Thursday September 5, 2024

Beyond String grammars: 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.

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

(2) For any set Σ, we define Σ^* as the smallest set such that:
a. $\epsilon \in \Sigma^*$, and
b. 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:ϵ)), 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. $L_6 = \{ww^R \mid w \in \Sigma^*\}$
g. $L_7 = \{ww \mid w \in \Sigma^*\}$

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:

(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.

2.1 The set of trees over an alphabet

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

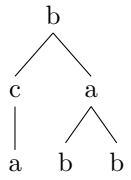
The square brackets in this definition are analogous to the colon in the definition of Σ^*. The colon makes strings out of symbols, and the square brackets make trees out of symbols. (These pieces of punctuation correspond to *constructors* in Haskell.)

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

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

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)



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^n_Σ as the set of all trees in T_Σ in which every node as at most n daughters.

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

The largest number of daughters of any node in a tree is sometimes called the tree's **branching degree**. So T^n_Σ 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 $\Sigma = \{a, b\}$, 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:

a. Q is a finite set of states;
b. Σ, the alphabet, is a finite set of symbols;
c. $F \subseteq Q$ is the set of ending states; and
d. $\Delta \subseteq Q^* \times \Sigma \times Q$ is the set of transitions, which must be finite.

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

(11) a. $under_G(x[])(q) = \Delta([], x, q)$

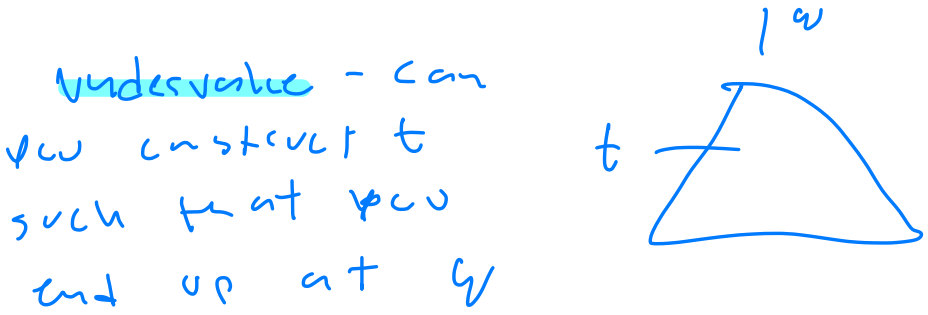
b. $under_G(x[t_1, \ldots, t_k])(q)$

$$= \bigvee_{q_1 \in Q} \cdots \bigvee_{q_k \in Q} \Big[\Delta([q_1, \ldots, q_k], x, q) \wedge under_G(t_1)(q_1) \wedge \cdots \wedge under_G(t_k)(q_k)\Big]$$

And given a way to work out $under_G(t)(q)$ for any tree and any state, we can easily use this to check for membership in $\mathcal{L}(G)$:

(12) $t \in \mathcal{L}(G) \Leftrightarrow \bigvee_{q \in Q} \Big[under_G(t)(q) \wedge F(q)\Big]$

As usual, slot in your favorite 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).

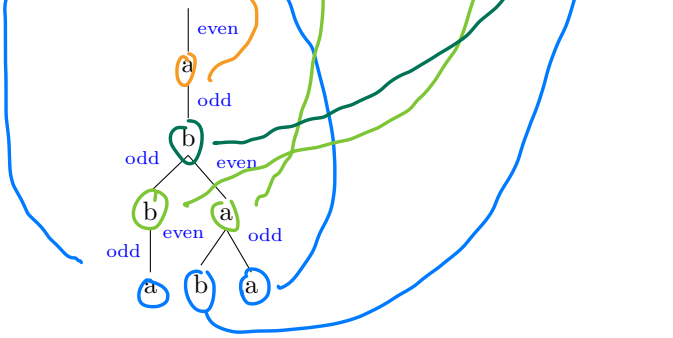
(13) $G_1 = (\{\text{even,odd}\}, \{\text{a,b}\}, \{\text{even}\}, \Delta)$

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)}

For example:

(14)



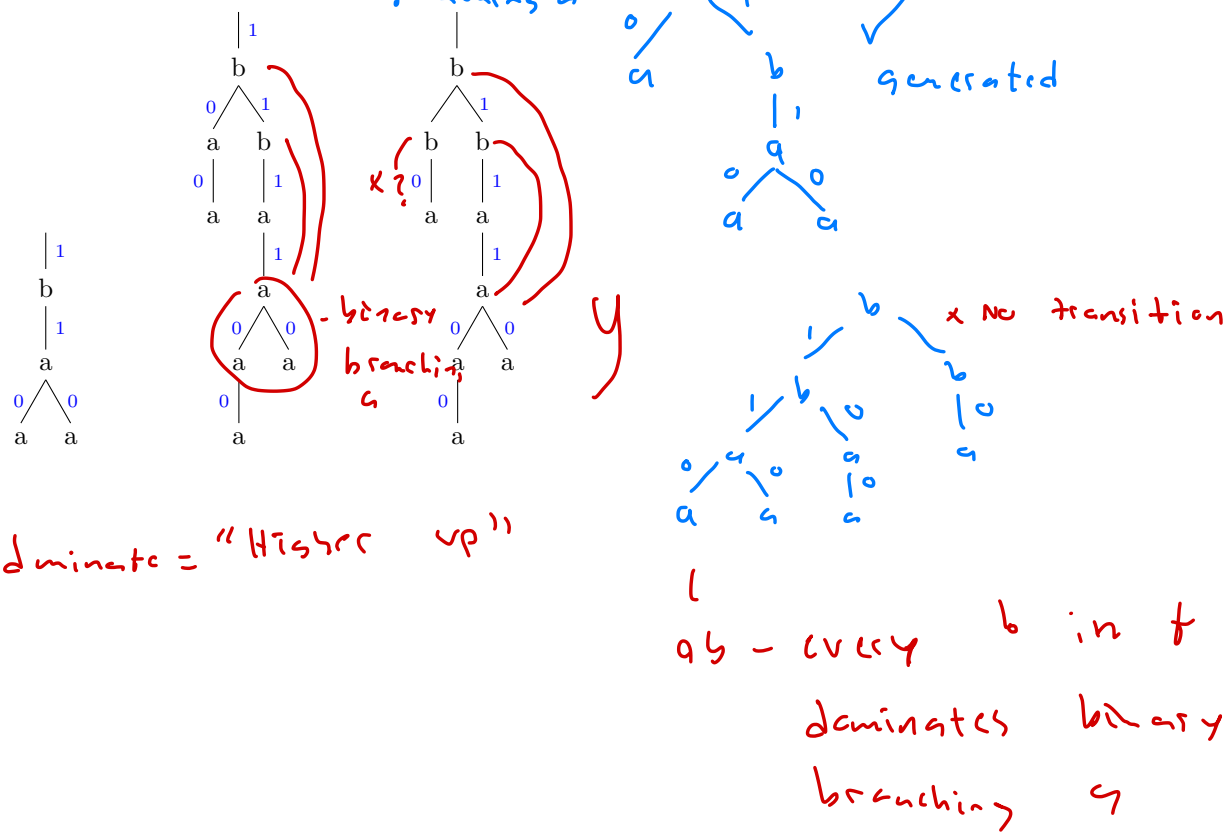
This grammar is *bottom-up deterministic*: given a sequence of "child states" and a symbol, there's at most one applicable transition. This reflects the fact that there's a function that determines whether a tree contains an even or odd number of 'a's. But this grammar is not *top-down deterministic*: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

2.4.2 Another abstract example

(15) $G_2 = (\{0,1\}, \{a,b\}, \{0,1\}, \Delta)$ where $\Delta =$

{([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)}			

(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 $\star$.

(17) $\Sigma = \{\star, \text{the, cat, dog, anbody, ever, not, nobody} \ldots\}$

Then the FSTA in 19 encodes a simple version of the *NPI-licensing constraint*: a Negative Polarity Item (NPI) such as 'anybody' or 'ever' must be c-commanded by a licensor such as 'not' or 'nobody'.

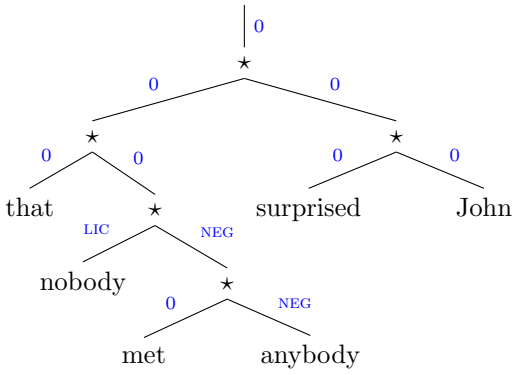
(18) a. Nobody met anybody
b. * John met anybody
c. Nobody thinks that John met anybody
d. The fact that nobody met anybody surprised John
e. * The fact that nobody met John surprised anybody

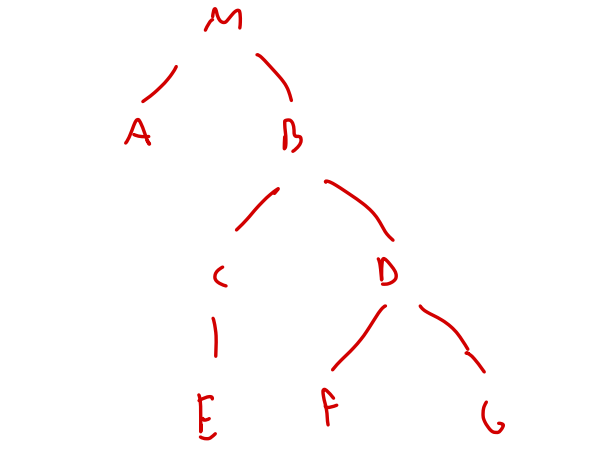
(19) $G_3 = (\{0,\text{LIC},\text{NEG}\}, \Sigma, \{0,\text{LIC}\}, \Delta)$

where $\Delta =$

{([NEG,NEG],	$\star$,	NEG),	([],	anybody,	NEG),	
([0,NEG],	$\star$,	NEG),	([],	ever,	NEG),	
([NEG,0],	$\star$,	NEG),	([],	not,	LIC),	
([0,0],	$\star$,	0),	([],	nobody,	LIC),	
([LIC,NEG],	$\star$,	0),	([],	s,	0)	for any other $s \in \Sigma - \{\star\}$,
([LIC,0],	$\star$,	0),				
([0,LIC],	$\star$,	0),				
([LIC,LIC],	$\star$,	0)}				

(20)

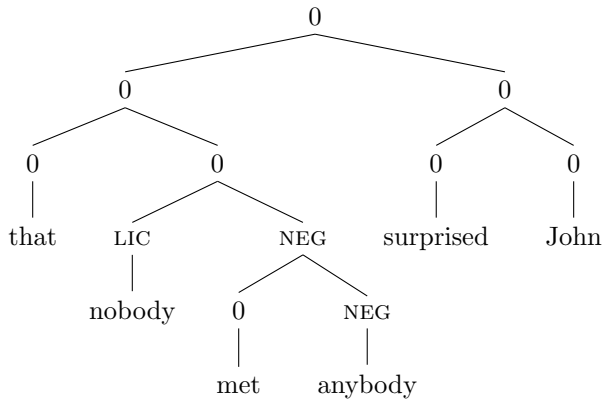




2.5 So what do FSTAs gain for us?

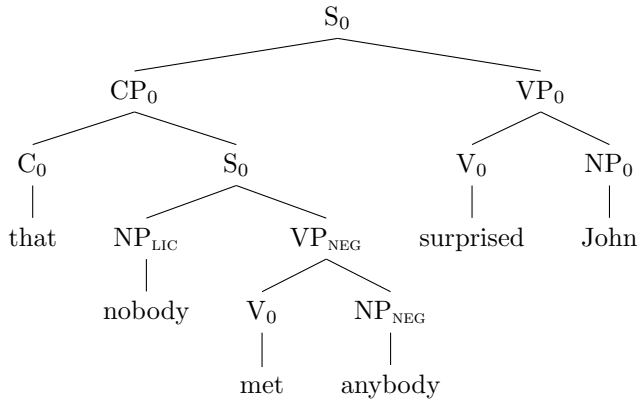
But wait a minute— if the goal is just to account for the *facts about strings* in (18), then we can do the same thing with a plain old CFG.

(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 phrase structure requirements. But we can just bundle all that information together.[1]

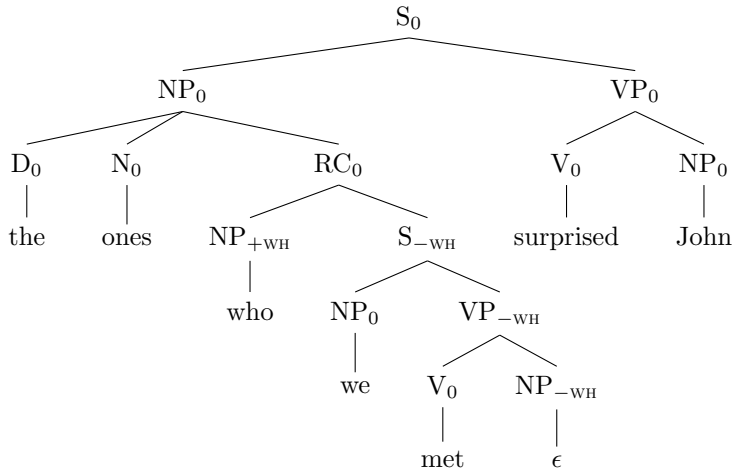
(22)



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

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

(23)

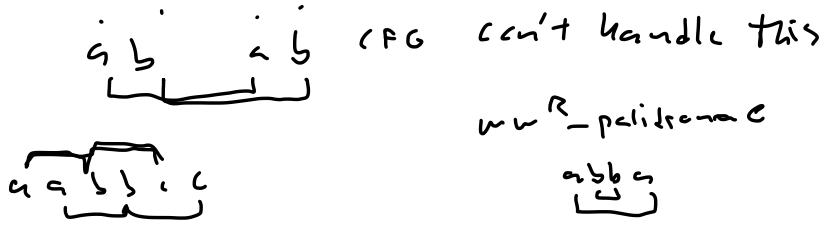


So while FSTAs are a useful tool for conceptualizing long-distance dependencies (such as NPI-licensing or *wh*-movement), it turns out that if a stringset can be derived by "reading along the bottom" of all the trees generated by an FSTA, then that stringset can also be generated by a CFG!

3 Stringsets beyond context-free

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

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



These all exhibit *crossing dependencies*, rather than *nesting dependencies* of the sort that CFGs can handle. (Imagine trying to recognize these stringsets 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

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]

This is beyond an FSA, but possible with a CFG: the very first NP is associated with the very last verb. The pattern is analogous to $\{N_1^iN_2^jV_2^jV_1^i \mid i \geq 0, j \geq 0\}$.

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]

This pattern is analogous to $\{N_1^iN_2^jV_1^iV_2^j \mid i \geq 0, j \geq 0\}$, 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 the Hans.DAT the house.ACC helped paint
"that we helped Hans paint the house"

b. *daß mer d'chind em Hans es huus lond hälfe aastrüche*
that we the children.ACC the Hans.DAT the house.ACC let help
"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 an amusing account of the historical development of the ideas.

3.2 A more powerful grammar formalism

Here's the big idea, building on FSTAs:

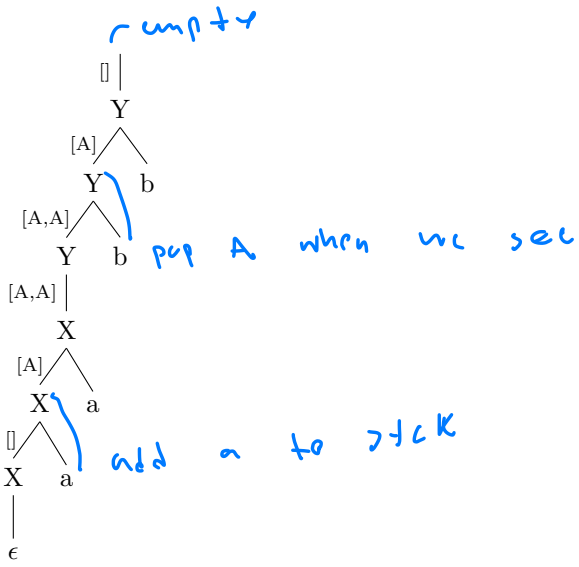
- We've seen that in order to allow a left-to-right string-processing automaton to recognize patterns like a^nb^n, we need memory in the form of an *unbounded stack*, rather than just a single state.
- Let's introduce the same kind of unbounded stack memory into a bottom-to-top tree-processing

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.

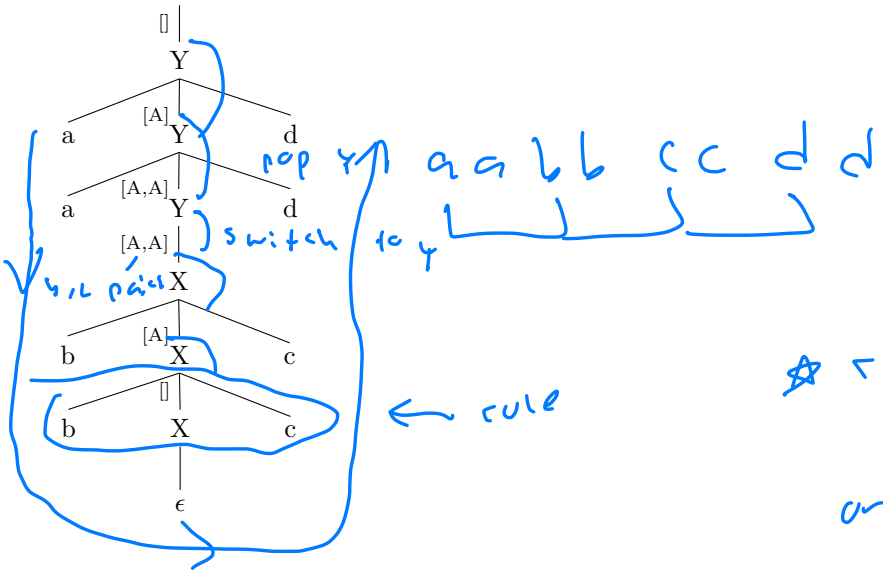
(30)



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

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

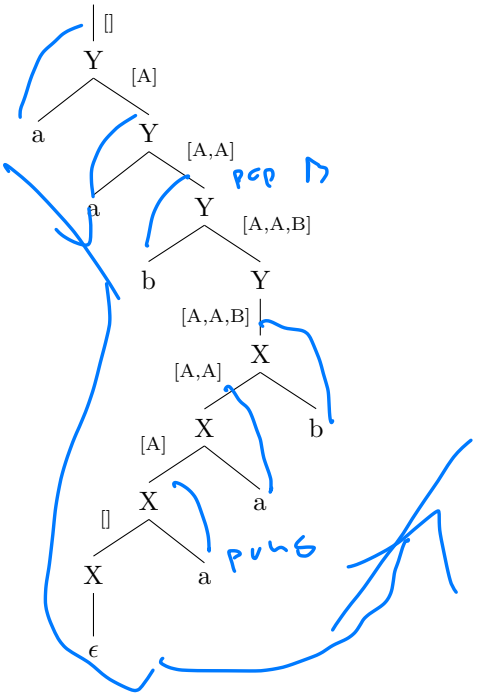
(31)



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'. 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 $\{ww \mid w \in \{\text{a,b}\}^*\}$.

(32)



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.