Lecture 11

Moving From Strings to Trees

We have been able to show that English is not a regular language due to the structural complexity of center embedding. This leaves us with two choices: we may either stick with finite-state models and claim that center embedding does not pose a challenge in practice, or we can once again move to a more expressive model. The first choice is a valid option in a variety of applications but requires some major sacrifices. We will see that these sacrifices cannot be reconciled with our project of exploring the properties of language from a computational perspective, and thus we will take a hint from linguists and expand our model by moving from strings to trees.

1 Finite-State Methods and the Role of Unboundedness

As mentioned last time, the reason that center-embedding patterns — abstractly represented as $a^n b^n$ — are beyond the capabilities of finite-state methods is that there is no upper bound on the depth of embedding. An FSA has to memorize the exact number of as in order to ensure that the same number of bs follow, but since an FSA has a fixed number k of states, it can only partition strings into $k + 1$ distinct equivalence classes (one equivalence class for each state, plus one for strings for which there is no defined transition). Hence some strings with distinct numbers of as must wind up in the same equivalence class and thus can be followed by the same number of bs according to the automaton, which is clearly not the case for the language $a^n b^n$.

This argument is mathematically flawless, but it faces a major empirical challenge: center-embedding is **not** unbounded in the data we have access to. No speaker spontaneously produces a sentence with ten levels of center-embedding, and even if we found a speaker with a propensity for convoluted center embedding constructions and convinced that person to spend the rest of their life uttering a single sentence full of center embeddings, that person could only produce a fixed number of embeddings before they die of old age. Even if we somehow could found a tradition of center-embedding performance art, where one speaker starts a sentence with center embeddings that are then continued by another speaker, and then another, on and on until the end of time, we could only reach a final level of embeddings before the universe collapses in on itself. We live in a finitistic universe, wherefore unboundedness is not an empirically verifiable property.

A slightly different, more application-oriented argument posits that even if humans might be theoretically capable of unbounded center-embedding, there is little reason to incorporate this property into our model if they never make use of it. Why make your model more powerful if that power is never needed? This is indeed a valid concern for industrial-grade applications, where time equals money and programs should run as quickly as possible. But even there this issue is not cut and dry. After all, "time equals money" also means that programs should be easy to extend, modify and maintain, since the manhours spent on these tasks do not come for free. When given a choice between an efficient but complicated tool on the one hand or a slower yet elegant tool on the other, the latter might be the better solution. In addition, the elegant tool might actually be the faster one in practice — just because it can theoretically perform much worse does not entail that is does so for the specific task at hand. As you can see, the unboundedness questions does not have a clear cut answer, it all depends on what your goals are.

The ideal solution would be an elegant tool that can be automatically translated into an efficient one if necessary, but that is not always feasible.

Our inquiry is driven by scientific curiosity, so questions of efficiency affect us only to the extent that the resource usage of our model has to be reconcilable with what we know about human cognition. And even this restriction does not outweigh our desire to state insightful generalizations. That is why we put such a high premium on abstraction: by deliberately excluding certain factors we can home in on broad, appealing generalizations. These generalizations still hold in a more detailed model that is closer to the wetware, but they are much harder to discern due to all the complicating factors that come with a more faithful model.

For our purposes, unboundedness is an essential assumption because boundedness acts as a great equalizer that pushes everything with the bounds of finite-state machinery. Recall that we assumed in our discussion of phonology that some long-distance processes are unbounded even though for all practical purposes the length of words is bounded in the same way as the number of center embeddings. We did this because it allowed us to formalize important differences between local and non-local processes without losing track of their commonalities. Similarly, assuming that center embedding is unbounded brings out an important difference between this non-regular kind of embedding and right embedding, which is still regular. This contrast is even reflected in human processing, where right embeddings are much easier to parse than center embeddings. Distinguishing bounded center embedding from bounded right embedding would be more involved.

A brief glance at an FSA for bounded center embedding also reveals that redundancy of this account. Each level of embedding corresponds to a specific subgraph of the automaton, but all these subgraphs look exactly the same. So we have a big number of states that all do exactly the same work, the only difference is that one is used in embedding level 2 and another one in level 5. This also raises the question why natural language grammars treat all those levels exactly the same — if the automaton distinguishes level 2 from level 3, why can't 3 use the opposite word order of 2? This level-sensitive automaton would have exactly the same number of states as the one that treats all levels of embedding the same. If we assume that center embedding is unbounded, though, then it might be possible to show that level-sensitive formalisms are more complicated than those that treat all levels the same.

To sum up, unboundedness is not an undisputable fact, first and foremost it is yet another abstraction in the service of exploring specific questions. However, what we find may serve as indirect evidence for unboundedness when embedded in a system of ancillary assumptions. For example, the enormous size of FSAs with bounded center embedding and the lack of level-sensitive embeddings in natural language suggest that unboundedness is cognitively real, at least if one assumes that small, succinct grammars are preferred for some reason. With other constructions, there may be less of a reason to assume that they are unbounded (for instance multiple wh-movement), so we will always have to weigh carefully what the benefits of unboundedness may be on a case-by-case basis.

2 Tree Languages

2.1 Context-Free Grammars

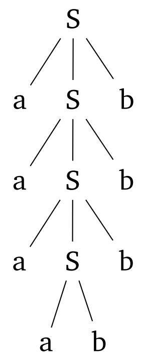
While unbounded center embedding exceeds the limits of finite-state methods, it is easily accounted for with phrase structure rules. The language $a^n b^n$, $n \geq 1$, is generated by two rules, which can even be conflated into a single one with some syntactic sugar:

$$\text{S} \quad \rightarrow \quad \text{ab} \mid \text{aSb}$$

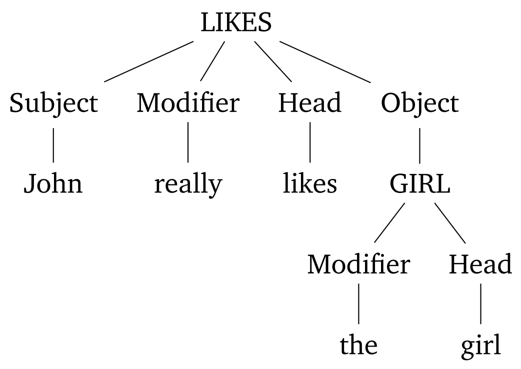
We can interpret this rule as a mechanism for rewriting strings, where we start with S and then apply rules until no more rewriting steps are possible.

Rule	**String**
start	S
S → aSb	aSb
S → aSb	aaSbb
S → aSb	aaaSbbb
S → ab	aaaabbbb

Linguists are more familiar with the tree-based representation of rule application.



Phrase structure grammars are also known as *context-free grammars* (CFGs). The latter term focuses on the rule format, which must be of the form $A \rightarrow \alpha$, where α is a string of symbols. These rules lack the context specification used by SPE, among others, so they are indeed context-free. The term phrase structure grammar instead focuses on what the grammar is meant to describe, namely the phrase structure of sentences. Obviously CFGs can be used to describe other kinds of structures. For instance, the sentence *John really likes the girl* could be assigned the tree below, which represents the functional relations between words (this tree is inspired by Dependency Grammar, which we will discuss at a later point).



As you can see, the term context-free grammars is slightly more general even though it refers to exactly the same kind of mathematical object. Fans of generality that we are, we will henceforth speak of CFGs rather than phrase structure grammars.

Definition 11.1 (CFG). A *context-free grammar* is a triple $G := \langle \Sigma, S, R \rangle$, where

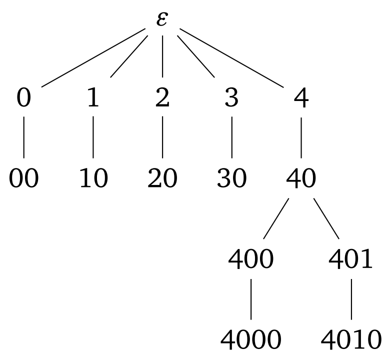
- Σ is an alphabet,
- $S \in \Sigma$ is the *start symbol*,
- R is finite set of rules $A \rightarrow \alpha$ such that $A \in \Sigma$ and $\alpha \in \Sigma^*$.

A symbol $a \in \Sigma$ is *terminal* iff R contains no rule of the form $a \rightarrow \alpha$. Otherwise a is *non-terminal*. The corresponding subsets of Σ are denoted T_Σ and N_Σ. We always require $S \in N_\Sigma$. The language $L(G)$ generated by G is the smallest set containing all strings that I) can be obtained from S by finitely many applications of rules in R, and II) contain no non-terminal.

Since CFGs can handle center embedding and are equivalent to the familiar linguistic formalism of phrase structure rules, they are a promising starting point for a formal model of syntax. We will soon see that they are indeed very closely related to a model that we have explored in great detail.

2.2 Trees and Tree Languages

Before we move on, it will be useful to formalize trees. This can be done in a plethora of ways, but we will opt for the definition in terms of *Gorn domains* (Gorn 1967) because it couples each node with an address that directly represents its location in the tree. Intuitively, a Gorn domain is a set of addresses of the form $m \cdot l$, where m is the address of the node's mother and l the number of left siblings it has. So the tree for *John really likes the girl* that was given in the previous section would be associated with node addresses as shown below:



The address for the root is ε since it has neither a mother nor a left sibling. For all other nodes, the formula $m \cdot l$ applies as described.

Notice that an entry like 40 is read "four-zero" since it is the leftmost daughter of the fifth daughter of the root, whereas 40 "forty" would refer to the 41st daughter of the node. This ambiguity is due to the decimal system being incapable of representing the difference between $4 \cdot 0$ and 40. Strictly speaking, an address like 401 should actually be written as $4-0-1$ to distinguish it from $40-1$ and 401, but this creates clutter that is best avoided when possible.

Definition 11.2 (Gorn domain). A *Gorn domain* D is a subset of $\mathbb{N}^*$ such that

dominance closure $ui \in D$ implies $u \in D$,

left sibling closure $ui \in D$ implies $uj \in D$ for all $0 \leq j < i$.

We call $u \in D$ a *leaf* iff there is no $i \in \mathbb{N}$ such that $ui \in D$. Given some subset S of D, ui is a *root* of S iff $u \notin S$.

A tree is a Gorn domain that maps each node address to a node label.

Definition 11.3 (Tree). A (finite) Σ-*tree* is a pair $t := \langle D, \ell \rangle$, where

- D is a (finite) Gorn Domain,
- $\ell : D \to \Sigma$ is a total function.

The *string yield* $\text{yd}(t)$ of t is the longest string $s_1 \cdots s_n$ such that

- s_i is a leaf of D $(0 \leq i \leq n)$,
- for $s_i := mu$ and $s_j := nv$ $(m, n \in \mathbb{N}, u, v \in \mathbb{N}^*)$, $i < j$ iff $m < n$.

The *depth* of subtree t with root u is the length of the longest i such that $ui \in D$.

Unless indicated otherwise, all trees are assumed to be finite.

Since we now have both strings and trees as mathematical objects, it makes sense to distinguish between *string languages* and *tree languages*. The former are sets of strings, the latter sets of trees. All the languages we have seen so far were string languages.

Definition 11.4 (Tree Language). A *tree language* L over Σ is a set of Σ-trees. Its string yield is the string language $\text{yd}(L) := \{\text{yd}(t) \mid t \in L\}$.

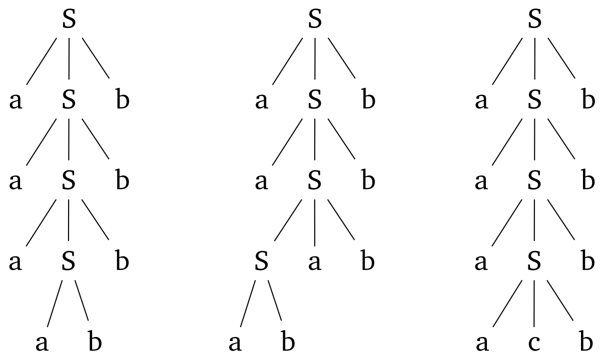
2.3 Tree Languages of Context-Free Grammars

Our definition of CFGs defines the string language generated by the grammar, but not the tree language even though we saw that sequences of rewriting steps can be represented as trees. The intuition for going from rewriting rules to trees is simple enough:

1. Draw a node with label S.
2. If A is rewritten as $\alpha := a_1 \cdots a_n$, add $a_1, \ldots, a_n$ as daughters of the corresponding node (in the same left-to-right order).

This way a CFG can be treated as a mechanism for building trees rather than just strings. But what exactly is the tree language generated by some random CFG?

Let us first think about whether certain trees are generated by a specific grammar, and why this is the case. Below you have several trees. The left one is generated by the CFG $\langle \{S, a, b\}, \{S \rightarrow aSb, S \rightarrow ab\} \rangle$, the others are not.



The left one satisfies all the conditions the rewrite rules establish with respect to the mother-of and the left-sibling relation. The tree in the middle contains a subtree where S is a left sibling of a, which can never happen with the specified rewrite rules. The third tree contains c as a daughter of S, which is also impossible. Notice that all these violations can be verified in a local fashion: we only have to look at a node and its daughters to find ill-formed subtrees. So we can postulate the following:

Definition 11.5 (CFG Tree Language). A CFG G generates a tree t iff

- the root of t is the start symbol, and
- all leaves of t are terminal, and
- for every subtree s of t such that s is of the form $[_A A_1 \cdots A_n]$, G contains a rule $A \rightarrow A_1 \cdots A_n$.

The tree language of G is the set of all trees that are generated by G.

This is a definition, not a theorem. We are simply describing what kind of trees are built via the translation procedure used by linguists. But we can show that this definition is sensible in the sense that the tree language yields the same string language as the grammar.

Theorem 11.6. Let G be a CFG that generates string language L and tree language T. Then $L = \text{yd}(T)$. ⌟

Proof. To see that $L \subseteq \text{yd}(T)$, take any string $w \in L$. By definition, w was obtained from S via a finite number of applications of rewrite rules of G. The standard translation from such rule applications to trees yields a tree that is generated by G and has w as its string yield.

In the other direction $\text{yd}(T) \subseteq L$ follows from the fact that if t is generated by G, then each subtree $[_A\, A_1 \cdots A_n]$ is matched by a rewrite rule of G. Hence there is a sequences of rewrite rules that produces the string yield of t. Since t's string yield consists only of terminal symbols, and t's root is S, $\text{yd}(t)$ is generated by G. □

2.4 Strictly Local Tree Languages

The condition that every subtree of depth 1 must be matched by a rewrite rule could be simplified by converting each rewrite rule into a tree of depth 1. Instead of a finite set of rewrite rules, one would then have a finite set of subtrees of depth 1, and a tree is generated by the grammar iff all its subtrees of depth 1 are included in this set. This idea should sound awfully familiar to you: it is the tree analogue of strictly local grammars.

Definition 11.7 (*k*-trees). A *k-tree* over alphabet Σ is a tree of depth $k-1$. A *k-augment* is a unary branching tree of depth $k-1$ where every node is labeled $\rtimes$. Given a Σ-tree t, its *k-augmented* counterpart $\hat{t}_k$ is the result of adding a k-augment above the root and below each leaf. The set of k-trees of a tree t is given by 2-trees$(t) := \{s \mid s \text{ is a subtree of } t \text{ with depth } k-1\}$.

Definition 11.8 (Strictly Local Tree Language). A finite set of k-trees is called a *strictly k-local tree grammar*. A positive strictly k-local tree grammar G generates the tree language $L(G) := \{t \mid k\text{-trees}(t) \subseteq G\}$. A negative strictly k-local tree grammar G generates the tree language $L(G) := \{t \mid k\text{-trees}(t) \cap G = \emptyset\}$. A tree language L is strictly k-local iff it is generated by some strictly k-local tree grammar. The class of strictly k-local tree languages is denoted SL^{T}_k, and the class of all *strictly local tree languages* is given by $\text{SL}^{\text{T}} := \bigcup_{k \geq 1} \text{SL}^{\text{T}}_k$.

Many of the theorems that we established for strictly local string languages can easily be lifted to strictly local tree languages. Before we do that, though, let us verify that strictly local tree languages are indeed closely related to the tree languages generated by context-free grammars.

Theorem 11.9. The class of tree languages generated by CFGs is properly included in the class of strictly 2-local tree languages. ⌟

Proof. For every CFG G one can construct an equivalent strictly 2-local tree grammar G_2. For every rewrite rule $A \to A_1 \cdots A_n$ of G, G_2 contains the 2-tree $[_A\, A_1 \cdots A_n]$. For every terminal symbol a, we add $[_a \rtimes]$. Finally, G also contains $[_\rtimes S]$. It is easy to see that G_2 generates exactly the same tree language.

Inclusion is proper because a symbol can be both terminal and non-terminal in a strictly 2-local tree language. Consider for instance the grammar $\{[_\rtimes S], [_S \rtimes]\}$, which generates only the tree S. □

2.5 Properties of Strictly Local Tree Languages

Just as we did with for strictly local string languages, we first show that it does not matter whether a grammar is positive or negative. That way, we can freely choose between the two in all other proofs.

Lemma 11.10. Let $T(\Sigma, n)$ be the set of Σ trees that are at most n-ary branching (i.e. every node has at most n daughters). For every positive strictly k-local grammar ($k \geq 1$) that generates some tree language L over Σ, there is a negative strictly k-local grammar that generates a tree language L' over Σ such that $L \cap T(\Sigma, n) = L' \cap T(\Sigma, n')$. ⌟

Proof. Since there is only a finite number of at most n-ary branching k-trees over Σ, we can adopt the strategy we used in the string case: the negative grammar is defined as the set of all these k-trees that are not contained by the positive grammar. When restricted to $T(\Sigma, n)$, both grammars generate the same set of trees. □

The equivalence of positive and negative grammars is no longer as strong as in the string case because it now hinges on a restriction on the allowed maximum branching factor. If the branching factor is not fixed, then the two grammar formats diverge significantly: a positive grammar can only generate trees with a fixed branching factor (namely the maximum branching factor of all k-trees in the grammar), whereas a negative grammar has no such upper bound (if all its k-trees are binary branching, than all trees that are not binary branching are automatically well-formed). This was not an issue for strings, which are essentially unary branching trees. It won't affect us much either, as most theories of syntax assume that branching is binary, or at least less than 4-ary, and we will henceforth assume that the branching factor is restricted to some suitable number.

One exception is "flat" analyses of coordination, where each conjunct adds a new branch. There is plenty of evidence, though, that such an analysis is incorrect.

All other properties of strictly local string languages also carry over with some minor adjustments.

Theorem 11.11. For all $k \geq 1$, $\mathrm{SL}^{\mathrm{T}}_{k-1} \subsetneq \mathrm{SL}^{\mathrm{T}}_{k}$. ⌟

Proof. We use the fact that every string $w := a_1 a_2 \cdots a_{n-1} a_n$ can be viewed as a unary branching tree $t(w) := [_{a_1}[_{a_2}[\cdots[_{a_{n-1}} a_n]\cdots]]]$. Just like the singleton string language $L^S_n := \{a^n\}$ is strictly $n+1$-local but not strictly n-local, the singleton tree language $L^T_n := \{t(a^n)\}$ is in $\mathrm{SL}^{\mathrm{T}}_{n+1}$ but not in $\mathrm{SL}^{\mathrm{T}}_{n}$. □

By viewing strings as unary branching trees, several non-closure results also carry over immediately.

Theorem 11.12. The class of strictly local tree languages is not closed under union, relative complement, or relabeling. ⌟

Proof. Remember that the union of $\left\{a^k b^k\right\} \in \mathrm{SL}_{k+1}$ and $\left\{b^+\right\} \in SL_2$ is not strictly local. Since each string can be viewed as a unary branching tree, this also establishes non-closure under union for strictly local tree languages.

For relative complement, consider the tree analogue T of the language $\{ab^*a\}$, which is in $\mathrm{SL}^{\mathrm{T}}_{2}$. Yet $T(\Sigma, n) \setminus T$ is not strictly local.

Non-closure under relabeling is once again witnessed by the languages $(ab)^+$ and $(aa)^+$, lifted from strings to unary branching trees. The former is in $\mathrm{SL}^{\mathrm{T}}_{2}$, whereas the latter is not strictly local. Yet the latter is a relabeling of the former. □

Theorem 11.13. For every $k \geq 0$, $\mathrm{SL}^{\mathrm{T}}_k$ is closed under intersection. ⌟

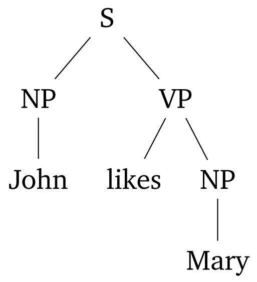
Proof. The strategy is once again to show that $L(G_1) \cap L(G_2) = L(G_1 \cap G_2)$, where G_1 and G_2 are positive strictly k-local tree grammars. This is left as an exercise to the reader. □

2.6 Local Subtree Substitution

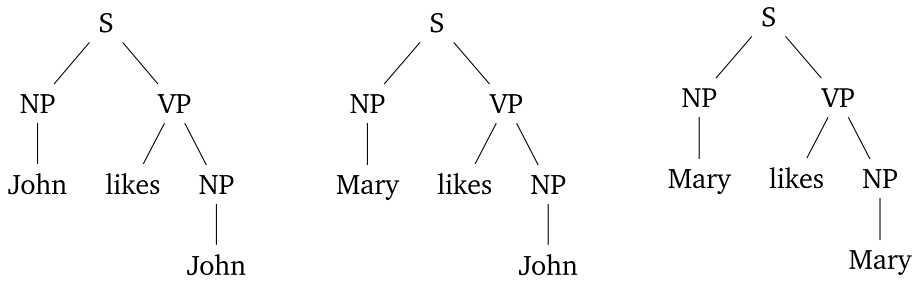
Strictly local string languages are fully characterized by local subtree substitution closure, as was discussed in Sec. 2.3. Intuitively, it is fairly obvious that a similar property holds for strictly local tree languages.

Example 11.1 Context-Free Grammars and Subtree Substitution

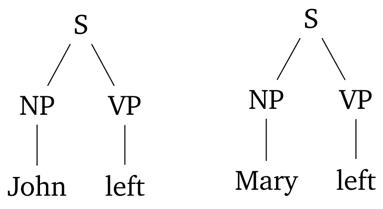
Suppose we have a context-free grammar that generates the tree below.



Then the grammar must generate at least three more trees:



This follows immediately from the fact that these three trees only use k-trees that also occur in the first one. If we furthermore know that the left tree below is generated by the grammar, then we know that it also generates the right one. In this case, there is no previous tree that contains all k-trees of the right tree, but its k-trees are a subset of the union of the k-trees of some trees that are already known to be grammatical.



For strictly 2-local tree languages, all trees with identical root labels can be freely substituted for one another. Note that these are exactly the subtrees that are identical up to depth 0. As the locality domain increases, so does the amount of material that two trees must have in common before they can be exchanged. For example, in a 4-local tree language we can substitute subtree t for u iff t and u are identical up to and including depth 2. We use the notation $s[u \leftarrow t]$ for the tree that is obtained from tree s by replacing the subtree of s rooted in address u by the tree t. So if s the tree $[_{\mathrm{S}}$ $[_{\mathrm{NP}}$ John] $[_{\mathrm{VP}}$ left]], then $s[00 \leftarrow \text{Mary}]$ is $[_{\mathrm{S}}$ $[_{\mathrm{NP}}$ Mary] $[_{\mathrm{VP}}$ left]]. The notation can also be nested:

$$s[00 \leftarrow \text{Mary}[\varepsilon \leftarrow \text{John}]] = s[00 \leftarrow \text{John}] = s$$

And several substitutions can be carried out at once as long as no address is a prefix of another (that is to say, the nodes at which we substitute do not stand in the dominance relation). For example, $s[00 \leftarrow \text{Mary}, 10 \leftarrow \text{laughed}] = [_{\mathrm{S}}$ $[_{\mathrm{NP}}$ Mary] $[_{\mathrm{VP}}$ laughed]].

Definition 11.14 (Subtree Substitution). Let $s := \langle D_s, \ell_s \rangle$ and $t := \langle D_t, \ell_t \rangle$ be Σ-trees and $u \in D_s$. Then $s[u \leftarrow t]$ is the unique tree $\langle D, \ell \rangle$ s.t.

- $D := (D_s \setminus \{a \mid a = ui, i \in \mathbb{N}^*\}) \cup \{a \mid a = ui, i \in D_t\}$,
- if $a \in \{a \mid a = ui, i \in D_t\}$, then $\ell(a) = \ell_t(a)$,
- otherwise $\ell(a) = \ell_s(a)$.

Suppose $u_1, \ldots, u_n$ are elements of D_s such that no u_i is a prefix of some distinct u_j. Then

$$s[u_1 \leftarrow t_1, u_2 \leftarrow t_2, \ldots, u_n \leftarrow t_n] := (\ldots((s[u_1 \leftarrow t_1])[u_2 \leftarrow t_2])\ldots)[u_n \leftarrow t_n]$$

With that little bit of notation out of the way, we can finally characterize strictly local tree languages in terms of local subtree substitution closure.

Definition 11.15 (Local Subtree Substitution Closure). A tree language L^T satisfies *local subtree substitution closure* iff there is some $k \geq 1$ such that if

- L^T contains $s[u \leftarrow x[x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n]]$, and
- L^T contains $t[u' \leftarrow x[x_1 \leftarrow v'_1, \ldots, x_n \leftarrow v'_n]]$, and
- x is a tree of depth $k - 1$,

then L also contains $s[u \leftarrow x[x_1 \leftarrow v'_1, \ldots, x_n \leftarrow v'_n]]$.

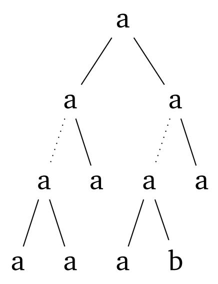
As in the string case, we omit the proof here, which is rather lengthy. The interested reader is referred to Rogers (1997).

Example 11.2 Some Non-Local Tree Languages

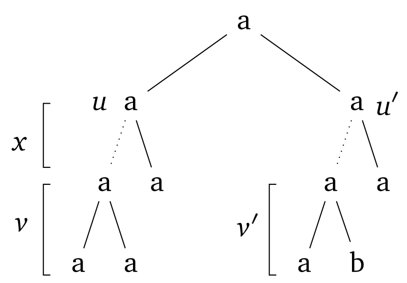
The simplest application of local subtree substitution shows that the language L of unary branching trees over the alphabet $\{a\}$ that contain an even number of as is not strictly local. The counterexample is exactly the one we already know from the string case. Suppose $k \geq 2$ is even. Then for s we pick $[_{\mathrm{a}}\ \mathrm{a}\ ]$ and for t just a; x is the tree $[_{\mathrm{a}} \cdots [_{\mathrm{a}} a] \cdots]$ with $k-1$ instances of a (an odd number). Finally, $v = t$ and $v' = s$. Then we have $s[0 \leftarrow x[0^{k-2} \leftarrow v]] \in L$, $t[\varepsilon \leftarrow x[0^{k-2} \leftarrow v']] \in L$, yet $s[0 \leftarrow x[0^{k-2} \leftarrow v']] \notin L$. Let us quickly verify this for $k = 4$:

$$\begin{array}{rll}
x[0^{k-2} \leftarrow v] & = [_{\mathrm{a}}\, [_{\mathrm{a}}\ \mathrm{a}\ ]][00 \leftarrow \mathrm{a}] & = [_{\mathrm{a}}\, [_{\mathrm{a}}\ \mathrm{a}\ ]] \\
s[0 \leftarrow x[0^{k-2} \leftarrow v]] & = [_{\mathrm{a}}\ \mathrm{a}\ ][0 \leftarrow [_{\mathrm{a}}\, [_{\mathrm{a}}\ \mathrm{a}\ ]]] & = [_{\mathrm{a}}\, [_{\mathrm{a}}\, [_{\mathrm{a}}\ \mathrm{a}\ ]]] \in L \\
\hline
x[0^{k-2} \leftarrow v'] & = [_{\mathrm{a}}\, [_{\mathrm{a}}\ \mathrm{a}\ ]][00 \leftarrow [_{\mathrm{a}}\ \mathrm{a}]] & = [_{\mathrm{a}}\, [_{\mathrm{a}}\, [_{\mathrm{a}}\ \mathrm{a}\ ]]] \\
t[\varepsilon \leftarrow x[0^{k-2} \leftarrow v']] & = \mathrm{a}[\varepsilon \leftarrow [_{\mathrm{a}}\, [_{\mathrm{a}}\, [_{\mathrm{a}}\ \mathrm{a}\ ]]]] & = [_{\mathrm{a}}\, [_{\mathrm{a}}\, [_{\mathrm{a}}\ \mathrm{a}\ ]]] \in L \\
\hline\hline
s[0 \leftarrow x[0^{k-2} \leftarrow v']] & = [_{\mathrm{a}}\ \mathrm{a}][0 \leftarrow [_{\mathrm{a}}\, [_{\mathrm{a}}\, [_{\mathrm{a}}\ \mathrm{a}\ ]]]] & = [_{\mathrm{a}}\, [_{\mathrm{a}}\, [_{\mathrm{a}}\, [_{\mathrm{a}}\ \mathrm{a}\ ]]]] \notin L
\end{array}$$

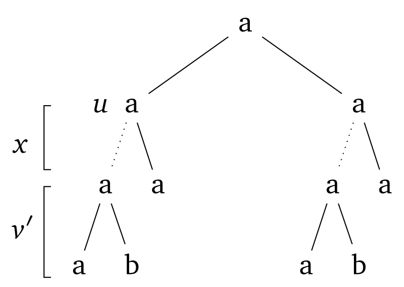
While the reasoning is simple, the notation makes it very hard to follow, so in general it is advisable to use a tree-based representation instead, as we shall do for our next example. Let $L_{b=1}$ be the language of all strictly binary branching trees over $\{a, b\}$ that contain exactly one instance of b. This language contains at least the following tree, where the dashed branch spans a subtree of depth $k-1$.



Then we can carve up the tree in a way so that we can assemble the parts into an illicit tree via local subtree substitution.



The whole tree is chosen for both s and t. Then we have $s[u \leftarrow x[0^{k-2} \leftarrow v]] = t[u' \leftarrow x[0^{k-2} \leftarrow v']] = s \in L_{b=1}$. But $s[u \leftarrow x[0^{k-2} \leftarrow v']]$ is not in $L_{b=1}$.


a
u
a
a
x
a
a
a
a
v′
a
b
a
b