X] \rightarrow acc\ a\ acc \\
&[\vdash, X]\ acc \rightarrow acc \\
&acc\ [\vdash, X] \rightarrow acc \\
&acc \rightarrow \epsilon
\end{aligned}$$

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

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

an $\mathtt{a}$. The transitions that are relevant are $\delta(s, \vdash) = (q, \vdash, R)$ and $\delta(q, \mathtt{a}) = (acc, \llcorner\!\lrcorner, R)$. (That last transition erases the first $\mathtt{a}$ 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 $\mathtt{ab}$:

$$
\begin{aligned}
A_1 &\to sA_2 \\
&\to s\ [\vdash, \vdash]\ A_2 \\
&\to s\ [\vdash, \vdash]\ [\mathtt{a}, \mathtt{a}]\ A_2 \\
&\to s\ [\vdash, \vdash]\ [\mathtt{a}, \mathtt{a}]\ [\mathtt{b}, \mathtt{b}]\ A_2 \\
&\to s\ [\vdash, \vdash]\ [\mathtt{a}, \mathtt{a}]\ [\mathtt{b}, \mathtt{b}]\ A_3 \\
&\to s\ [\vdash, \vdash]\ [\mathtt{a}, \mathtt{a}]\ [\mathtt{b}, \mathtt{b}] \\
&\to [\vdash, \vdash]\ q\ [\mathtt{a}, \mathtt{a}]\ [\mathtt{b}, \mathtt{b}] \\
&\to [\vdash, \vdash]\ [\mathtt{a}, \llcorner\!\lrcorner]\ acc\ [\mathtt{b}, \mathtt{b}] \\
&\to [\vdash, \vdash]\ acc\ \mathtt{a}\ acc\ [\mathtt{b}, \mathtt{b}] \\
&\to acc\ \mathtt{a}\ acc\ [\mathtt{b}, \mathtt{b}] \\
&\to \mathtt{a}\ acc\ [\mathtt{b}, \mathtt{b}] \\
&\to \mathtt{a}\ acc\ \mathtt{b}\ acc \\
&\to \mathtt{a}\ \mathtt{b}\ acc \\
&\to \mathtt{a}\ \mathtt{b}
\end{aligned}
$$

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 undecidable to determine if a grammar can generate a given string. (If it were possible, then we could use that to solve the halting problem. 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 undecidable, there cannot be a way to determine whether G_M can generate w.)