Automata for subregular syntax: Syntax with strings attached

Thomas Graf and **Kenneth Hanson**
Department of Linguistics
Stony Brook University
mail@thomasgraf.net and mail@kennethhanson.net

Abstract

Building on recent work in subregular syntax, we argue that syntactic constraints are best understood as operating not over trees, but rather strings that track structural relations such as dominance and c-command. Even constraints that seem intrinsically tied to trees (e.g. constraints on tree tiers) can be reduced to such strings. We define *serial constraints* as an abstraction that decomposes string constraints into a *context function* (which associates nodes with strings) and a *requirement function* (which enforces constraints on these strings). We provide a general procedure for implementing serial constraints as deterministic tree automata. The construction reveals that the many types of constraints found in subregular syntax are variants of the same computational template. Our findings open up a string-based perspective on syntactic constraints and provide a new, very general approach to the automata-theoretic study of subregular complexity.

1 Introduction

One of the most common assumptions in theoretical and computational linguistics alike is that syntax does not operate over strings but rather trees, DAGs, or even more complex structures. This is the case for all major syntactic formalisms, including a.o. Minimalism, HPSG, LFG, TAG, and CCG. Even in formal language theory, where many findings focus on the complexity of syntax as a set of well-formed strings (Huybregts, 1984; Kornai, 1985; Shieber, 1985; Radzinski, 1991; Michaelis and Kracht, 1997; Kobele, 2006, a.o.), there is a large body of work that analyzes these strings as the yield of tree structures (e.g. the characterization of multiple context-free string languages as the string yields of MSO-definable tree languages under an MSO tree-to-tree transduction). But even though syntax may well do a lot of work with richly structured objects, this does not entail that this structure is readily accessible to all parts of syntax. To the contrary, recent work in *subregular syntax* (Graf, 2022a,b) suggests that syntactic constraints are so limited that they are better understood as operating over strings, albeit strings that encode linguistic relations like dominance and c-command rather than linear precedence (cf. Frank and Vijay-Shanker, 2001).

For example, Principle A of binding theory requires a reflexive such as *herself* to be c-commanded by a compatible DP like *Mary* or *the woman* within a specific locality domain. As explained in Graf and Shafiei (2019), enforcing Principle A does not require access to the full tree structure as we only need to know the list of c-commanders of the reflexive, which can be represented as a string. Even wh-movement, one of the most fundamental aspects of syntax, can be understood as a constraint that a wh-landing site imposes on its string of *wh-tier daughters* (Graf, 2022a, p.275f). Thus, while syntax may build tree structures for use at the interfaces (meaning, prosody), its constraints appear to be limited to particular types of strings that do not provide nearly as much information as the tree they are obtained from.

This paper puts this general observation on a formal foundation. We introduce the notion of *serial constraints*, which are pairs consisting of a *context function* and a *requirement function* (Sec. 2). The context function con associates every node n of tree t with a string that encodes its syntactic context in t, e.g. its string of ancestors or its string of wh-tier daughters. The requirement function req maps each n to a string language. Then t is well-formed with respect to the serial constraint iff it holds for every node n of t that $con(n) \in req(n)$. We argue that all the proposals put forward in the subregular syntax literature so far are instances of serial constraints (Sec. 2.3–2.5). We then show how serial constraints can be implemented as deterministic tree automata (Sec. 3). For some constraints,

this takes the form of deterministic bottom-up tree automata (Sec. 3.1, 3.2), while for others it takes the form of sensing tree automata, which are deterministic top-down automata with a look-ahead of 1 (Sec. 3.3, 3.4). Despite that difference in directionality, the automata follow a common construction that can be expressed in algebraic terms as a formula of Boolean matrix multiplication steps. These formulas can be tweaked in various ways to define new types of string representations, opening up a novel perspective on subregular automata for syntax.

Our findings have several implications. First of all, our framework provides the first automata-theoretic description of tier-based strictly local tree languages. While there has been a lot of work on tier-based strict locality for strings (Lambert and Rogers, 2020), extending it to trees is not trivial. Since a node can have arbitrarily many tier daughters, one cannot simply store them all in the states of the tree automaton. Our automata construction resolves this challenge and might even provide a new foundation to develop a subregular theory of tree automata. Second, serial constraints formally link two branches of subregular syntax that seem to have been moving in different directions: tree tiers with local constraints VS strings with tier-based strictly local constraints. Our findings reassert the status of subregular syntax as a unified program that furnishes computationally restricted yet linguistically flexible ways of analyzing syntactic phenomena. Finally, the reduction of syntactic constraints from trees to strings opens up new attack vectors for syntactic learning. For example, neural networks could be trained on corpora that lack full tree structures but include relevant c-command relations, encoded as a string.

It is also important to emphasize what this paper is not about. We do not claim that tree structure is redundant for syntax. As mentioned above, the structure-building aspect of syntax seems crucial for prosody and semantic interpretation. Following the two-step approach (Morawietz, 2003; Mönnich, 2006), we regard syntax as the interaction of two components: syntactic constraints that define the set of well-formed structures, and a transductive component that maps syntactic objects to output structures that are used at the PF and LF interfaces. We are not currently aware of any method to reduce the latter to strings, and even subregular work on the transductive component presupposes trees for this (Graf, 2023). But syntactic constraints are amenable to such a reduction and all the methodological simplifications this may provide — as long as strings are built over pertinent syntactic relations rather than linear precedence.

2 Serial constraints for syntax

We take as our starting point recent proposals from subregular syntax (see Graf 2022a,b for a recent overview). In subregular syntax, syntactic structures are feature-annotated dependency trees that encode derivations of a variant of Minimalist grammars (MGs; Stabler, 1997, 2011) where licensee features are unordered (Sec. 2.1; see also appendix A for additional background on MGs and their dependency trees). These syntactic structures are regulated by various subregular constraints, and we define *serial constraints* (Sec. 2.2) as a general mechanism that unifies the many proposals in the subregular literature (Sec. 2.3–2.5). Serial constraints could also be used with other kinds of tree structures, but this paper limits itself to the kind of MG dependency trees used in the subregular literature.

2.1 MG derivations as dependency trees

We treat trees as *labeled Gorn domains* (Gorn, 1967), but for convenience we assume that daughters are numbered from right to left. A *Gorn address* is a string of natural numbers ($s \in \mathbb{N}^*$), including the empty string ε. A *Gorn domain* D is a set of Gorn addresses such that I) $ui \in D$ implies $u \in D$ for all $u \in \mathbb{N}^*$ and $i \in \mathbb{N}$ (mother-of closure), and II) $uj \in D$ implies $ui \in D$ for all $u \in \mathbb{N}^*$, $i, j \in \mathbb{N}$, and $i < j$ (right sibling closure). We occasionally use ux to refer to the unique address $ui \in D$ such that $u \in \mathbb{N}^*$, $i \in \mathbb{N}$, and $u(i+1) \notin D$. A Σ-*tree* is a pair $\langle D, \ell \rangle$ where D is a Gorn domain and $\ell : D \to \Sigma$ the (total) *labeling function*. When clear from context (and particularly in Sec. 3), we use the term *node* to refer to either a Gorn address or its label.

Let Lex be an MG lexicon, i.e. a finite set of lexical items and thus an alphabet. We call a Lex-tree an **MG dependency tree** (MDT) over Lex. Given node u of MDT t, node ui is interpreted as the i-th argument of u (see Fig. 1). Since there is a fixed upper bound j on the number of arguments a lexical item may take, we may assume w.l.o.g. that there is a fixed bound $k \geq j$ such that every Lex-tree is at most k-ary branching. Limited branching is crucial for our automata implementation in Sec. 3.

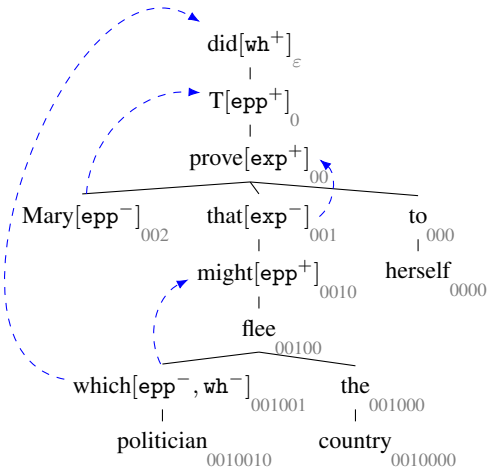


Figure 1: Dependency tree for **Which politician did Mary prove to herself that might flee the country?*, with Gorn address subscripts and dashed movement arrows added as visual aids

Even though MGs make heavy use of movement, all phrases in an MDT remain in their base positions. Movement is indicated via movement features, with the actual displacement left to a post-syntactic transduction step. Negative features (e.g. $\texttt{wh}^-$, $\texttt{epp}^-$) mark the head of the moving phrase, and positive features (e.g. $\texttt{wh}^+$, $\texttt{epp}^+$) mark the head that provides the corresponding landing site. For additional background on the linguistic interpretation of MDTs, the reader is referred to appendix A.

2.2 Serial constraint = context + requirement

We define string-based constraints on trees as the interaction of two functions. The **context function** defines a system for mapping each node n of a tree t to a specific string that is derived from the structural relations of t, relative to n. In some cases, a set Ω of diacritic symbols is used to distinguish multiple structural relations within the string. The **requirement function** then regulates the shape of the string n is mapped to. While all kinds of requirement functions could be considered, the proposals from the subregular literature can be captured with maximally simple ones that constrain the string of n based solely on the label of n. Combining a context function with a requirement function yields a **serial constraint**.

Definition 1. An **Ω-augmented context function** over Σ is a total function con that takes as inputs a Σ-tree $\langle D, \ell\rangle$ and Gorn address $a \in D$ and maps them to a (possibly empty) string $\langle \ell(a_1), \omega_1\rangle \cdots \langle \ell(a_n), \omega_n\rangle$ such that $n \geq 0$ and for all $0 \leq i \leq n$, both $a_i \in D$ and $\omega_i \in \Omega$. If $|\Omega| = 1$, the context function is **unaugmented**. ⌟

To avoid visual clutter, we write $\ell(a)^\omega$ instead of $\langle \ell(a), \omega\rangle$, and we completely omit any mention of Ω for unaugmented context functions. To further increase readability, we use $\cdot$ to explicitly separate the symbols in the outputs of context functions.

Example 1. The (unaugmented) *daughter string* context function drs maps every node to its string of daughters, ordered from left to right. In Fig. 1, $drs(\text{prove}[\texttt{exp}^+])$ is $\text{Mary}[\texttt{epp}^-]\cdot\text{that}[\texttt{exp}^-]\cdot\text{to}$. ⌟

Definition 2. An **Ω-augmented requirement function** over alphabet Σ is a total function $req : \Sigma \to \wp((\Sigma \times \Omega)^*)$ that associates every symbol with a (possibly empty) string language over $\Sigma \times \Omega$. We say that req is **regular** iff $req(\sigma)$ is a regular string language for every $\sigma \in \Sigma$. ⌟

Again we will use superscripts instead of pair notation, and we will omit Ω for unaugmented requirement functions. Hence the co-domain of unaugmented req is simplified to $\wp(\Sigma^*)$.

Example 2. The requirement function *Merge* maps every lexical item to its set of possible argument configurations, each one represented as a string. For example, the transitive verb *eat* is mapped to the set L_{DD} of all strings consisting of exactly two lexical items that each are of category D. Intransitive *eat* would instead require exactly one such D (and thus its image under *Merge* is L_D). If the grammar formalism does not disambiguate between the two, then *eat* is mapped to $L_{DD} \cup L_D$. ⌟

Definition 3. An Ω-augmented **serial constraint** over Σ is a pair $\langle con, req\rangle$ such that con is an Ω-augmented context function over Σ and req is an Ω-augmented requirement function over Σ. A Σ-tree $t := \langle D, \ell\rangle$ is well-formed with respect to $\langle con, req\rangle$ iff it holds for every $a \in D$ that $con(t, a) \in req(\ell(a))$. ⌟

Example 3. Selectional restrictions of lexical items can be regarded as a serial constraint that combines the context function drs over MDTs with the requirement function *Merge*. ⌟

2.3 Types of context functions: a-strings

We now turn to how the various string representations used in the subregular literature (Graf and Shafiei, 2019; Shafiei and Graf, 2020; Graf, 2022a) can be reconceptualized as context functions. We start our discussion with a-strings as they are the most intuitive.

Definition 4 (a-string). Given Gorn address u of MDT $t := \langle D, \ell \rangle$, the **a[ncestor]-string** context function as maps t and u the string of nodes in t that properly dominate u (in top-down order):[1]

$$as(t,u) := \begin{cases} \varepsilon & \text{if } u = \varepsilon \\ as(t,v) \cdot \ell(v) & \text{if } u = vi, i \in \mathbb{N} \end{cases}$$

Example 4. The a-string of which[$\texttt{epp}^-$, $\texttt{wh}^-$] in Fig. 1 is did[$\texttt{wh}^+$]·T[$\texttt{epp}^+$]·prove[$\texttt{exp}^+$] ·that[$\texttt{exp}^-$] ·might[$\texttt{epp}^+$]·flee. ⌟

A-strings can be used to enforce constraints on movement paths. This includes domain conditions like island constraints, but also morphological alternations triggered by movement, e.g. wh-agreement in Irish (McCloskey, 2001; Georgi, 2017; Graf, 2022c).

Example 5. If a subordinate clause is headed by *that*, then its subject cannot be extracted out of this clause. This is known as the *that*-trace effect. This constraint is violated by *which politician* in Fig. 1. We can model this with the context function as in combination with a fairly simple requirement function req. If n carries the subject movement feature $\texttt{epp}^-$ then one of the following must hold: the rightmost complementizer in $as(n)$ is not *that*, or for every movement feature $\texttt{f}^-$ of n, at least one $\texttt{f}^+$ occurs in $as(n)$ to the right of the rightmost complementizer. If n does not include $\texttt{epp}^-$, $req(n)$ is Σ^*. The MDT in Fig. 1 is ill-formed because $as(which[\texttt{epp}^-, \texttt{wh}^-])$ is rejected by req due to the rightmost complementizer being *that* with no $\texttt{wh}^+$ occurring after it. ⌟

2.4 Types of context functions: c-strings

Whereas a-strings are mostly used to capture effects related to movement, c-strings track licensing requirements that are mediated by c-command.

Definition 5 (c-string). Given Gorn address u of MDT $t := \langle D, \ell \rangle$, its **c[ommand]-string** context function cs is recursively defined as[2]

$$cs(t,u) := \begin{cases} \varepsilon & \text{if } u = \varepsilon \\ cs(t,v) \cdot \ell(v) & \text{if } u = vx \\ cs(t,vi) \cdot \ell(vi)^{\leftarrow} & \text{if } u = v(i-1) \end{cases}$$

Example 6. The c-string of which[$\texttt{epp}^-$, $\texttt{wh}^-$] in Fig. 1 is did[$\texttt{wh}^+$]·T[$\texttt{epp}^+$]·prove[$\texttt{exp}^+$] ·Mary[$\texttt{epp}^-$]$^{\leftarrow}$·that[$\texttt{exp}^-$]·might[$\texttt{epp}^+$]·flee.
The c-string of *herself* is did[$\texttt{wh}^+$]·T[$\texttt{epp}^+$] ·prove[$\texttt{exp}^+$]·Mary[$\texttt{epp}^-$]$^{\leftarrow}$·that[$\texttt{exp}^-$]$^{\leftarrow}$·to. ⌟

Intuitively, the c-string of n is obtained by traversing the tree from n towards the root in a leftmost manner, never moving right or down. This approximates the linguistic notion of c-command but does not track how movement may create new c-command relations or destroy existing ones (but we believe that the automata-theoretic view in Sec. 3 furnishes the right tools for addressing this in the future). In addition, c-strings also make an explicit distinction between containing c-commanders (X) and non-containing c-commanders ($X^{\leftarrow}$), which is crucial for some constraints such as Principle A. C-strings are our only instance of such an augmented context function.

Example 7. Consider a simplified version of Principle A: if n is a reflexive, then the smallest TP containing n must contain a DP that c-commands n. In our framework, this means that $cs(t,n)$ must contain some $X^{\leftarrow}$ such that X carries category feature D and occurs to the right of the rightmost T in the string. If n is not a reflexive, the Principle A requirement function PrA puts no restrictions on it (we set $PrA(n) := \Sigma^*$). ⌟

Note that for every node n of any MDT t, $as(t,n)$ is the longest subsequence of $cs(t,n)$ that does not contain any symbols with the superscript $^{\leftarrow}$. In subregular terms, $as(t,n)$ is a *tier* of $cs(t,n)$. It follows that every regular constraint over a-strings can be restated as a regular constraint over c-strings. Hence our automata-theoretic treatment of c-strings in Sec. 3 is also an implicit treatment of a-strings.

2.5 Types of context functions: T-strings

Our last and perhaps most abstract type of strings is defined via tree tiers. Intuitively, a tree tier $T(t)$ of tree t is constructed by fixing a set of node labels,

[1] The definition in Shafiei and Graf (2020) uses a bottom-up order for a-strings, which is formally equivalent but less elegant for our purposes. Moreover, Shafiei and Graf always include $\ell(n)$ in $as(n)$ in order to track which constraints should apply to the string. Since our approach leaves constraint selection to req, including $\ell(n)$ in $as(n)$ is redundant. In fact, factoring out constraint selection reduces the complexity of the string constraints in Shafiei and Graf (2020) from IOTSL to OTSL.

Note that the same differences also hold for the definition of c-strings in Graf and Shafiei (2019) and our Def. 5.

[2] The original definition in Graf and Shafiei (2019) does not include the augmentation symbol $^{\leftarrow}$.

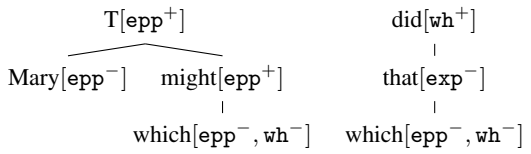


Figure 2: Two tiers of the MDT in Fig. 1: epp-tier (left) and *that*-trace tier (right)

the *tier alphabet* T, and removing from t all nodes that do not belong to T. Figure 2 shows two tiers of the MDT in Fig. 1. The epp-tier is obtained by removing all nodes whose label does not include $\texttt{epp}^+$ or $\texttt{epp}^-$, whereas the *that*-trace tier keeps all instances of the complementizer *that* and all nodes that carry $\texttt{wh}^+$ or $\texttt{wh}^-$. On a tree tier, the label of a node determines the shape that its string of daughters must have. Hence tree tiers are a visual metaphor for a context function that associates every node with its string of tier daughters.

Definition 6 (Tier strings). Given Gorn address u of MDT $\langle D, \ell \rangle$ and $T \subseteq \Sigma$, we say that u is *on* T iff $\ell(u) \in T$. Then u is the *T-mother* of v (and v is a *T-daughter* of u) iff u and v are both on T, $v = uv_1 \cdots v_n$ ($v_i \in \mathbb{N}$, $n \geq 1$), and for all $1 \leq i < n$ it holds that $uv_1 \cdots v_i$ is not on T. Furthermore, w *T-precedes* w' iff w and w' have the same T-mother and there exist $u, v, v' \in \mathbb{N}^*$ and $i, j \in \mathbb{N}$ such that $i > j$, $w = uiv$, and $w' = ujv'$.

The **T-string** context function T maps t and u to the set of T-daughters of u in t, ordered by T-precedence. When u has no T-daughters, $T(n) := \varepsilon$. ⌟

Example 8. Let epp be the alphabet of the epp-tier, which includes all labels that contain $\texttt{epp}^+$ or $\texttt{epp}^-$, and wh the corresponding alphabet for the wh-tier. Then the epp-string of which$[\texttt{epp}^-, \texttt{wh}^-]$ in Fig. 1 is ε, and so is its wh-string. The epp-string of T$[\texttt{epp}^+]$ is Mary$[\texttt{epp}^-]\cdot$might$[\texttt{epp}^+]$. The wh-string of did$[\texttt{wh}^+]$ is which$[\texttt{epp}^-, \texttt{wh}^-]$, whereas its *that*-trace string is that$[\texttt{exp}^-]$. ⌟

Like a-strings, tier strings can be used to enforce island constraints and other conditions on individual movement paths. In contrast to a-strings, they also capture constraints on how distinct movement paths may interact.

Example 9. In MGs, every landing site must be targeted by exactly one mover. This can be captured over tier strings: for every n with movement feature $\texttt{f}^+$ it must be the case that the f-string of n contains exactly one lexical item with $\texttt{f}^-$. A-strings, by contrast, can enforce the presence of a landing site for a mover (if n carries $\texttt{f}^-$, then $as(n)$ must contain $\texttt{f}^+$) but cannot guarantee that this landing site isn't targeted by multiple movers (e.g. C$[\texttt{wh}^+]$ when the subject and object both carry $\texttt{wh}^-$, as neither one appears in the other's a-string).

Daughter strings as defined in example 1 are identical to T-strings with $T = \Sigma$. Hence our automata-theoretic treatment of T-strings also subsumes daughter strings.

3 Tree automata for serial constraints

The previous section has identified several string representations and constraints that have been invoked in the subregular literature, and we have recast all of them as context functions that can be combined with suitable requirement functions. Since the constraints from the subregular literature all define subregular string languages, they can all be captured with regular requirement functions and thus finite-state string automata (FSAs). While requirement functions are easily understood and implemented, then, the formal status of context functions is less clear.

We propose to model context functions as deterministic tree automata whose only purpose is to decide which nodes should be fed as input to the requirement function. These tree automata simulate the (FSAs of the) requirement functions in their state space, while different context functions correspond to minimally different matrix multiplication formulas for updating states. This has the advantage that the tree automaton implicitly produces and evaluates in a single run all n string representations that the corresponding context function would produce for a tree with n nodes. The matrix multiplication formulas also provide a very general template that can be easily adapted to new kinds of string representations.

3.1 Automata for T-string context functions

T-string context functions are implemented as (deterministic) bottom-up tree automata. We present the general template here and provide an illustrative example in Sec. 3.2. Our construction assumes that regular requirement functions are decomposed into FSAs, one per symbol in the alphabet. The FSAs are subsequently decomposed into Boolean matrices. As mentioned in the introduction, this addresses a central challenge of dealing with T-

strings: even when MDTLs are assumed to be at most k-ary branching, there is no upper bound on the number of T-daughters a node may have. Hence one cannot represent the entire string of T-daughters in the states of the tree automata. Instead, one has to store how the T-daughters seen so far would cause the FSA of $req(\sigma)$ to transition between states. The matrix representation of FSAs makes this very easy. Readers unfamiliar with this construction are referred to Appendix B for additional background.

A bottom-up tree automaton is a 4-tuple $\mathfrak{A} := \langle \Sigma, Q, F, \Delta \rangle$ where Σ is an alphabet, Q is a finite set of states, $F \subseteq Q$ is the set of final states, and Δ is a set of transitions. Transitions are of the form $\sigma(q_1, \ldots, q_n) \Rightarrow q$ ($\sigma \in \Sigma$, $q_i \in Q$, $n \geq 0$).[3] Intuitively, the automaton processes trees from the leaves to the root, assigning each node n a state $q \in Q$ based on I) the label of n, and II) the states of n's daughters. The automaton recognizes tree t iff the root of t is assigned some $q \in F$.

Let $\mathcal{B}$ be the Boolean matrix representation of some FSA with $m \geq 1$ states that generates the string language $req(\sigma)$, where $\sigma \in \Sigma$ and req is some regular, Ω-augmented requirement function. We use $\mathbf{I}$ for the initial matrix, $\mathbf{F}$ for the final matrix, $\mathbf{b}(\sigma)$ for the Boolean matrix corresponding to symbol $\sigma \in \Sigma \times \Omega$, and $\mathbf{id}_m$ for the identity element for matrix multiplication of Boolean $m \times m$ matrices. We use $\otimes$ to denote Boolean matrix multiplication.

For every $\sigma \in \Sigma$, we construct a bottom-up tree automaton $\mathfrak{A}_\sigma$ that ensures for every tree t and node n with $\ell(n) = \sigma$ that $T(t, n) \in req(\sigma)$. Intersecting all $\mathfrak{A}_\sigma$ for $\sigma \in \Sigma$ yields a bottom-up tree automaton that enforces requirement function req over T-strings.

Our construction automatically assembles $\mathfrak{A}_\sigma$ from the specification of just two attributes for each node label.

Definition 7 (Node attributes). Let $V, O \subseteq \Sigma$ be the set of **visible** and **opaque** nodes, respectively, and $\sigma \in \Sigma$ the **restricted** node. Then the *value* $\mathbf{v}(n^\omega)$ of $n^\omega \in \Sigma \times \Omega$ is $\mathbf{b}(n^\omega)$ if $n \in V$, and $\mathbf{id}_m$ otherwise. Given a Boolean matrix q, $q \oplus n^\omega$ is $\mathbf{v}(n^\omega)$ if $n \in O$ and $q \otimes \mathbf{v}(n^\omega)$ otherwise. Given initial matrix $\mathbf{I}$ and final matrix $\mathbf{F}$, $\mathbf{r}_n(q)$ is undefined if both $n = \sigma$ and $\mathbf{I} \otimes q \otimes \mathbf{F} = 0$; otherwise $\mathbf{r}_n(q) = q$. ⌟

Intuitively, visible nodes are those that can cause the underlying FSA to transition to a new state. For T-strings, those are simply the nodes that are on the tier. Opaque nodes induce locality domains by overwriting the result of previous matrix multiplications with their own value. For T-strings, every visible node is also opaque. The restricted nodes for $\mathfrak{A}_\sigma$ are exactly those labeled σ, i.e. the ones whose T-string must be well-formed according to the underlying FSA.

Definition 8 (T-string automaton). We define $\mathfrak{A}_\sigma := \langle \Sigma, Q_\mathcal{B}, Q_\mathcal{B}, \Delta \rangle$. Here $Q_\mathcal{B}$ is the result of closing the set of square matrices in $\mathcal{B}$ under Boolean matrix multiplication. For every $n \in \Sigma$ and all $q_1, \ldots, q_k \in Q_\mathcal{B}$ ($k \geq 1$), we set

$$q := \mathbf{r}_n \left(\bigotimes_{i=1}^{k} q_i \right) \oplus \mathbf{v}(n)$$

such that $\sigma(q_1, \ldots, q_k) \Rightarrow q \in \Delta$ iff q is defined. Furthermore $\sigma() \Rightarrow \mathbf{v}(\sigma) \in \Delta$ for every $\sigma \in \Sigma$. ⌟

Since the formula above yields at most one value for q, $\mathfrak{A}_\sigma$ is deterministic even if $\mathcal{B}$ is the Boolean representation of a non-deterministic FSA. Note that $Q_\mathcal{B}$ and Δ are automatically constructed from σ, Σ, $\mathcal{B}$, and the attributes V and O. Also, all states of $\mathfrak{A}_\sigma$ are final (a tree is rejected iff there is a node that no state can be assigned to). Hence $\mathfrak{A}_\sigma$ could instead be defined as a 5-tuple $\langle \Sigma, \sigma, \mathcal{B}, V, O \rangle$.

3.2 Example: epp-string requirement function

Example 9 mentions that MGs require every landing site to be targeted by exactly one mover. If a node carries the feature $\texttt{epp}^+$, then its epp-string must contain exactly one node that carries $\texttt{epp}^-$. We can think of this as a requirement function $\mathbf{1}$ that maps each lexical item to one of two regular string languages. If l does not carry $\texttt{epp}^+$, then $\mathbf{1}(l)$ is Σ^*; otherwise, $\mathbf{1}(l)$ is $L_\mathbf{1} := \overline{E}^* E \overline{E}^*$ where $E \subseteq \Sigma$ is the set of lexical items that carry $\texttt{epp}^-$ and $\overline{E} := \Sigma - E$. It is easy to define an FSA for $L_\mathbf{1}$, which is then decomposed into its Boolean representation $\mathcal{B}$ (see Fig. 3).

We now construct $\mathfrak{A}$ for a single lexical item, which is $might[\texttt{epp}^+]$. The epp-tier contains all items that carry $\texttt{epp}^+$ or $\texttt{epp}^-$, so all of those are visible and opaque. Figure 4 shows the states assigned by the automaton as well as the attributes of

[3] In the tree automata literature, it is more common to write the transition rules in the format $\sigma(q_1(x_1), \ldots, q_n(x_n)) \Rightarrow q(\sigma(x_1, \ldots, x_n))$, with each x_i a variable representing a subtree (see Gécseg and Steinby 1997 and Comon et al. 2008, p.20). We omit these variables to reduce clutter.

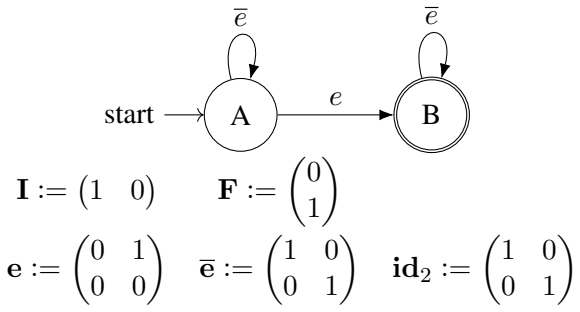


Figure 3: FSA for L_1 and corresponding Boolean matrices, with $e/\overline{e}$ as a shorthand for every lexical item on the epp-tier that does/doesn't carry $\texttt{epp}^-$.

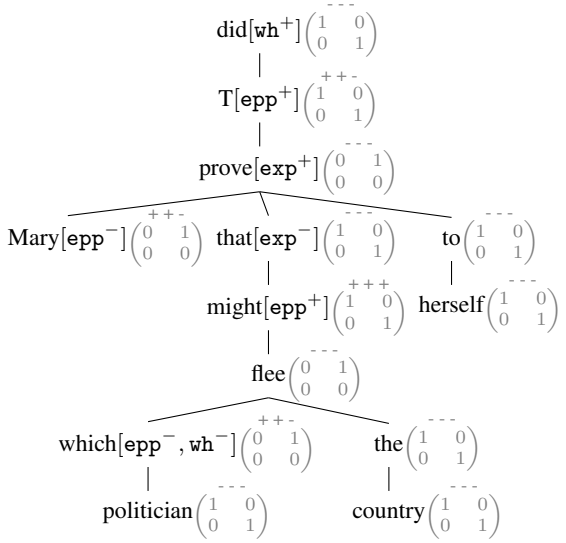


Figure 4: Run of automaton enforcing L_1 over epp-strings of $might[\texttt{epp}^+]$; $+$ and $-$ indicate whether the node is visible, opaque, and/or restricted

each node (in the order *visible-opaque-restricted*). The tree is correctly recognized as well-formed with respect to the epp-string requirement function because the root is assigned a state (remember that all states are final).

Let us consider a few specific nodes from Fig. 4. The leaf node *politician* is not visible, so its value is the 2×2 identity matrix $\mathbf{id}_2$. Since it is a leaf, we use the transition $\sigma() \Rightarrow \mathbf{v}(\sigma)$, for which it is irrelevant whether *politician* is opaque or restricted. Now consider *which*, right above *politician*. It is visible and opaque, but not restricted. Since it is visible and carries $\texttt{epp}^-$, its value is the Boolean matrix $\mathbf{e}$. Since it isn't restricted, $\mathbf{r}_n(\bigotimes_{i=1}^{k} q_i)$ is not undefined, but since it is opaque its state is just its value $\mathbf{e}$. The node *flee* above *which* is neither visible nor opaque or restricted. Hence its state is the result of matrix multiplying the states of *which* and *the* with its value $\mathbf{id}_2$. Finally, *might* is visible, opaque, and restricted. Its value is the Boolean matrix $\overline{\mathbf{e}}$, which is identical to $\mathbf{id}_2$ in this case. As *might* is restricted, $r_n(\bigotimes_{i=1}^{k} q_i)$ could be undefined. But fortunately multiplying $\mathbf{I}$ with the state of *flee* and $\mathbf{F}$ yields 1 (confirming that the epp-tier string of *might* is a member of L_1). If the result had been 0, no state would have been assigned and the computation would have halted, causing the tree to be rejected. Instead, *might* is assigned its value as its state because it is opaque. The computation continues from there, but since no other nodes in the tree are restricted, we are guaranteed to assign some state to the root (which is final because all states are final).

By constructing such an automaton for every lexical item with $\texttt{epp}^+$, we ensure that every lexical item with $\texttt{epp}^+$ has exactly one $\texttt{epp}^-$ among its epp-tier daughters.

3.3 Automata for c-string context functions

We now turn to the implementation of c-strings (which subsumes a-strings as discussed at the end of Sec. 2.4). Instead of bottom-up automata, we will use *sensing tree automata* as these have previously been proposed by Graf and De Santo (2019) as a model of c-string constraints.[4] For convenience, we use a slightly different notation for defining the transitions of these automata, and we allow the initial state to be determined by the label of the root. These changes will make it easier to see that the state assignment template for sensing tree automata is almost the same as for bottom-up tree automata. In particular, the attributes and operations from Def. 7 carry over unaltered. This shows that our treatment of context functions is independent of the specific types of tree automata.

A sensing tree automaton is a 3-tuple $\mathfrak{A} := \langle \Sigma, Q, \delta \rangle$ where Σ is an alphabet, Q is a finite set of states, and δ is a set of transition rules that may take two distinct forms. For interior nodes, we have $\langle q, \sigma(\sigma_1, \ldots, \sigma_k), i \rangle \Rightarrow q_i$ $(1 \leq i \leq k)$. This means that if σ_i has mother σ with state q, left siblings $\sigma_1, \ldots, \sigma_{i-1}$, and right siblings $\sigma_{i+1}, \ldots, \sigma_k$, then σ_i is assigned state q_i. For root nodes, the transition $\sigma \Rightarrow q$ assigns state q to σ. Intuitively, sensing tree automata assign states in a top-down

[4] Sensing tree automata cannot be used to capture tier-string constraints. As noted in Graf and De Santo (2019), sensing tree automata cannot regulate movement steps that aren't restricted by both the specifier island constraint and the adjunct island constraint. For example, a sensing tree automaton cannot ensure that every $\texttt{epp}^+$ is targeted by exactly one $\texttt{epp}^-$. But as we just saw, this is easily enforced over epp-tier strings.

fashion and make the assigned state contingent on the mother's state and the labels of the node, its siblings, and its mother. Since sensing tree automata are deterministic, δ must not contain distinct transitions rules with the same left-hand side. A tree t is accepted by $\mathfrak{A}$ iff every node of t is assigned a state.

With these preliminaries out of the way, it is easy to define the sensing tree automaton $\mathfrak{A}_\sigma$ for some requirement function req. As before, $\mathcal{B}$ is the Boolean representation of an FSA with $m \geq 1$ states that generates $r(\sigma)$, and $Q_\mathcal{B}$ is the result of closing the set of square matrices in $\mathcal{B}$ under Boolean matrix multiplication.

Definition 9. We define $\mathfrak{A}_\sigma := \langle \Sigma, Q_\mathcal{B}, \delta \rangle$. For every state $q \in Q$ and all $1 \leq j \leq k$ and $\sigma, \sigma_1, \ldots, \sigma_k \in Q_\mathcal{B}$, we set

$$q_j := \mathbf{r}_{\sigma_j}\left(q \oplus \bigoplus_{i=1}^{j-1} \mathbf{v}(\sigma_i^{\leftarrow})\right) \oplus \mathbf{v}(\sigma_j)$$

such that $\langle q, \sigma(\sigma_1, \ldots, \sigma_j, \ldots, \sigma_k), j \rangle \Rightarrow q_j \in \delta$ iff q_j is defined. Furthermore, $\sigma \Rightarrow \mathbf{v}(\sigma)$ for every $\sigma \in \Sigma$.

The formula for c-strings differs only marginally from T-strings, namely in the argument of $\mathbf{r}$. Quite generally, this is the area where differences between context functions are expressed. To wit, a-strings would also differ in only this area by simplifying the argument of $\mathbf{r}$ to just q. The specific differences between c-strings and T-strings are due to siblings taking on a similar role in c-strings to ancestors in T-strings. With c-strings, an opaque node renders inaccessible all information about its left siblings, and thus the values of siblings have to be combined with $\oplus$ instead of $\otimes$. Also, each sibling σ_i is a c-commander and hence its value must be $\mathbf{v}(\sigma_i^{\leftarrow})$ rather than $\mathbf{v}(\sigma_i)$. These minor changes in the formulas cannot distract from the fact, however, that a-strings, c-strings and T-strings (which includes daughter strings) have remarkably similar automaton implementations.

3.4 Example: Binding and (reduced) c-strings

As with T-strings, we provide a linguistic example of the automaton construction for c-strings. Consider once more the simplified version of Principle A from example 7: if n is a reflexive, then $cs(n)$ must have a non-containing D-head $\mathrm{D}^{\leftarrow}$to the right of the rightmost containing T-head.

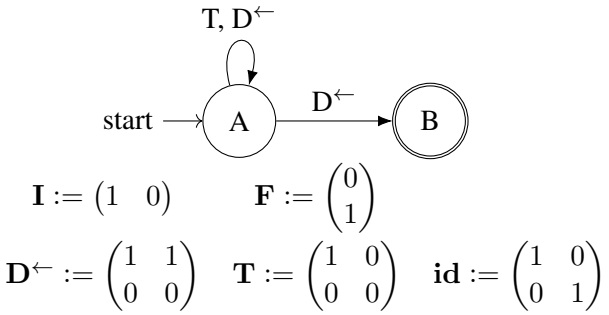


Figure 5: FSA and Boolean matrices for Principle A over reduced c-strings that only contain $\mathrm{D}^{\leftarrow}$ and T.

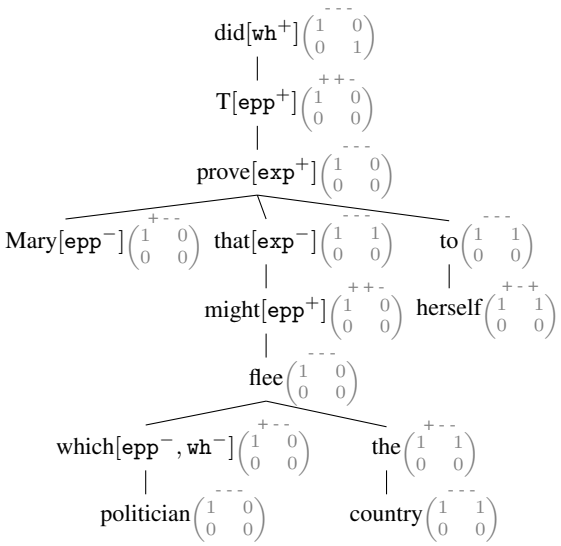


Figure 6: Run of automaton enforcing L_A over reduced c-strings that only contain D and T; + and − indicate whether the node is visible, opaque, and/or restricted (D-heads are visible only when they are non-containing c-commanders, and T-heads are visible only when they are containing c-commanders)

In order to make the example more insightful, we will implement this as a tree automaton that constructs a reduced version of the c-string that only contains Ds and Ts, ignoring all c-commanders that are immaterial to this constraint. That is to say, the task of ignoring irrelevant nodes is shifted from the requirement function into the context function. Hence the requirement function maps reflexives to the string language $L_A := \{\mathrm{T}, \mathrm{D}^{\leftarrow}\}^* \ \mathrm{D}^{\leftarrow}$ (see Fig. 5). The set of visible nodes consists of all $\mathrm{D}^{\leftarrow}$ and all T (but not D or $\mathrm{T}^{\leftarrow}$). The only opaque nodes are visible T-heads, and the only restricted node is *herself*.

The run of the resulting sensing tree automaton is shown in Fig. 6. Since every node is assigned a state, the tree is correctly recognized as well-formed. The important thing to keep in mind

is that each node n may now exhibit a dual behavior depending on whether the formula uses $\mathbf{v}(n)$ or $\mathbf{v}(n^{\leftarrow})$. For example, the state of *Mary* is computed with $\mathbf{v}(\textit{Mary}) = \mathbf{id}_2$, and hence it is identical to the state of its mother *prove*. On the other hand, the state of *that* is computed with $\mathbf{v}(\textit{Mary}^{\leftarrow}) = \mathbf{D}^{\leftarrow}$, inducing a state change. The same effect obtains with the state of *which* relative to *the*. Also note how *might*, by virtue of being opaque, receives the state $\mathbf{v}(\textit{might}) = \mathbf{T}$ and thus renders *Mary* inaccessible from within that subtree.

4 Conclusion

All syntactic constraints that have been put forward in the subregular literature can be analyzed as serial constraints. Serial constraints consists of a context function that associates every node in a tree with a string derived from the tree, and a (regular) requirement function that requires the node's associated string to belong to a (regular) string language. While requirement functions are fairly unremarkable from a formal perspective, context functions require additional considerations.

This paper shows that the context functions for T-strings (and by extension daughter strings) as well as c-strings (and by extension a-strings) can all be implemented as tree automata that follow a universal template. Crucially, the states of these automata store no information beyond what is needed to simulate the requirement function. All other decisions are made based only on the information available directly in each transition rule: whether a node is visible, opaque, and/or restricted. How exactly this information is used to compute states can be succicntly expressesd via matrix multiplication formulas. Each such formula is of the form $q := \mathbf{r}_{\sigma}(\phi) \oplus \mathbf{v}(\sigma)$, where ϕ is a Boolean matrix computed from the states and/or the values of nodes accessible in the transition rule. The general upshot is that even though selectional constraints and constraints on tree tiers seem intuitively different from a-string and c-string constraints, they are but minor variations of a common theme.

One surprising implication of these findings is that (most, perhaps even all) syntactic constraints can be regarded as operating over strings rather than trees. All the regulating work is done by the requiremnt function (which is an FSA), with tree automata serving as a simple wrapper that passes information into this function. Of course this may be due to serial constraints providing a lot more power than it currently seems. For example, even the requirement that a tree must have an odd number of nodes can be implemented as a serial constraint (e.g. via a preorder traversal). On the other hand, it seems that no serial constraint can express the requirement that a tree must contain an even number of nodes that each properly dominate at least two nodes. Further work is needed to properly assess the power of serial constraints and how it varies with the chosen automaton model.

Acknowledgements

The work carried out for this project was supported by the National Science Foundation under Grant No. BCS-1845344. Kenneth Hanson received additional funding from the Institute for Advanced Computational Science at Stony Brook University. We are grateful to three anonymous reviewers for their detailed feedback and suggestions. Reviewer 3 in particular went far beyond the call of duty, and their thoughtful observations will be very useful for future iterations of this work.

References

H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. TommasiK. 2008. Tree automata: Techniques and applications. Published online: `http://www.grappa.univ-lille3.fr/tata`. Release from November 18, 2008.

Robert Frank and K Vijay-Shanker. 2001. Primitive c-command. *Syntax*, 4(3):164–204.

Doreen Georgi. 2017. Patterns of movement reflexes as the result of the order of merge and agree. *Linguistic Inquiry*, 48:585–626.

Saul Gorn. 1967. Explicit definitions and linguistic dominoes. In *Systems and Computer Science, Proceedings of the Conference held at University of Western Ontario, 1965*, Toronto. University of Toronto Press.

Thomas Graf. 2022a. Diving deeper into subregular syntax. *Theoretical Linguistics*, 48:245–278.

Thomas Graf. 2022b. Subregular linguistics: Bridging theoretical linguistics and formal grammar. *Theoretical Linguistics*, 48:145–184.

Thomas Graf. 2022c. Typological implications of tier-based strictly local movement. In *Proceedings of the Society for Computation in Linguistics (SCiL) 2022*, pages 184–193.

Thomas Graf. 2023. Subregular tree transductions, movement, copies, traces, and the ban on improper

movement. In *Proceedings of the Society for Computation in Linguistics (SCiL) 2023*, pages 289–299.

Thomas Graf and Aniello De Santo. 2019. Sensing tree automata as a model of syntactic dependencies. In *Proceedings of the 16th Meeting on the Mathematics of Language*, pages 12–26, Toronto, Canada. Association for Computational Linguistics.

Thomas Graf and Nazila Shafiei. 2019. C-command dependencies as TSL string constraints. In *Proceedings of the Society for Computation in Linguistics (SCiL) 2019*, pages 205–215.

Ferenc Gécseg and Magnus Steinby. 1997. Tree languages. In Gregorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 1–68. Springer, New York.

Riny Huybregts. 1984. The weak adequacy of context-free phrase structure grammar. In Ger J. de Haan, Mieke Trommelen, and Wim Zonneveld, editors, *Van Periferie naar Kern*, pages 81–99. Foris, Dordrecht.

Gregory M. Kobele. 2006. *Generating Copies: An Investigation into Structural Identity in Language and Grammar*. Ph.D. thesis, UCLA.

Andras Kornai. 1985. Natural language and the Chomsky hierarchy. In *Proceedings of the EACL 1985*, pages 1–7.

Dakotah Lambert and James Rogers. 2020. Tier-based strictly local stringsets: Perspectives from model and automata theory. In *Proceedings of the Society for Computation and Linguistics*, volume 3, pages 330–337.

James McCloskey. 2001. The morphosyntax of wh-extraction in Irish. *Journal of Linguistics*, 37:67–100.

Jens Michaelis and Marcus Kracht. 1997. Semilinearity as a syntactic invariant. In *Logical Aspects of Computational Linguistics*, volume 1328 of *Lecture Notes in Artifical Intelligence*, pages 329–345. Springer.

Uwe Mönnich. 2006. Grammar morphisms. Ms. University of Tübingen.

Frank Morawietz. 2003. *Two-Step Approaches to Natural Language Formalisms*. Walter de Gruyter, Berlin.

Daniel Radzinski. 1991. Chinese number names, tree adjoining languages, and mild context sensitivity. *Computational Linguistics*, 17:277–300.

Nazila Shafiei and Thomas Graf. 2020. The subregular complexity of syntactic islands. In *Proceedings of the Society for Computation in Linguistics (SCiL) 2020*, pages 272–281.

Stuart M. Shieber. 1985. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8(3):333–345.

Edward P. Stabler. 1997. Derivational Minimalism. In Christian Retoré, editor, *Logical Aspects of Computational Linguistics*, volume 1328 of *Lecture Notes in Computer Science*, pages 68–95. Springer, Berlin.

Edward P. Stabler. 2011. Computational perspectives on Minimalism. In Cedric Boeckx, editor, *Oxford Handbook of Linguistic Minimalism*, pages 617–643. Oxford University Press, Oxford.

A Background: MGs and tree structures

Every MG is fully specified by its *lexicon*, which is a finite set of feature-annotated lexical items that are combined by the structure-building operations Merge and Move. Merge features encode the lexical item's category (*category feature* $\texttt{F}^-$) and its subcategorization requirements (*selector features* $\texttt{F}^+$). Move features indicate whether the lexical item furnishes any landing sites for movement (*licensor features* $\texttt{f}^+$), and whether it undergoes any movement of its own (*licensee features* $\texttt{f}^-$). To reduce clutter, we only indicate Move features throughout this paper. Some variants of MGs allow licensor features to indicate whether the landing site is linearized to the left or to the right. We allow this option in this paper and use it for extraposition of the *that*-clause in Fig. 1, but nothing hinges on that.

As is common in subregular syntax, but unlike standard MGs, we assume that licensee features are unordered (this has no effect on generative capacity). For example, a lexical item with both $\texttt{epp}^-$ and $\texttt{wh}^-$ has to undergo both epp-movement and wh-movement, but the order is unspecified. If the closest epp-landing site is closer than the closest wh-landing site, epp-movmeent will precede wh-movement, otherwise it will follow it. When a lexical item undergoes movement, it does not move by itself but moves along the entire phrase it is the head of.

MG derivations can be represented as dependency trees. The mother-of relation corresponds to Merge steps, and the right-to-left order of siblings matches the order in which they are merged with the mother. For example, *flee* in Fig. 7 first merges with (the phrase headed by) *the*, taking it as a complement. After that, (the phrase headed by) *which* is merged as a specifier. Movement is only indicated by licensor and licensee features — movers are not displaced from their base position. A mover with $\texttt{f}^-$ always targets the cloest landing site from its base position that is provided by a matching $\texttt{f}^+$.

While MG dependency trees look different from

standard phrase structure trees, they encode all necessary syntactic information. However, they do so much more compactly than more common alternatives such as X′-trees (cf. Fig.7).

B Background: FSAs as Boolean matrix multiplication

An FSA is a 5-tuple $A := \langle \Sigma, Q, I, F, \Delta \rangle$ where Σ is the alphabet, Q is a finite set of states, $I \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of final states, and Δ is a finite set of transition rules of the form $q \overset{\sigma}{\Rightarrow} q'$. We assume w.l.o.g. that Q is not empty.

The corresponding **Boolean representation** is constructed as follows. First, we fix an arbitrary enumeration of all $n \geq 1$ states of Q. Then every $\sigma \in \Sigma$ is associated with a $n \times n$ matrix $\mathbf{b}(\sigma)$ such that the cell $\mathbf{b}(\sigma)_{i,j}$ in row i, column j ($1 \leq i, j \leq n$) is 1 if Δ contains the transition $q_i \overset{\sigma}{\Rightarrow} q_j$. Otherwise, the cell is 0. The *initial matrix* $\mathbf{I}$ is a $1 \times n$ matrix such that $\mathbf{I}_{1,j}$ is 1 if $q_j \in I$ and 0 otherwise. Similarly, the *final matrix* $\mathbf{F}$ is a $n \times 1$ matrix such that $\mathbf{F}_{i,1}$ is 1 if $q_i \in F$ and 0 otherwise.

Example 10. The smallest deterministic FSA over $\Sigma := \{a, c\}$ that recognizes $a(aa)^*$ corresponds to the matrices below.

$$\mathbf{I} := \begin{pmatrix} 1 & 0 \end{pmatrix} \qquad \mathbf{F} := \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$
$$\mathbf{b}(a) := \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \mathbf{b}(c) := \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

⌟

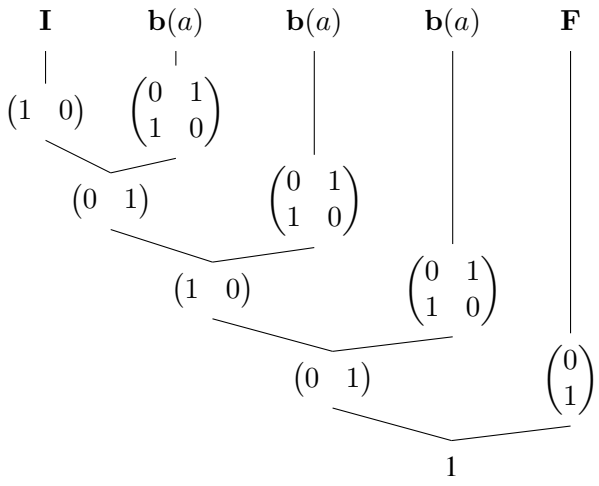
A string $\sigma_1 \cdots \sigma_k$ is recognized by A iff $\mathbf{I} \otimes \mathbf{b}(\sigma_1) \otimes \cdots \otimes \mathbf{b}(\sigma_k) \otimes \mathbf{F} = 1$, where $\otimes$ denotes Boolean matrix multiplication. Given a Boolean $u \times v$ matrix A and $v \times w$ matrix B, $A \otimes B$ is a $u \times w$ matrix C such that for all $1 \leq i \leq u$ and $1 \leq j \leq w$

$$C_{i,j} := \bigvee_{k=1}^{v} (A_{i,k} \wedge B_{k,j})$$

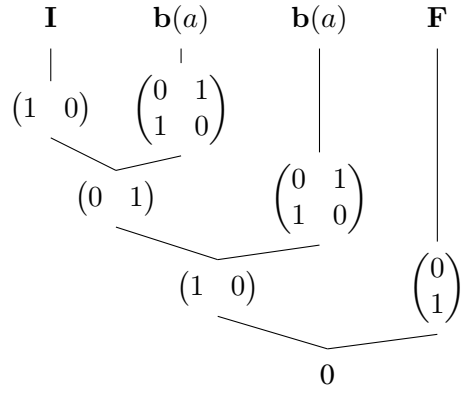
Example 11. Continuing the previous example, the FSA recognizes aaa as we have

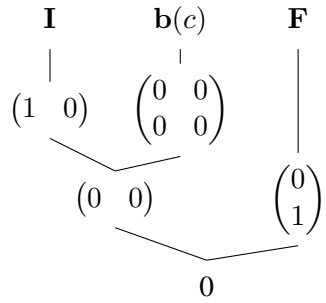
$$\mathbf{I} \otimes \mathbf{b}(a) \otimes \mathbf{b}(a) \otimes \mathbf{b}(a) \otimes \mathbf{F} = 1$$

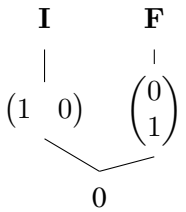
which can be gleaned from the following tree:



But the FSA rejects aa, c, and the empty string:







⌟

The identity matrix $\mathbf{id}_n$ of size n is the $n \times n$ square matrix such that $\mathbf{id}_{i,j} = 1$ if $i = j$, and 0 otherwise. When M is an $m \times n$ matrix, $\mathbf{id}_m \otimes M = M \otimes \mathbf{id}_n = M$.

Example 12. Multiplying $\mathbf{b}(a)$ with $\mathbf{id}_2$ yields

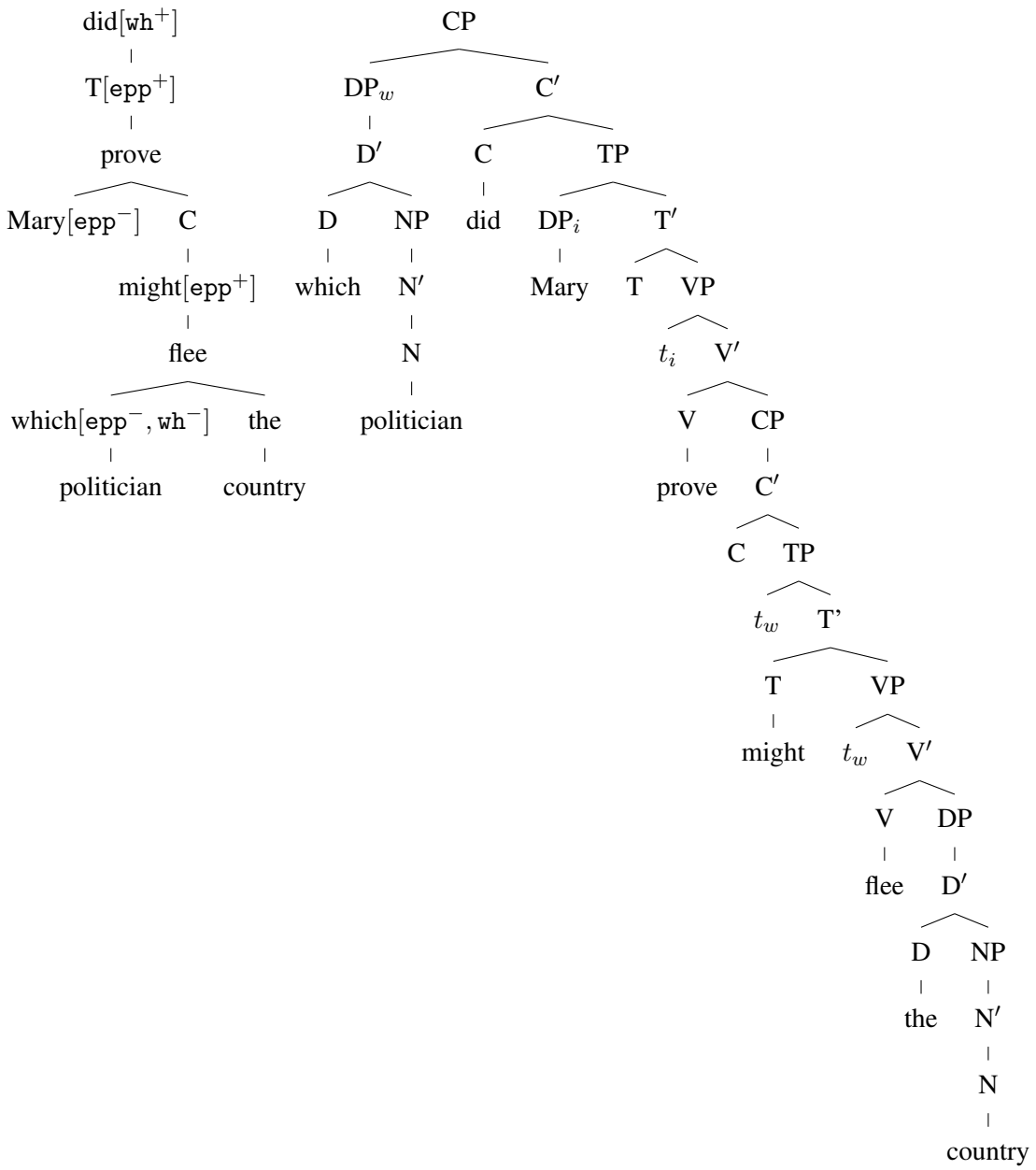


Figure 7: MG dependency tree and corresponding X′-tree for *Which politician did Mary prove might flee the country*

$\mathbf{b}(a)$.

$$\begin{aligned}
\mathbf{b}(a) \times \mathbf{id}_2 &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \\
&= \mathbf{id}_2 \otimes \mathbf{b}(a)
\end{aligned}$$

⌟