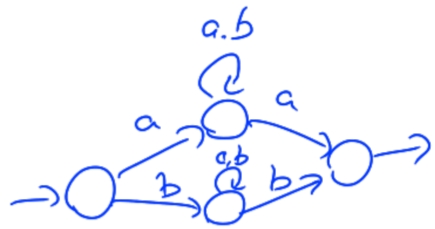
8. Tree grammars



1 Review: Stringsets and string grammars

The kind of thing we've done with strings many times now follows this pattern:

(1) 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.

first sym = last sym

aabbababbbaa

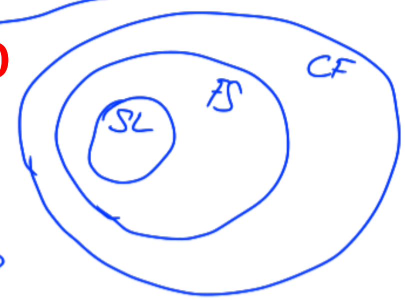
CFG ✓

FSA ✓

SLG ×

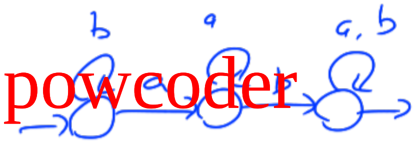
Remember that step (1b) involves an important recursive definition:

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



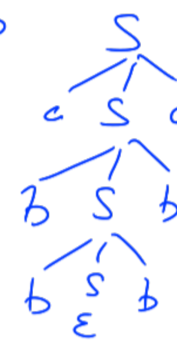
So if $\Sigma = \{a, b\}$, then Σ^* contains things like $a{:}(a{:}(b{:}\epsilon))$, which we abbreviate as 'aab'.

Then, in step (1c), we identify some stringsets that we might be interested in:



(3) a. $L_1 = \{w \mid w \in \Sigma^*$ and every 'a' is immediately followed by a 'b'$\}$
 b. $L_2 = \{w \mid w \in \Sigma^*$ and $w \neq \epsilon$ and the first and last symbols of w are the same$\}$
 c. $L_3 = \{w \mid w \in \Sigma^*$ and the number of occurrences of 'a' in w is even$\}$
 d. $L_4 = \{w \mid w \in \Sigma^*$ and w contains an 'a' that is followed (not necessarily immediately) by a 'b'$\}$
 e. $L_5 = \{a^n b^n \mid n \in \mathbb{N}$ and $n \geq 0\}$
 f. $L_6 = \{w + w^R \mid w \in \Sigma^*\}$ (where w^R is the reverse of the string w)
 g. $L_7 = \{w + w \mid w \in \Sigma^*\}$ CFG ×

$S \to aSa$
$S \to bSb$
$S \to \epsilon$



And for each such stringset L, we can ask (step (1d)) what kinds of grammars can generate exactly L.

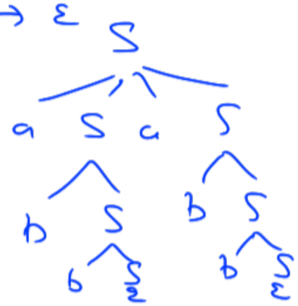
2 Generalizing: Treesets and tree grammars

Things will follow an analogous pattern here:

$S \to aSaS$
$S \to bSbS$
$S \to \epsilon$

(4) 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.

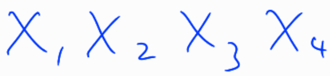
abbabb



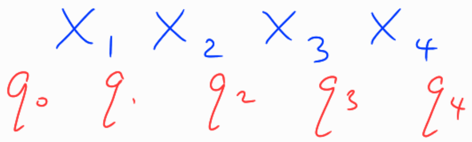
Σ^* T_Σ

$L \subseteq \Sigma^*$ $L \subseteq T_\Sigma$

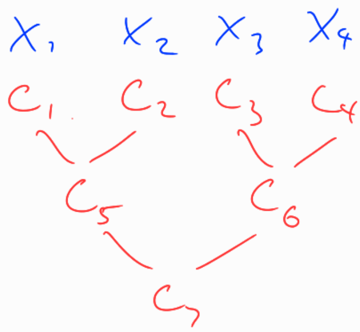
SLG



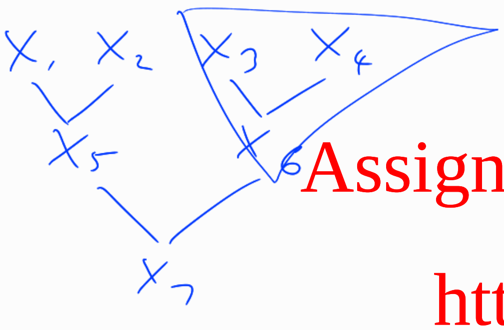
FSA



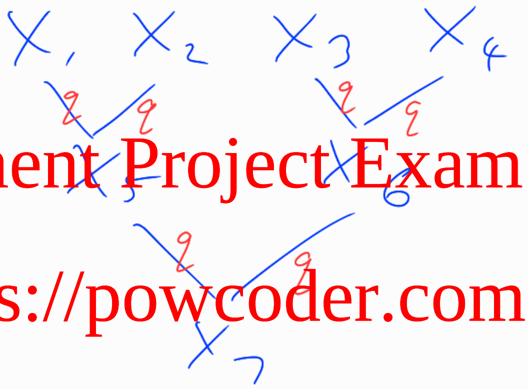
CFG

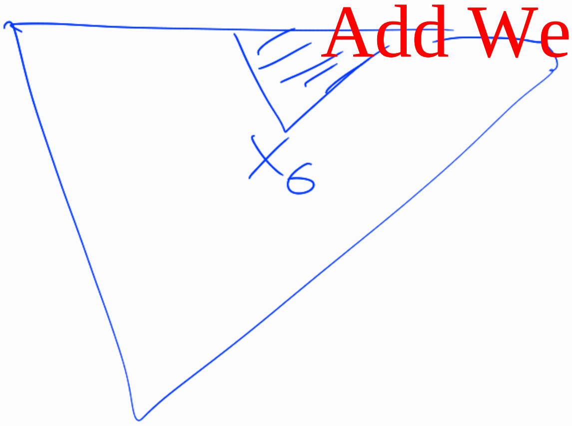


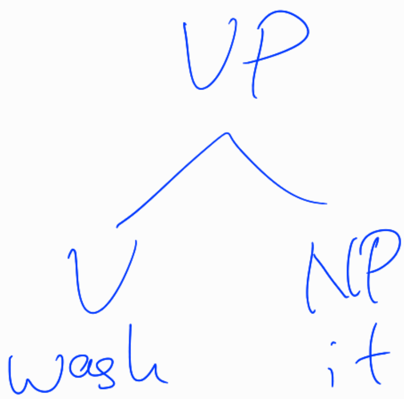
SLTG

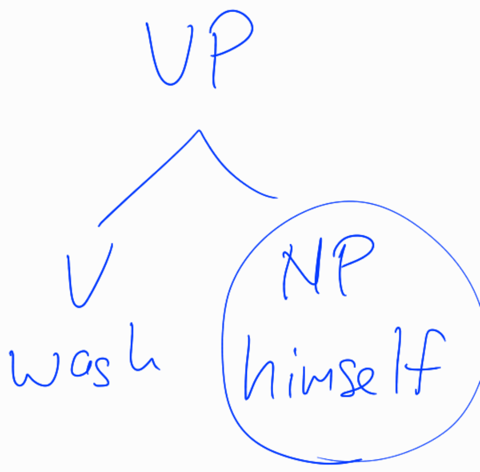


FSTG









2.1 The set of trees over an alphabet

$x[] \in T_\Sigma$

(5) For any set Σ, we define T_Σ as the smallest set such that:
- if $x \in \Sigma$, then $x[] \in T_\Sigma$, and $k=0$
- if $x \in \Sigma$ and $t_1, t_2, \ldots, t_k \in T_\Sigma$, then $x[t_1, t_2, \ldots, t_k] \in T_\Sigma$. $k \geq 1$

data Tree a = Node a [Tree a]

~~data Tree a = Leaf a | NonLeaf a [Tree a]~~

~~Leaf 1 :: Tree Int~~

~~NonLeaf 1 [] :: Tree Int~~

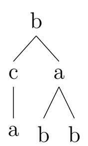
Those square brackets in this definition are analogous to the colon in the definition of Σ^*. The colon makes strings out of symbols, and the square brackets make trees out of symbols. (These pieces of punctuation correspond to *constructors* in Haskell.)

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

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

But just as we allow ourselves to write a:(a:(b:ϵ)) more conveniently as 'aab', we allow ourselves to write b[c[a[]], a[b[], b[]]] more conveniently as:

(7)



$\in T_\Sigma^2$ $\notin T_\Sigma^1$

$\in T_\Sigma^3$

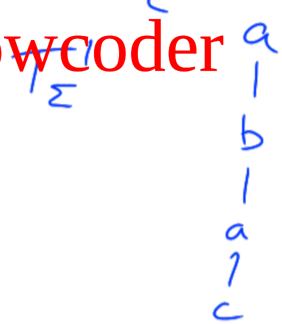
Also it's sometimes convenient to leave off empty pairs of brackets, so instead of b[c[a[]], a[b[], b[]]] we sometimes write b[c[a], a[b, b]].

One more definition is useful:

(8) For any set Σ and any natural number n, we define T_Σ^n as the set of all trees in T_Σ in which every node has at most n daughters.

$\Sigma = \{a, b, c\}$

So the tree in (7), for example, is a member of T_Σ^2 and is also a member of T_Σ^3, but is not a member of T_Σ^1.

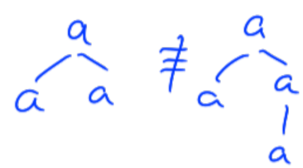


The largest number of daughters of any node in a tree is sometimes called the tree's *branching degree*. So T_Σ^n is the set of all trees in T_Σ with branching degree less than or equal to n. The branching degree of the tree in (7) is 2.

2.2 Subsets of T_Σ ("treesets")

Using the alphabet $\Sigma = \{\text{a}, \text{b}\}$, here are some treesets we might be interested in:

(9)
a. $L_1 = \{t \in T_\Sigma^2 \mid$ the number of occurrences of 'a' in t is even$\}$
b. $L_2 = \{t \in T_\Sigma^2 \mid$ every 'b' in t dominates a binary-branching 'a'$\}$
c. $L_3 = \{t \in T_\Sigma^2 \mid t$ contains a binary-branching 'a' whose left daughter subtree contains an 'a' and whose right daughter subtree contains a 'b'$\}$
d. $L_4 = \{t \in T_\Sigma^2 \mid t$ contains equal numbers of occurrences of 'a' and 'b'$\}$



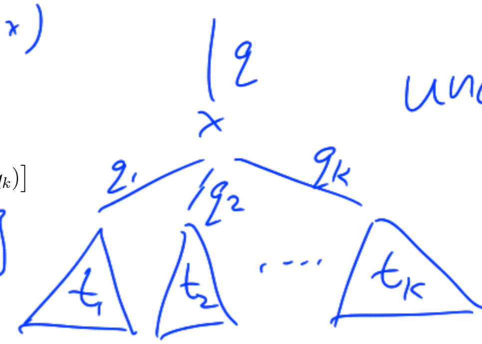
2.3 One kind of tree grammar

(10) A (bottom-up) 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; and
- $\Delta \subseteq Q^* \times \Sigma \times Q$ is the set of transitions, which must be finite.

(q_1, x, q_2) $((q_1, q_2, q_3), x, q_4)$

$\text{inside}(x)(n) = R(n, x)$

$\text{under}_G(t)(q)$



For any FSTA $G = (Q, \Sigma, F, \Delta)$, under_G is a function from $T_\Sigma \times Q$ to booleans:

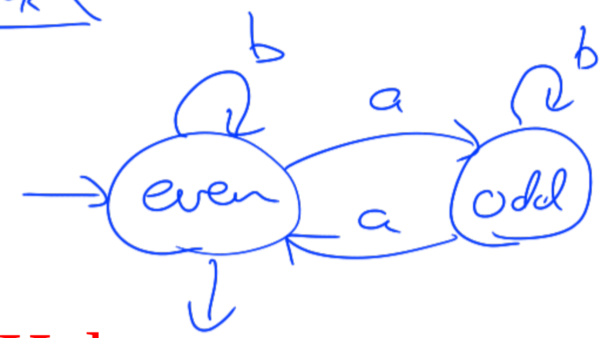
(11)
$$\text{under}_G(x[])(q) = \Delta([], x, q)$$
$$\text{under}_G(x[t_1, \ldots, t_k])(q) = \bigvee_{q_1 \in Q} \cdots \bigvee_{q_k \in Q} [\Delta([q_1, \ldots, q_k], x, q) \wedge \text{under}_G(t_1)(q_1) \wedge \cdots \wedge \text{under}_G(t_k)(q_k)]$$

$$\bigvee_{q_1} [\Delta([q_1], x, q) \wedge \text{under}(t_1)(q_1)]$$

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

(12)
$$t \in \mathcal{L}(G) \iff \bigvee_{q \in Q} [\text{under}_G(t)(q) \wedge F(q)]$$

As usual, slot in your favourite semiring as desired!



2.4 Examples

2.4.1 Even/odd

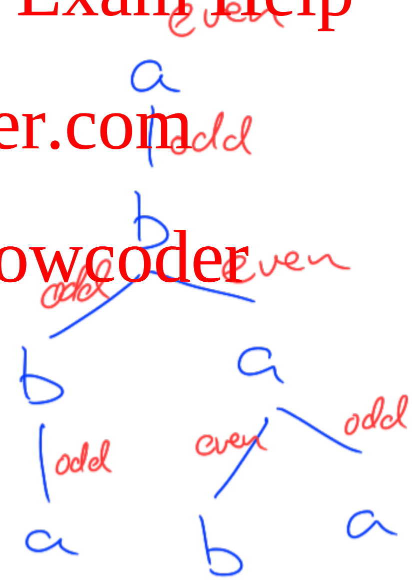
The FSTA G_1 in (13) generates the treeset L_1 from (9) above (requiring an even number of 'a's).

(13) $G_1 = (\{\text{even}, \text{odd}\}, \{\text{a}, \text{b}\}, \{\text{even}\}, \Delta)$

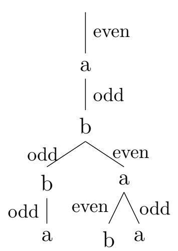
where $\Delta = \{$

([even, even],	a,	odd),	([even, even],	b,	even),
([even, odd],	a,	even),	([even, odd],	b,	odd),
([odd, even],	a,	even),	([odd, even],	b,	odd),
([odd, odd],	a,	odd),	([odd, odd],	b,	even),
([even],	a,	odd),	([even],	b,	even),
([odd],	a,	even),	([odd],	b,	odd),
([],	a,	odd),	([],	b,	even),

$\}$



(14)

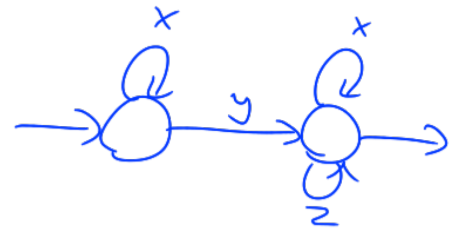


This grammar is *bottom-up deterministic*: given a sequence of "child states" and a symbol, there's at most one applicable transition. This reflects the fact that there's a *function* that determines whether a tree contains an even or odd number of 'a's. But this grammar is not *top-down deterministic*: note the "choices" one has to make at binary-branching nodes when working top-down.

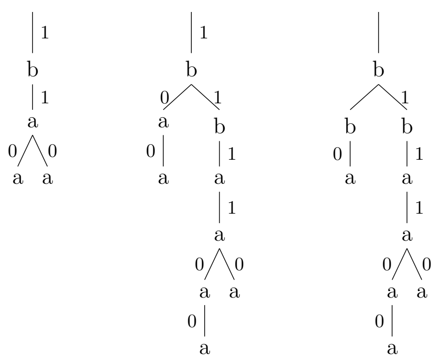
2.4.2 Another abstract example

The grammar in (15) generates the treeset L_2 from (9) above (requiring that every 'b' dominates a binary-branching 'a').

(15) $G_2 = (\{0,1\}, \{a,b\}, \{0,1\}, \Delta)$ where
$$\Delta = \left\{ \begin{array}{llllll} ([0,0], & a, & 1), & & & \\ ([0,1], & a, & 1), & ([0,1], & b, & 1), \\ ([1,0], & a, & 1), & ([1,0], & b, & 1), \\ ([1,1], & a, & 1), & ([1,1], & b, & 1), \\ ([0], & a, & 0), & & & \\ ([1], & a, & 1), & ([1], & b, & 1), \\ ([\,], & a, & 0), & & & \end{array} \right\}$$



(16)



2.4.3 A more linguistic example

Now let's suppose that the alphabet Σ is the set of English words, plus the additional symbol $*$.

(17) $\Sigma = \{*, \text{the}, \text{cat}, \text{dog}, \text{anybody}, \text{ever}, \text{not}, \text{nobody}, \dots\}$

Then the FSTA in (19) encodes 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'.

negative polarity items

(18) 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

NEG

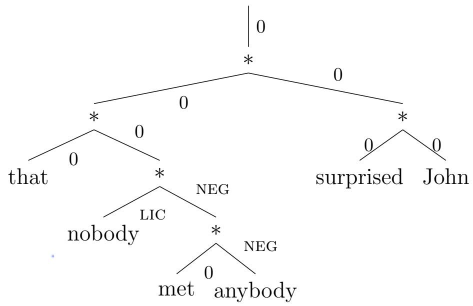
Nobody said [anybody met anybody]

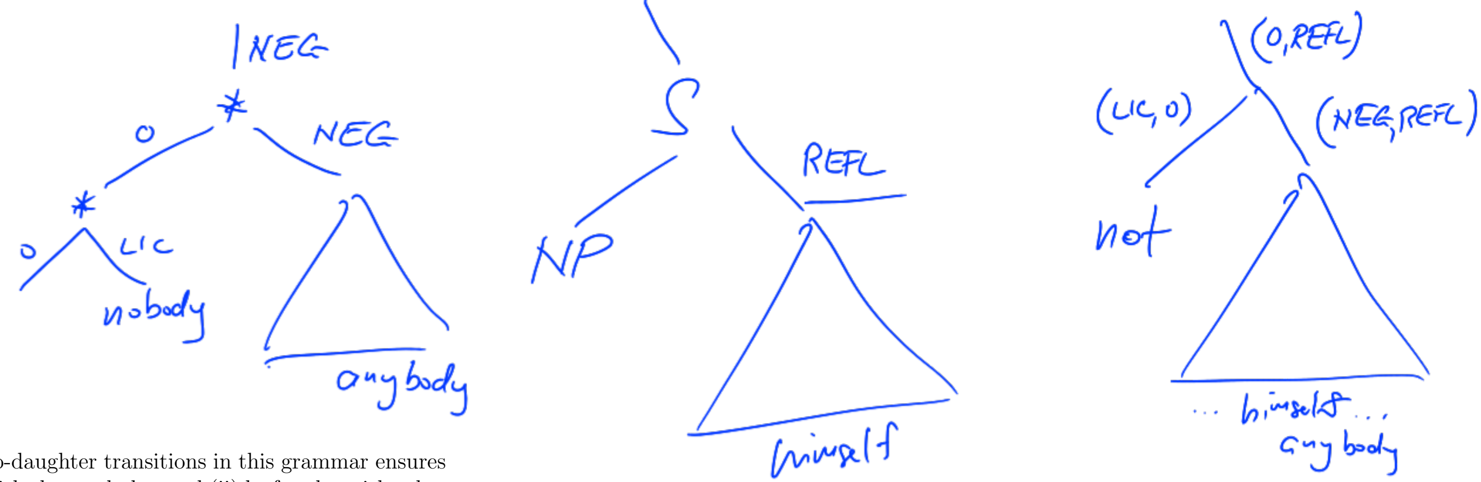
(19) $G_3 = (\{0, \text{LIC}, \text{NEG}\}, \Sigma, \{0, \text{LIC}\}, \Delta)$
where
$$\Delta = \left\{ \begin{array}{llllllll} ([\text{NEG},\text{NEG}], & *, & \text{NEG}), & ([\,], & \text{anybody}, & \text{NEG}), & & \\ ([0,\text{NEG}], & *, & \text{NEG}), & ([\,], & \text{ever}, & \text{NEG}), & & \\ ([\text{NEG},0], & *, & \text{NEG}), & ([\,], & \text{not}, & \text{LIC}), & & \\ ([0,0], & *, & 0), & ([\,], & \text{nobody}, & \text{LIC}), & & \\ ([\text{LIC},\text{NEG}], & *, & 0), & ([\,], & s, & 0) & \text{for any other } s \in \Sigma - \{*\}, & \\ ([\text{LIC},0], & *, & 0), & & & & & \\ ([0,\text{LIC}], & *, & 0), & & & & & \\ ([\text{LIC},\text{LIC}], & *, & 0) & & & & & \end{array} \right\}$$

([LIC,0], *, LIC)
([0,LIC], *, LIC)

John thinks [Mary likes himself]

(20)





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.

intersecting FSTA

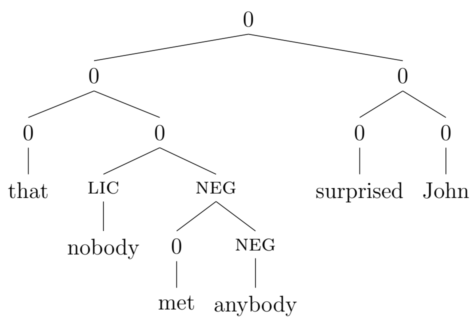
2.5 So what do FSTAs gain for us?

But wait a minute — if the goal is just to account for the *facts about strings* in (18), then we can do the same thing with a plain old CFG.

$0 \rightarrow$ LIC NEG

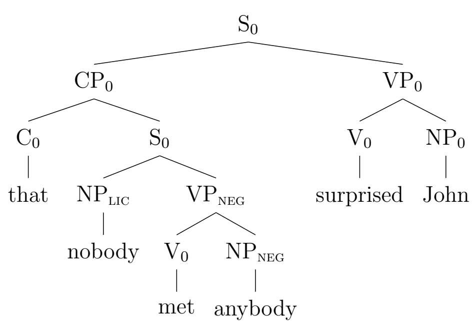
NEG $\rightarrow$ 0 NEG

(21)



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.[1]

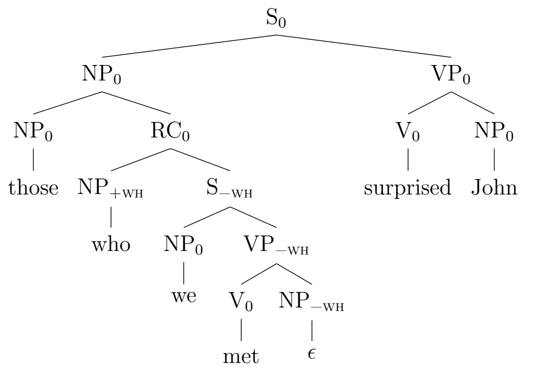
(22)



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

[1]Because the cartesian product of finite sets is necessarily finite.

(23)



"slash passing"
GPSG (1980s)

What [did John eat]$_{S/WHNP}$?
Where [did John go]$_{S/WHPP}$?

So while FSTAs are a useful tool for conceptualizing long-distance dependencies (such as NPI-licensing or wh-movement), it turns out that *if a stringset can be derived by "reading along the bottom" of all the trees generated by an FSTA, then that stringset can also be generated by a CFG.*

S → a S b c

3 Stringsets beyond context-free

Some examples of stringsets that cannot be generated by a CFG:

(24) a. $\{w + w \mid w \in \{a, b\}^*\}$
b. $\{a^n b^n c^n \mid n \geq 0\}$
c. $\{a^n b^n c^n d^n \mid n \geq 0\}$
d. $\{a^i b^j c^i d^j \mid i \geq 0, j \geq 0\}$

aabaab
$a^i b^j c^j d^i$
aaabbcccddd

These all exhibit *crossing dependencies*, rather than *nesting dependencies* of the sort that CFGs can handle. (Imagine trying to recognize these stringsets by by moving through strings from left to right, with an unbounded stack as your available memory.)

3.1 Non-context-free string patterns in natural language

≈ ECM / raising-to-object
We expect John to leave

To start, consider the following kinds of sentences in English:

(25) a. we [paint houses]
b. we [help [$_{SC}$ John paint houses]]
c. we [let [$_{SC}$ children help [$_{SC}$ John paint houses]]]

The subject of each "small clause" (SC) gets its case from the verb just above it. In many languages this would be shown overtly on the noun phrases somehow. And in many languages, the choice of verb would affect exactly which case (e.g. accusative or dative) gets assigned to each small clause subject. So we can imagine that the surface strings in fact look like this:

(26) a. we [paint houses]
b. we [help-DAT [$_{SC}$ John-DAT paint houses]]
c. we [let-ACC [$_{SC}$ children-ACC help-DAT [$_{SC}$ John-DAT paint houses]]]

Let's restrict attention to cases where the accusative-subject small clauses are all "outside" the dative-subject small clauses. Then the English word order pattern can be generated by an FSA: each accusative-assigning verb needs an accusative NP immediately after it, and likewise for each dative-assigning verb. The pattern is analogous to $\{(V_1N_1)^i(V_2N_2)^j \mid i \geq 0, j \geq 0\}$.

In a head-final language, we might expect to see a word-order like this:

(27) a. we [houses paint]
b. we [[SC John-DAT houses paint] help-DAT]
c. we [[SC children-ACC [SC John-DAT houses paint] help-DAT] let-ACC]

$a^i b^j c^j d^i$

This is beyond an FSA, but possible with a CFG: the very first NP is associated with the very last verb. This is analogous to $\{N_1^i N_2^j V_2^j V_1^i \mid i \geq 0, j \geq 0\}$.

Now the amazing fact: in (at least a certain dialect of) Swiss German, we find the following word order:

(28) a. we [houses paint]
b. we [John-DAT houses help-DAT paint]
c. we [children-ACC John-DAT houses let-ACC help-DAT paint]

$a^i b^j c^i d^j$

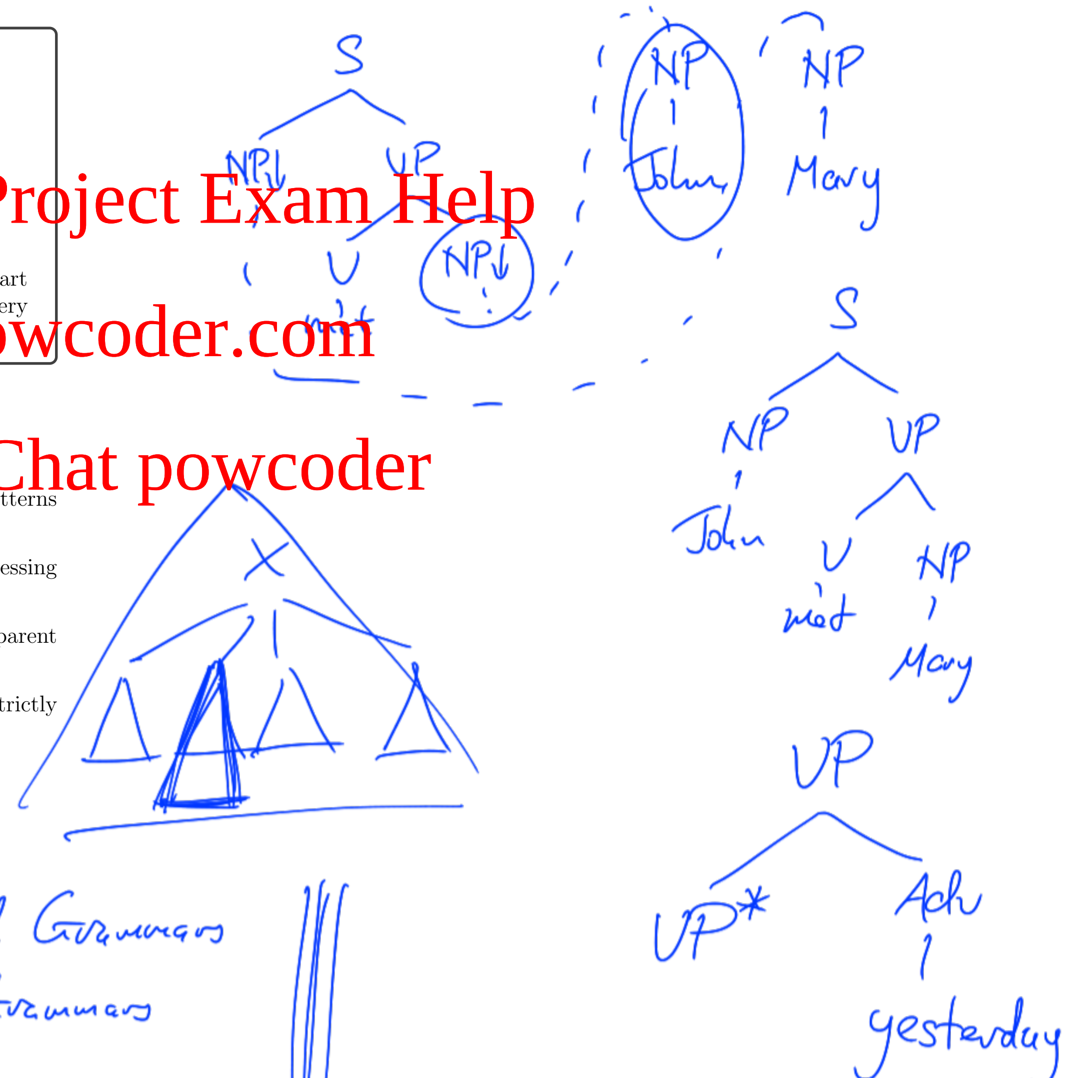
This pattern is analogous to $\{N_1^i N_2^j V_1^i V_2^j \mid i \geq 0, j \geq 0\}$, which no CFG can generate.

For the record, this is what the relevant parts of the actual sentences look like.

(29) a. daß mer em Hans es huus hälfe aastrüche
that we Hans.DAT the house.ACC helped paint
"that we helped Hans paint the house"

b. daß mer d'chind em Hans es huus lönd hälfe aastrüche
that we the children.ACC Hans.DAT the house.ACC let help paint
"that we let the children help Hans paint the house"

Papers presenting this argument were published in the mid-1980s by Riny Huybregts and by Stuart Shieber. Geoff Pullum's short article entitled "Footloose and context-free" (1986, *NLLT*) is a very amusing little (six-page) account of the historical development of the ideas.



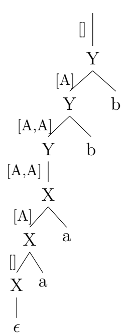
3.2 A more powerful grammar formalism

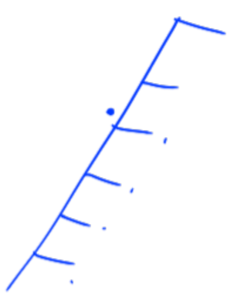
Here's the big idea, building on FSTAs:

- We've seen that in order to allow a left-to-right string-processing automaton to recognize patterns like $a^n b^n$, we need memory in the form of an *unbounded stack*, rather than just a single *state*.
- Let's introduce the same kind of unbounded stack memory into a bottom-to-top tree-processing automaton.
- We'll restrict/simplify the use of the stack slightly: stack information can only flow between a parent and *one* of its daughters.

This means that we can generate patterns like $a^n b^n$ along the leaves of a strictly left-branching (or strictly right-branching) tree. It's sort of like CFG parsing in disguise.

(30)



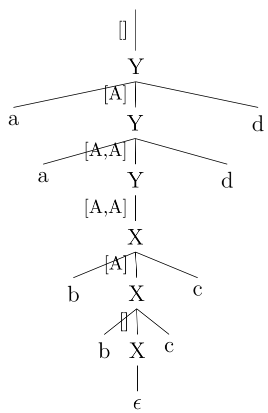


→ Linear Indexed Grammars
→ Tree-adjoining Grammars
Head Grammar
Combinatory Categorial Grammars

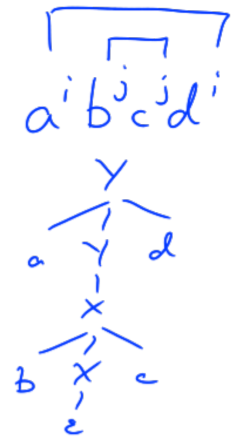
But when we combine this stack-based memory with center-embedding structures, magic happens!

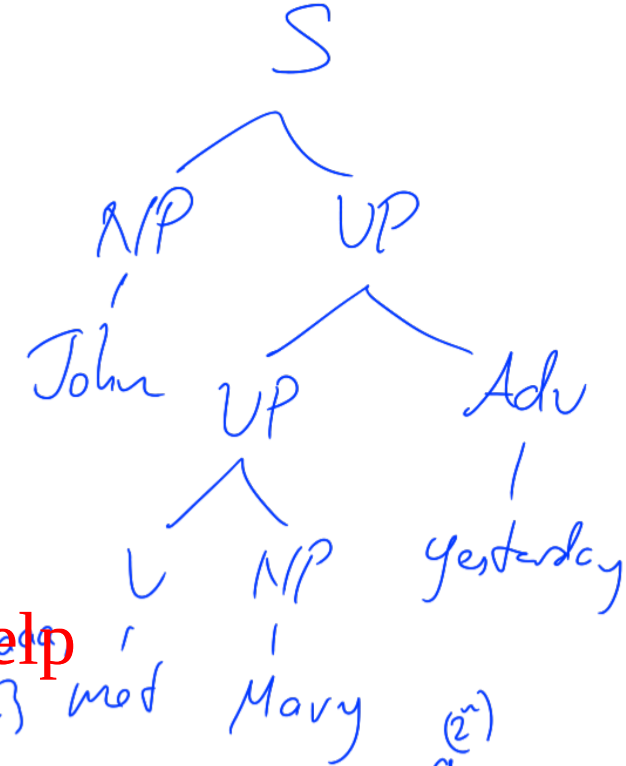
Here's the rough idea for how we can generate $\{a^n b^n c^n d^n \mid n \geq 0\}$.

(31)



$Y \to a\,Y\,d$
$Y \to X$
$X \to b\,X\,c$
$X \to \varepsilon$





Notice that the highest/outermost (a,d) pair is "matched up with" the lowest/innermost (b,c) pair: thinking bottom-up, the lowest (b,c) pair pushed the deepest 'A' onto the stack, and the highest (a,d) pair popped off that 'A'.

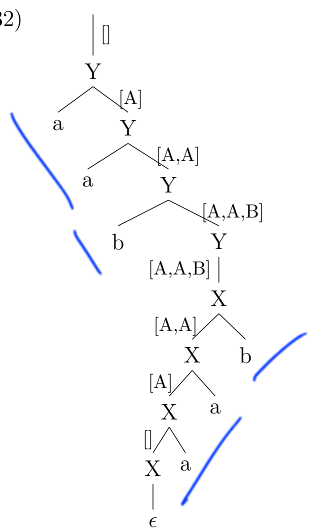
$a^{(2^n)} = \{a, aa, aaaa, aaaaaaaa \ldots\}$

This means that the *first* 'a' is in a dependency with the *first* 'c' — so we have generated *crossing* dependencies!

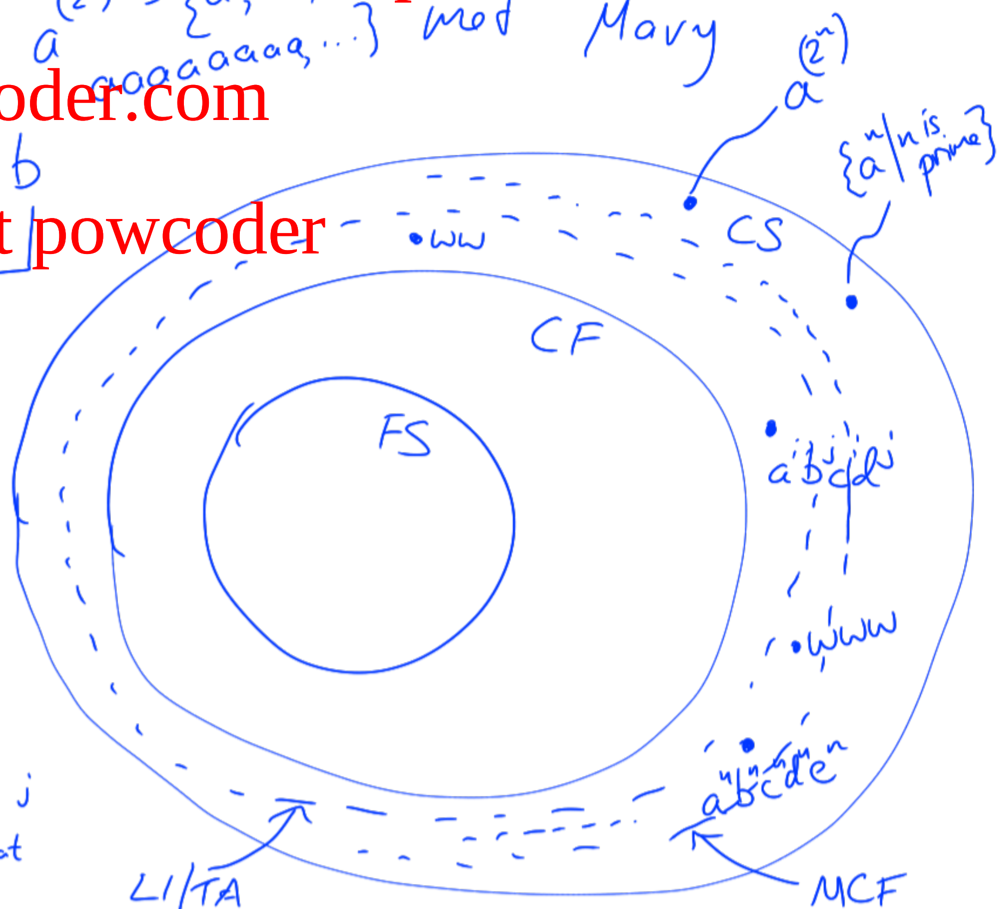
Here's the rough idea for how we can generate $\{w + w \mid w \in \{a, b\}^*\}$.

aabaab

(32)



$Y \to a\,Y$
$Y \to b\,Y$
$X \to X\,a$
$X \to X\,b$
$Y \to X$
$X \to \varepsilon$



In a similar way we can generate $\{a^i b^j c^i d^j \mid i \geq 0, j \geq 0\}$, which corresponds to the Swiss German case-marking pattern.

$NP_{acc}^{i}\ NP_{dat}^{j}\ V_{acc}^{i}\ V_{dat}^{j}$