4. Tree grammars

Here's the plan:

- The main goal here is to introduce the idea of grammars that generate *trees* rather than strings.
- As a first step, I'll introduce a new, but extremely simple kind of string-generating grammar — a *strictly local* grammar — that is simpler than all of the others we've seen so far (i.e. simpler than FSAs).
- This will be useful because the most familiar way we're used to thinking about constructing trees corresponds to a *strictly local* tree grammar.
- From there we'll see clearly how to "scale up" to other kinds of tree grammars.

1 Strictly local (string) grammars

(1) A strictly local (string) grammar (SLG) is a four-tuple (Σ, I, F, Δ) where:
- Σ, the alphabet, is a finite set of symbols;
- $I \subseteq \Sigma$ is the set of initial symbols;
- $F \subseteq \Sigma$ is the set of ending symbols; and
- $\Delta \subseteq \Sigma \times \Sigma$ is the set of transitions.

There is "no state" — just the symbols. The "transitions" are just pairs of symbols, the *allowed bigrams*.

(2) An SLG (Σ, I, F, T) generates a string $x_1 x_2 \dots x_n \in \Sigma^*$ (for $n \geq 1$) iff:
- $x_1 \in I$, and
- for all $i \in \{2, \dots, n\}$, $(x_{i-1}, x_i) \in \Delta$, and
- $x_n \in F$.

Notice that by this definition, there is no way for an SLG to generate the empty string.[1]

Here's a very simple example SLG:

(3) $\Sigma = \{\texttt{s}, \texttt{t}, \texttt{n}, \texttt{i}, \texttt{a}\}$
$I = \{\texttt{s}, \texttt{t}\}$
$F = \{\texttt{i}, \texttt{a}\}$
$\Delta = \{(\texttt{s}, \texttt{t}), (\texttt{t}, \texttt{i}), (\texttt{t}, \texttt{a}), (\texttt{i}, \texttt{n}), (\texttt{a}, \texttt{n}), (\texttt{n}, \texttt{t})\}$

This grammar generates, for example, `ti`, `sta`, `tanti` and `stintanta`. (But not `tinata` or `stin`, for example.)

[1] This is non-standard. The usual definitions of strictly-local grammars in the literature include special start-of-string and end-of-string markers as components of bigrams, which makes it possible to generate the empty string. Also, to be more precise, what I'm describing here are 2-strictly-local grammars, which are a special case of the general idea of a k-strictly-local grammar which specifies allowable substrings of length k. See e.g. Jäger and Rogers (2012, p.1963).

2 From strings to trees

The game we've been playing so far follows this pattern:

(4) a. Identify an alphabet of symbols; call it Σ.
b. This determines a certain set of strings over this alphabet; usually written Σ^*.
c. Identify some subset of Σ^* as the stringset of interest; call this L, so $L \subseteq \Sigma^*$.
d. Ask what (string) grammar(s) can generate exactly that set of strings L.

Remember that step (4b) involves an important recursive definition:

(5) For any set Σ, we define Σ^* as the smallest set such that:
- $\varepsilon \in \Sigma^*$, and
- if $x \in \Sigma$ and $u \in \Sigma^*$ then $(x{:}u) \in \Sigma^*$.

Here's the new game, following the same pattern:[2]

(6) a. Identify an alphabet of symbols; call it Σ.
b. This determines a certain set of trees over this alphabet; usually written T_Σ.
c. Identify some subset of T_Σ as the treeset of interest; call this L, so $L \subseteq T_\Sigma$.
d. Ask what (tree) grammar(s) can generate exactly that set of trees L.

We have another important recursive definition for the set of trees over an alphabet:

(7) For any set Σ, we define T_Σ as the smallest set such that:
- if $x \in \Sigma$, then $x[] \in T_\Sigma$, and
- if $x \in \Sigma$ and $t_1, t_2, \ldots, t_k \in T_\Sigma$, then $x[t_1, t_2, \ldots, t_k] \in T_\Sigma$.

So for example, if $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, then the set T_Σ looks something like this:

(8) $T_\Sigma = \{\ \mathtt{a}[]\ ,\ \ \mathtt{b}[]\ ,\ \ \mathtt{c}[]\ ,\ \ \mathtt{a}[\mathtt{a}[]]\ ,\ \ \ldots,\ \ \mathtt{a}[\mathtt{b}[], \mathtt{b}[], \mathtt{c}[]]\ ,\ \ \ldots,\ \ \mathtt{b}[\mathtt{c}[\mathtt{a}[]], \mathtt{a}[\mathtt{b}[], \mathtt{b}[]]]\ ,\ \ \ldots\}$

But just as we allow ourselves to write $\mathtt{a}{:}(\mathtt{a}{:}(\mathtt{b}{:}\varepsilon))$ more conveniently as $\mathtt{aab}$ (and write $\mathtt{a}{:}\varepsilon$ as $\mathtt{a}$), we allow ourselves to write this as:

(9) $T_\Sigma = \left\{ \mathtt{a}\ ,\ \ \mathtt{b}\ ,\ \ \mathtt{c}\ ,\ \ \begin{array}{c}\mathtt{a}\\|\\\mathtt{a}\end{array}\ ,\ \ \ldots,\ \ \begin{array}{ccc}&\mathtt{a}&\\ \mathtt{b}&\mathtt{b}&\mathtt{c}\end{array}\ ,\ \ \ldots,\ \ \begin{array}{ccc}&\mathtt{b}&\\ \mathtt{c}&&\mathtt{a}\\|&&\\\mathtt{a}&\mathtt{b}&\mathtt{b}\end{array}\ ,\ \ \ldots \right\}$

3 Strictly local tree grammars

(10) A strictly local tree grammar (SLTG) is a four-tuple (Σ, F, Δ) where:
- Σ, the alphabet, is a finite set of symbols;
- $F \subseteq \Sigma$ is the set of ending symbols (or, maybe better, "root symbols"); and
- $\Delta \subseteq \Sigma^* \times \Sigma$ is the set of transitions, which must be finite.

[2] Good introductions to tree grammars include Thatcher (1973), Engelfriet (1975), Comon et al. (2007). Notice that it did not take long for mathematicians and computer scientists to generalize formal language theory from strings to trees, but (sadly?) it did take a while for the ideas to make their way back into linguistics.

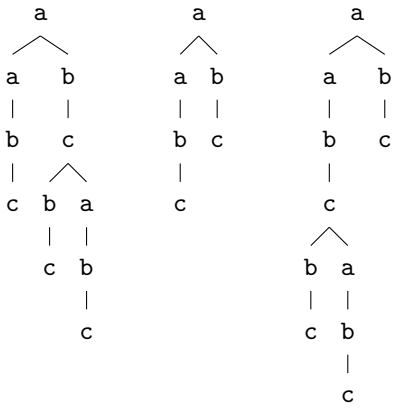
When specifying transitions, I'll switch to writing elements of Σ^* as "lists" rather than "strings", i.e. $[\mathtt{a}, \mathtt{a}, \mathtt{b}]$ rather than $\mathtt{aab}$. So the transition $([\mathtt{a}, \mathtt{a}, \mathtt{b}], \mathtt{c})$, for example, is a "tree bigram" in which $[\mathtt{a}, \mathtt{a}, \mathtt{b}]$ are the daughters of $\mathtt{c}$.

Here's an example SLTG:

(11) $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$
$F = \{\mathtt{a}\}$
$\Delta = \{([\mathtt{a}, \mathtt{b}], \mathtt{a}), ([\mathtt{b}, \mathtt{a}], \mathtt{c}), ([\mathtt{b}], \mathtt{a}), ([\mathtt{c}], \mathtt{b}), ([], \mathtt{c})\}$

Here are a few trees that are generated by this grammar:

(12)



Another SLTG that might look a bit more familiar (in a way):

(13) $\Sigma = \{\mathtt{s}, \mathtt{np}, \mathtt{vp}, \mathtt{v}, \mathtt{john}, \mathtt{mary}, \mathtt{met}, \mathtt{saw}\}$
$F = \{\mathtt{s}\}$
$\Delta = \{([\mathtt{np}, \mathtt{vp}], \mathtt{s}), ([\mathtt{v}, \mathtt{np}], \mathtt{vp}), ([\mathtt{john}], \mathtt{np}), ([\mathtt{mary}], \mathtt{np}), ([\mathtt{saw}], \mathtt{v}), ([\mathtt{met}], \mathtt{v}),$
$([], \mathtt{john}), ([], \mathtt{mary}), ([], \mathtt{met}), ([], \mathtt{saw})\}$

So there's a tight correspondence between:
- the trees generated by a strictly-local tree grammar (SLTG), and
- the trees that we typically use to represent the way strings are generated by a context-free (string) grammar (CFG).

4 Finite-state tree automata

As with strings, moving from strictly local to finite state allows us to enforce *long-distance dependencies* across tree structures.

(14) A finite-state tree automaton (FSTA) is a four-tuple (Q, Σ, F, Δ) where:
- Q is a finite set of states;
- Σ, the alphabet, is a finite set of symbols;
- $F \subseteq Q$ is the set of ending states (or "root states"); and
- $\Delta \subseteq Q^* \times \Sigma \times Q$ is the set of transitions, which must be finite.

For any FSTA $M = (Q, \Sigma, F, \Delta)$, under_M is a binary predicate relating trees to states:

(15)
$$\text{under}_M(x[])(q) = \Delta([], x, q)$$
$$\text{under}_M(x[t_1, \ldots, t_k])(q) = \bigvee_{q_1 \in Q} \ldots \bigvee_{q_k \in Q} \left[\Delta([q_1, \ldots, q_k], x, q) \wedge \text{under}_M(t_1)(q_1) \wedge \cdots \wedge \text{under}_M(t_k)(q_k)\right]$$

And $\mathcal{L}(M)$ is a subset of T_Σ:

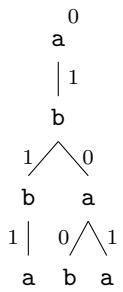
(16)
$$t \in \mathcal{L}(M) \iff \bigvee_{q \in Q} \left[\text{under}_M(t)(q) \wedge F(q)\right]$$

4.1 A "binary counting" example FSTA

Here's a grammar that cares about odd and even numbers ...

(17)
$$\begin{aligned} Q &= \{0, 1\} \\ \Sigma &= \{\texttt{a}, \texttt{b}\} \\ F &= \{0\} \\ \Delta = \{ &([0,0], \texttt{a}, 1), ([0,0], \texttt{b}, 0), \\ &([0,1], \texttt{a}, 0), ([0,1], \texttt{b}, 1), \\ &([1,0], \texttt{a}, 0), ([1,0], \texttt{b}, 1), \\ &([1,1], \texttt{a}, 1), ([1,1], \texttt{b}, 0), \\ &([0], \texttt{a}, 1), ([0], \texttt{b}, 0), \\ &([1], \texttt{a}, 0), ([1], \texttt{b}, 1), \\ &([], \texttt{a}, 1), ([], \texttt{b}, 0), \\ \} & \end{aligned}$$

(18)

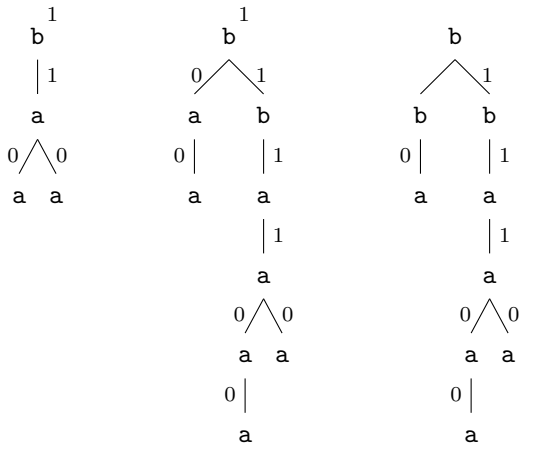


4.2 Another abstract example

Here's a grammar that requires that every b dominates a binary-branching a.

(19)
$$\begin{aligned} Q &= \{0, 1\} \\ \Sigma &= \{\texttt{a}, \texttt{b}\} \\ F &= \{0, 1\} \\ \Delta = \{ &([0,0], \texttt{a}, 1), \\ &([0,1], \texttt{a}, 1), ([0,1], \texttt{b}, 1), \\ &([1,0], \texttt{a}, 1), ([1,0], \texttt{b}, 1), \\ &([1,1], \texttt{a}, 1), ([1,1], \texttt{b}, 1), \\ &([0], \texttt{a}, 0), \\ &([1], \texttt{a}, 1), ([1], \texttt{b}, 1), \\ &([], \texttt{a}, 0), \\ \} & \end{aligned}$$

(20)



4.3 A more linguistic example

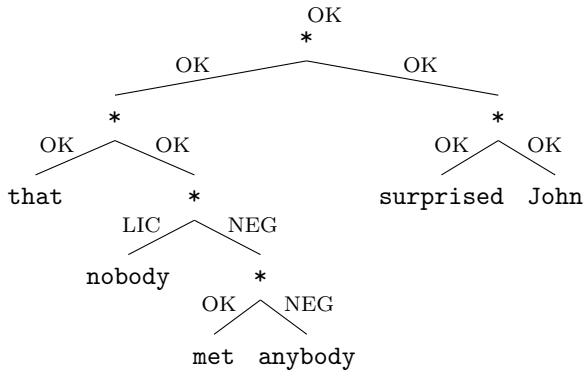
We can use an FSTA to encode a simple version of the NPI-licensing constraint: an NPI such as `anybody` or `ever` must be c-commanded by a licensor such as `not` or `nobody`.

(21)
a. Nobody met anybody
b. * John met anybody
c. Nobody thinks that John met anybody
d. The fact that nobody met anybody surprised John
e. * The fact that nobody met John surprised anybody

The alphabet Σ is the set of English words, plus the additional symbol `*`.

(22)
$Q = \{\text{OK}, \text{NEG}, \text{LIC}\}$
$\Sigma = \{\texttt{*}, \texttt{John}, \texttt{thinks}, \texttt{surprised}, \texttt{met}, \texttt{anybody}, \texttt{ever}, \texttt{not}, \texttt{nobody}, \dots\}$
$F = \{\text{OK}, \text{LIC}\}$
$\Delta =$ { ([NEG, NEG], `*`, NEG), ([], `anybody`, NEG),
([OK, NEG], `*`, NEG), ([], `ever`, NEG),
([NEG, OK], `*`, NEG), ([], `not`, LIC),
([OK, OK], `*`, OK), ([], `nobody`, LIC),
([LIC, NEG], `*`, OK), ([], x, OK) for any other $x \in \Sigma - \{\texttt{*}\}$,
([LIC, OK], `*`, OK),
([OK, LIC], `*`, OK),
([LIC, LIC], `*`, OK) }

(23)

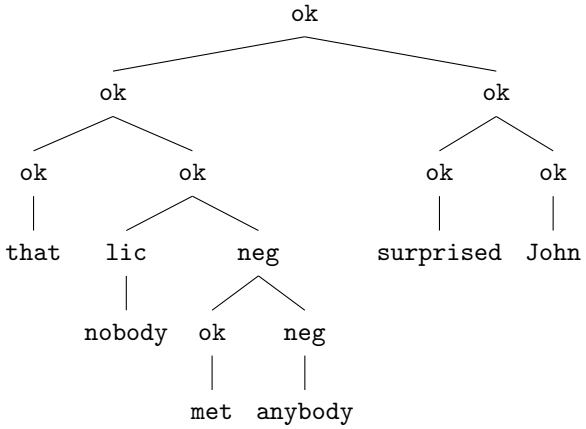


Notice that the pattern of two-daughter transitions and zero-daughter transitions in this grammar ensures that the generated trees will contain only (i) binary nodes with the symbol $*$, and (ii) leaf nodes with other symbols.

5 So what do FSTAs gain for us?

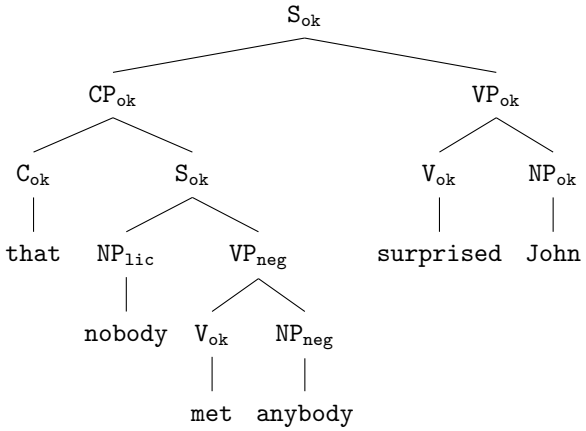
But wait a minute — how different is (23) from what we could already do with a strictly-local tree grammar?

(24) $\Sigma = \{\texttt{ok}, \texttt{lic}, \texttt{neg}, \texttt{that}, \texttt{met}, \texttt{surprised}, \texttt{nobody}, \texttt{anybody}, \texttt{ever}, \dots\}$
$\Delta = \{([\texttt{lic}, \texttt{neg}], \texttt{ok}), ([\texttt{ok}, \texttt{neg}], \texttt{neg}), \dots, ([\texttt{nobody}], \texttt{lic}), ([\texttt{anybody}], \texttt{neg}), ([\texttt{met}], \texttt{ok}), \dots ([], \texttt{met}), \dots\}$



Of course we're used to seeing other things as the labels for those internal nodes, and using *those* labels to enforce certain *other* requirements (e.g. the requirement that an S is made up of an NP and a VP). But we can just bundle all that information together.

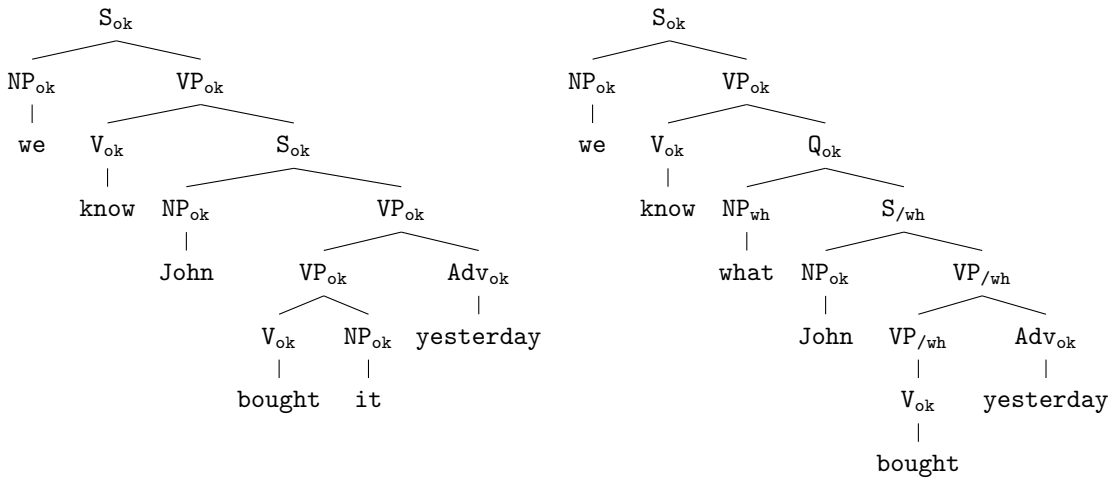
(25) $\Sigma = \{\texttt{S}_{\texttt{ok}}, \texttt{S}_{\texttt{lic}}, \texttt{S}_{\texttt{neg}}, \texttt{VP}_{\texttt{ok}}, \texttt{VP}_{\texttt{lic}}, \texttt{VP}_{\texttt{neg}}, \dots, \texttt{that}, \texttt{met}, \texttt{surprised}, \texttt{nobody}, \texttt{anybody}, \texttt{ever}, \dots\}$
$\Delta = \{([\texttt{NP}_{\texttt{ok}}, \texttt{VP}_{\texttt{ok}}], \texttt{S}_{\texttt{ok}}), ([\texttt{NP}_{\texttt{lic}}, \texttt{VP}_{\texttt{neg}}], \texttt{S}_{\texttt{ok}}), \dots\}$



We can even use a similar trick for "movement"!

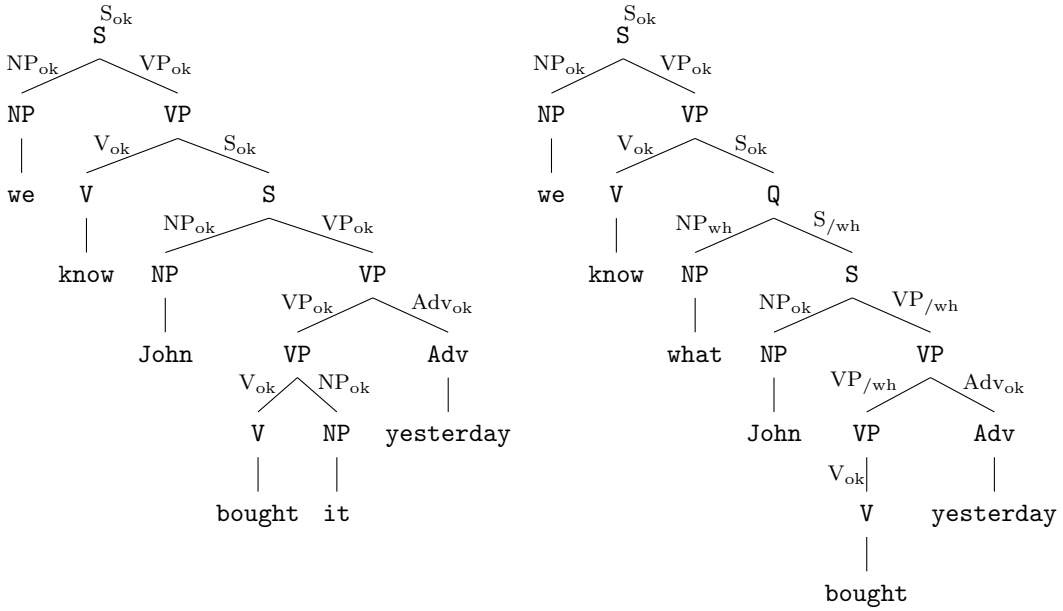
(26) a. We know John bought it yesterday
b. * We know John bought yesterday
c. * We know what John bought it yesterday
d. We know what John bought yesterday

(27)



If we felt that the trees in (27) were missing certain important similarities, we could keep the node labels looking more familiar and use the states of an FSTA to do the extra work.

(28)



But ultimately, the key question is **what information needs to be tracked in order for the grammar to know when two subtrees are intersubstitutable**, whether this is implemented via node labels, states, or a combination of the two.

We've now arrived at the same class of string languages via four different routes:

- the string languages that can be generated by a pushdown (string) automaton
- the string languages that can be generated by a context-free (string) grammar
- the yields of the tree languages that can be generated by a strictly-local tree grammar
- the yields of the tree languages that can be generated by a finite-state tree grammar

(The *yield* of a tree is the string you get by "reading along the leaves" in order in the familiar way; the yield of a tree language is the set of yields of the trees in the language.)

References

Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., and Tommasi, M. (2007). Tree automata: Techniques and applications. Available from: `http://www.grappa.univ-lille3.fr/tata`. release October, 12th 2007.

Engelfriet, J. (1975). Tree automata and tree grammars. `https://arxiv.org/abs/1510.02036`.

Jäger, G. and Rogers, J. (2012). Formal language theory: refining the Chomsky hierarchy. *Philosophical Transaction of the Royal Society B*, 367:1956–1970.

Thatcher, J. W. (1973). Tree automata: An informal survey. In Aho, A. V., editor, *Currents in the Theory of Computing*, pages 143–172. Prentice-Hall.