Jennifer Bellik* and Nick Kalivoda

Automated tableau generation using SPOT (Syntax Prosody in Optimality Theory)

https://doi.org/10.1515/lingvan-2017-0051
Received April 6, 2018; accepted July 12, 2018

Abstract: Much recent work on the syntax-prosody interface has been based in Optimality Theory. The typical analysis explicitly considers only a small number of candidates that could reasonably be expected to be optimal under some ranking, often without an explicit definition of GEN. Manually generating all the possible candidates, however, is prohibitively time-consuming for most input structures – the Too Many Candidates Problem. Existing software for OT uses regular expressions for automated generation and evaluation of candidates. However, regular expressions are too low in the Chomsky Hierarchy of language types to represent trees of arbitrary size, which are needed for syntax-prosody work. This paper presents a new computational tool for research in this area: Syntax-Prosody in Optimality Theory (SPOT). For a given input, SPOT generates all prosodic parses under certain assumptions about GEN, and evaluates them against all constraints in CON. This allows for in-depth comparison of the typological predictions made by different theories of GEN and CON at the syntax-prosody interface.

Keywords: syntax-prosody interface; Optimality Theory; Computational Tools in Phonology; prosody; OTWorkplace.

1 Introduction

Recent work on the syntax-prosody interface (Truckenbrodt 1999; Selkirk 2011; Elfner 2012; Ito and Mester 2013; Myrberg 2013, among many others) has fruitfully employed violable constraints within the framework of Optimality Theory (OT; Prince and Smolensky 2004 [1993]). An OT system consists of two functions, GEN (=generator) and CON (=constraints). In practice, most research has focused on CON – developing a constraint set that successfully models the phenomenon of interest. GEN has received less attention. The typical analysis explicitly considers only a small number of candidates that could reasonably be expected to be optimal under some ranking, often without an explicit definition of GEN. This is a serious problem because omitting candidates can render an analysis invalid (Bane and Riggle 2012).

Manually generating all the possible candidates, however, is prohibitively time-consuming for most input structures – the Too Many Candidates Problem. Although this is also true in other domains, such as metrical phonology (Karttunen 2006), syntax-prosody mapping poses an additional challenge to automated generation and evaluation of candidates, because both inputs and outputs take the form of trees. Existing software for automated generation and evaluation of metrical candidates, such as OTWorkplace (Prince et al. 2017) and PyPhon (Bane and Riggle 2010), uses regular expressions, which are too low in the Chomsky Hierarchy of language types to represent trees of arbitrary size.

To enable automatic generation and evaluation of tree structures, we have developed the JavaScript application SPOT (Bellik et al. 2017, available at https://github.com/syntax-prosody-ot). SPOT is useful for any constraint-based analysis of tree structures. Automatic tableau generation like that provided by SPOT is particularly important for typological investigations. The built-in constraints are designed for OT work on the syntax-prosody interface, but since the output is a violation tableau that can be manipulated using other tools, the SPOT framework is readily extensible to Harmonic Grammar or OT approaches to syntax.

***Corresponding author: Jennifer Bellik,** Department of Linguistics, University of California Santa Cruz, 1156 High Street, Santa Cruz, CA 95064, USA, E-mail: jbellik@ucsc.edu
Nick Kalivoda: Department of Linguistics, University of California Santa Cruz, 1156 High Street, Santa Cruz, CA 95064, USA

This paper illustrates the exponential increase in the number of prosodic output candidates as the number of terminals in the syntactic string increases (Section 2). Section 3 presents SPOT's implementation of GEN, an algorithm that generates all the trees with three prosodic categories and recursion at one prosodic level. Section 4 describes CON in SPOT, and points out the computational ambiguity of many constraint definitions in the syntax-prosody literature. Section 5 provides a case study of branchingness effects in Kinyambo phrasing, along with a step-by-step illustration of how to use SPOT for a syntax-prosody investigation. Section 6 concludes.

2 The Too Many Candidates Problem

The syntax-prosody interface is tasked with mapping a syntactic input tree to a prosodic output, as in Figure 1. Early approaches to the syntax-prosody mapping required prosodic outputs to conform to Strict Layering (Selkirk 1984; Nespor and Vogel 1986, though see Ladd 1986; Ladd 1988 for an early opposing view).

(1) Strict Layering
   a. Layeredness: Every child of a node of category C is of category $\leq$ C $-$ 1.
      E.g., every child of an ι is of category $\leq$ φ, every child of a φ is of category $\leq$ ω.
   b. Non-recursivity: No child of a node of category C is of category C.
      E.g., no child of an ι is an ι, no child of a φ is a φ, and no child of a ω is a ω.
   c. Exhaustivity: Every child of a node of category C is of category $\geq$ C $-$ 1.
      E.g., every child of a ι is of category φ or higher, and every child of a φ is of category ω or higher.

Following much recent work (e.g., Selkirk 2011; Ito and Mester 2013; Lee and Selkirk 2015), SPOT assumes the prosodic categories ι (intonational phrase), φ (phonological phrase), and ω (prosodic word), in the hierarchy depicted in Figure 2. In addition, we refer to the input syntactic words as lexical terminals (*xyz* in Figure 2). This is to draw a distinction between the segmental content of a morpheme, which is lexically specified, and its status as a prosodic word, which generally comes about in the mapping to prosodic structure (although see, e.g., Nespor and Vogel 1986; Zec 2005 for some exceptions). We refer to the ordered tuple of lexical terminals in a tree as its terminal string.

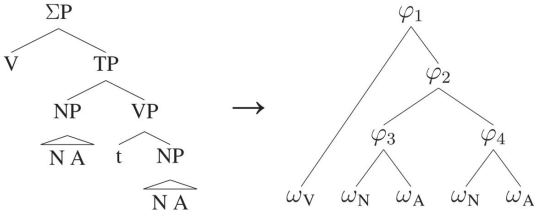

Figure 1: Syntax-prosody mapping example (from Elfner 2012 on Irish).

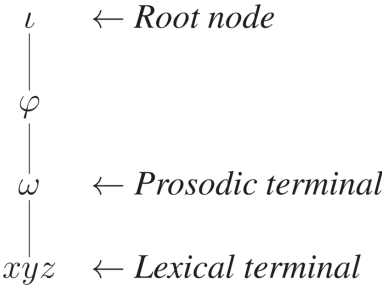

Figure 2: Prosodic hierarchy.

Only two prosodic parses (Figure 3) of a string of two lexical terminals conform to Strict Layering with the prosodic hierarchy in Figure 2. However, even under Strict Layering, the number of prosodic candidates increases exponentially with the number of lexical terminals (Figure 4); in fact, the number of Strict Layering prosodic candidates is $2^{(n-1)}$, where n is the number of syntactic words.[1] Since sentences in syntax-prosody investigations often contain six or more words, constructing all the prosodic candidates becomes tedious.

More recent work on the syntax-prosody interface has motivated a relaxing of the Strict Layering requirements, resulting in several versions of Weak Layering (Selkirk 1995; Ito and Mester 1992/2003). Weak Layering allows prosodic parses that violate one or more of the properties in (2) and would be ruled out under Strict Layering.

(2) Violable properties in Weak Layering

a. Non-recursivity: No child of a node of category C is of category C.
E.g., no child of an ι is an ι, no child of a φ is a φ, and no child of a ω is a ω.

b. Exhaustivity: Every child of a node of category C is of category $\geq$ C $-$ 1.
E.g., every child of a ι is of category φ or higher, and every child of a φ is of category ω or higher.

c. Headedness:[2] Any node of category C must dominate a node of category C $-$ 1.
E.g., every ι dominates at least one φ, and every φ dominates at least one ω.

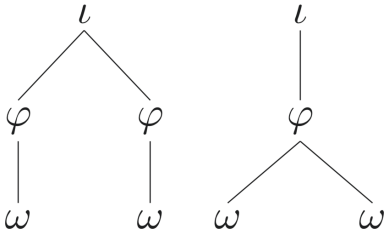

Figure 3: Strict Layering two-word trees in SPOT.

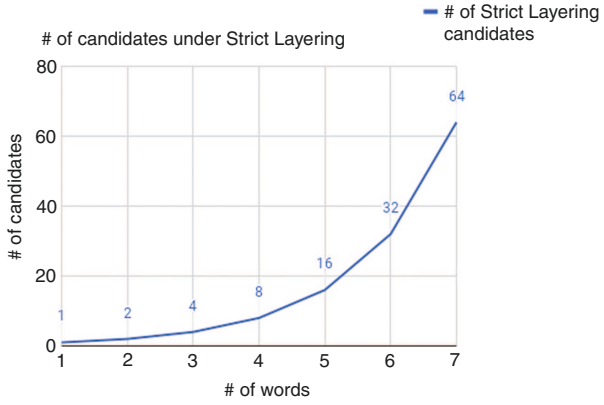

Figure 4: Number of prosodic parses under Strict Layering.

1 This is because Strictly Layered trees in SPOT have only three levels (since there are three prosodic categories and no recursion is allowed), and only differ at the middle level of phonological phrases, such that making a Strictly Layered tree consists of deciding whether to put a phi-boundary between each pair of adjacent words or not. That is, it consists of a decision between two alternatives (boundary/no boundary), made $n-1$ times.

2 Headedness is not uniformly considered violable by proponents of Weak Layering, but has been used as a violable constraint by Schiering et al. (2010). In addition, this makes SPOT's GEN more analogous to OTWorkplace's GEN for feet and metrical stress, which makes Headedness an option.

These relaxed requirements result in a dramatically increased candidate set, exacerbating the Too Many Candidates Problem. For example, if recursive structures, level-skipping, and unheaded intonational phrases are allowed, then there are 8 possible prosodic parses for a string of only two syntactic terminals (Figure 5).

Under Weak Layering, the exponential growth of the number of candidates becomes completely unmanageable for manual generation of all candidates (Figure 6). When a five-word sentence can have over 6000 parses, clearly the analyst cannot generate them all by hand. This "combinatorial explosion" is even more dramatic than the ones discussed for binary branching trees in, e.g., Wagner (2010) and Langendoen (1987), since supra-binary branching structures are considered. In fact, the number of Weak Layering trees with n terminals corresponds to the number of graphs with $n + 1$ nodes on a circle without crossing edges (OEIS A054726); the number of nodes on the circle corresponds to the number of potential positions for a φ-boundary (Ozan Bellik, p.c.). This sequence is related to the little Schröder numbers, or Super Catalan numbers (OEIS A001003; cf. Wagner 2010) but without the restriction that at least two items must be grouped together within each set of parentheses – i.e., unary branching φs are allowed.

SPOT addresses the Too Many Candidates Problem by automating GEN. The algorithm for GEN and its parameters are presented in the next section.

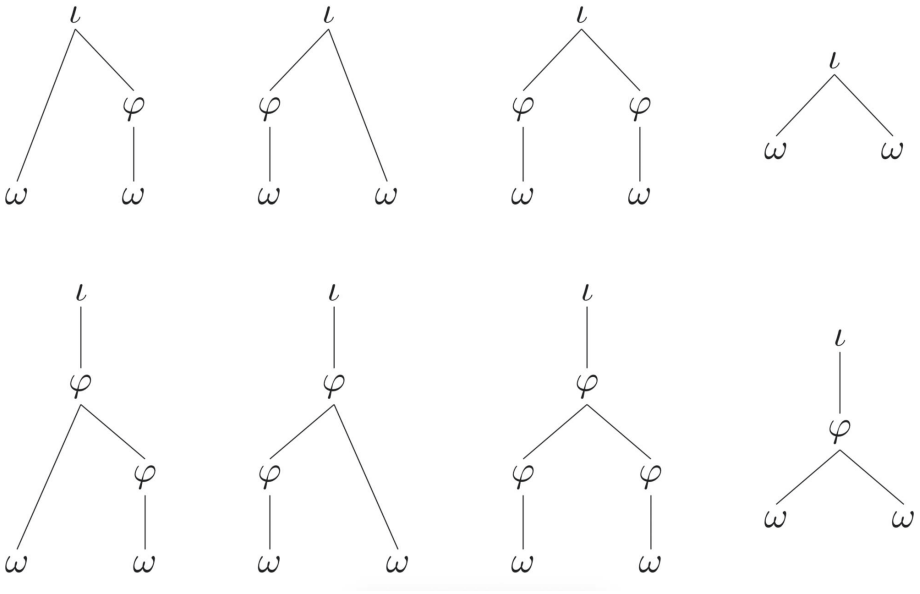

Figure 5: Prosodic parses of two-word terminal string under Weak Layering.

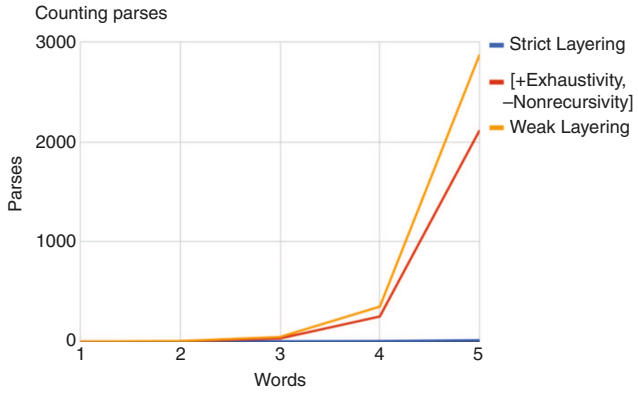

Figure 6: Number of prosodic parses under Weak Layering.

3 GEN in SPOT

An OT system consists of two components: GEN and CON (the constraint set). This section describes SPOT's implementation of GEN, which is designed for studying phonological phrasing, although it can be adapted for analyses of other levels of the prosodic hierarchy. GEN in SPOT is defined as follows:

(3) Gen(S) = {P such that P is a tree and P has all the properties in (4)}

(4) Properties of P
   a. P has a single root node ι.
   b. Every non-root, non-terminal prosodic node is of category φ.
   c. Every prosodic terminal node is of category ω.
   d. Every lexical terminal is mapped onto a node of category ω.
   e. The terminal string of P is identical to the terminal string of S.
   f. *Non-recursivity (Optional)*: No child of an ι is an ι, no child of a φ is a φ, and no child of a ω is a ω. [JavaScript: `obeysNonrecursivity`]
   g. *Exhaustivity (Optional)*: Every child of an ι is of category φ or higher, and every child of a φ is of category ω or higher. [JavaScript: `obeysExhaustivity`]
   h. *Headedness (Optional)*: Every ι dominates at least one φ, and every φ dominates at least one ω. [JavaScript: `obeysHeadedness`]
   i. *No vacuous recursion*: Two nodes of the same category must dominate non-identical terminal strings.

If all properties (a–i) are used, then GEN only creates candidates that obey Strict Layering (Selkirk 1986). However, in the default settings of GEN, only properties (a–e) and (i) are enforced – this is an implementation of Weak Layering (Selkirk 1995, Ito and Mester 2003). Candidates with recursion at the φ level (excluded if property f is enforced) are omitted or included based on the setting of the option `obeysNonrecursivity` to true or false. Candidates in which a phonological word may be directly dominated by the intonational phrase violate Exhaustivity (property g), and can be excluded by setting the option `obeysExhaustivity: true`. Finally, Headedness (property h) can be required by setting the option `obeysHeadedness: true`. These optional settings allow SPOT users to compare different implementations of GEN to test claims about, for example, the (in)violability of headedness (Selkirk 1995; McCarthy 2008; Schiering et al. 2010; Bennett 2012).

SPOT's current implementation of GEN only creates candidates with recursion at one level of the prosodic hierarchy, since most analyses[3] only concern themselves with recursion at one level at a time. (Considering all the candidates with non-vacuous recursion at just one prosodic level already results in such a large candidate set that it can be difficult to detect the key patterns.) Currently, SPOT always labels the recursive category as φ. The phonological phrase was chosen as the recursive level since it is intermediate between two other interface categories (the intonational phrase and phonological word). Recursive intonational phrases, as seen in the phrasing of embedded clauses (e.g., Myrberg 2013), can be studied in SPOT by representing the ι as a SPOT-φ, and the φ as a SPOT-ω. Likewise, recursive phonological words, as seen in the phrasing of compound words (e.g., Ito and Mester 2009; Bellik and Kalivoda 2017), can be studied in SPOT by representing ω as SPOT-φ and feet or syllables as SPOT-ω. To facilitate such analyses where the recursive prosodic level is the highest relevant category, SPOT includes an additional GEN setting: "Require φ Stem", according to which each prosodic tree has one φ that is exactly coextensive with the ι (option `requirePhiStem: true`).

The algorithm for GEN is outlined in Appendix A, and has complexity $O(12^n)$, where n is the number of lexical terminals (cf. OEIS A054726). Computation occurs very quickly for strings of up to six lexical terminals, but the resulting violation tableau may load slowly in the HTML interface.

3 Ito and Mester (2007), which simultaneously considers recursive phonological phrases and words, is the sole exception we are aware of. SPOT cannot currently generate such candidates. However, we plan to extend SPOT's GEN to allow for recursion at multiple prosodic levels in the future.

4 CON

In addition to the implementation of GEN, SPOT includes an implementation of CON. SPOT implements two classes of constraints: Mapping constraints, which play the role of Faithfulness by enforcing correspondence between syntax and prosody, and Markedness constraints, which demand various types of prosodic well-formedness.

4.1 Mapping constraints currently defined in SPOT

Early approaches to syntax-prosody mapping employed an edge-based approach, formalized in Optimality Theory as constraints on Alignment of syntactic edges to prosodic edges (Truckenbrodt 1995; Truckenbrodt 1999; Selkirk 2000). The discovery of more finely articulated prosodic structures, however, led to the development of Match Theory (Selkirk 2011), which posits much stricter constraints on the syntax-prosody mapping, such that every syntactic constituent should have a perfectly Matched prosodic constituent, and vice versa. SPOT can assess violations of both families of mapping constraints (Match Theory as well as Align/Wrap). SPOT's implementations of these constraints are defined in (6) and (7), using the mappings in (5):

(5) CategoryMap $=$ [CP $\longrightarrow$ ι, XP $\longrightarrow$ φ, $X^0 \longrightarrow$ ω]

(6) Match Theory (Selkirk 2011; Elfner 2012):
   a. Match-SP(c): Assign a violation for every node s such that s is a syntactic node of category c, where$c \in$ {CP, XP, X^0}, and s lacks a prosodic correspondent p, where p is a correspondent of s iff s and p dominate the same set of lexical terminals, and CategoryMap(category of s) $=$ category of p.
   b. Match-PS(c): Assign a violation for every node p such that p is a prosodic node of category c, where $c \in$ {ι, φ, ω} and p lacks a syntactic correspondent s, where s is a correspondent of p iff s and p dominate the same set of lexical terminals, and CategoryMap(category of s) $=$ category of p.

(7) Align/Wrap Theory (Truckenbrodt 1995; Truckenbrodt 1999, inspired by Selkirk 1986)
   a. Align-D(c): Assign a violation for every node n such that n is of category c with lexical terminal x at its D edge, and if n a syntactic node, then there is no prosodic node p that has x as its D edge and whose category is CategoryMap(c); or if n is a prosodic node, then there is no syntactic node s of category c' that has x as its D edge such that CategoryMap(c') $= c$; where $D \in$ {left, right} and $c \in$ {CP, XP, X^0, ι, φ, ω}.
   b. Wrap(c): Assign a violation for every node s such that s is of category c, where $c \in$ {CP, XP, X^0}, and s is not wrapped by any prosodic node p of category CategoryMap(c), where p wraps s iff the lexical terminals dominated by s are a subset of the lexical terminals dominated by p.

While Match in Elfner (2012) does not distinguish functional and lexical elements, Selkirk (2011) only requires matching of lexical elements. In SPOT, lexical and functional terminals can be distinguished by labeling functional elements in the syntactic tree with categories other than CP, XP, or X^0. Any other category label, such as `functional_head` or `invisible`, will make them ineligible as arguments to Match-XP, etc. If desired, distinctions between lexical categories could also be made,[4] such that, e.g., Match-NP could be ranked separately from Match-AP.

4 These types of distinctions cannot be made using the current GUI, but require only minor JavaScript programming to implement.

4.2 Markedness constraints currently defined in SPOT

Output prosodic trees are also subject to Markedness constraints. These constraints can demand particular structures, such as Binarity (minimal and maximal). Markedness constraints can also require nodes that are in particular structural relationships to have particular prosodic categories, such as EqualSisters, StrongStart, and NonRecursivity. SPOT implements Markedness in all of these families; a complete list can be found in Appendix B. For the details of the algorithms, please refer to the code (https://github.com/syntax-prosody-ot/main/tree/master/constraints).

SPOT includes multiple possible disambiguations of many constraints. These disambiguations become necessary when the entire candidate set is taken into account, because constraint definitions provided in existing analyses often do not specify the number of violations that should be defined when a node is more than binary branching. For example, how many violations of BinMax (maximal Binarity) should a node with four children incur? One violation because it is a single supra-binary branching node (*BinMax(branches) – categorical*)? Two violations, because it has two more than the maximal allowable number of children (*BinMax(branches) – gradient*)? Similar problems arise in the evaluation of EqualSisters and NonRecursivity. Which definitions yield the most desirable typological outcomes remains an open question, which we hope SPOT can help address.

5 How to use SPOT

The remainder of this paper illustrates how to use SPOT. For expository purposes, we will illustrate points throughout this section using data from Kinyambo (Bickmore 1990).

5.1 Case study: Kinyambo phrasing

In Kinyambo, a phonological process of high tone deletion (HTD) applies throughout the phonological phrase φ, deleting any high tone H that is not in the final ω in its φ. For example, while the word *abakózi* 'worker' is realized with a penultimate H in isolation, this tone is deleted when *abakozi* is the subject of a two-word intransitive (8):

(8) (φabak**o**zi bákajúna)
workers helped
'The workers helped.'

But subjects in Kinyambo do not always form a phonological phrase with the verb. When the subject is modified by an adjective, the noun and adjective form a phonological phrase separate from the verb, as shown by the deletion of the H on *abakozi* and the preservation of the H on *bakúru* in (9).

(9) (φabak**o**zi bak**ú**ru) (φbákajúna)
workers mature helped
'The mature workers helped.'

These examples show that Kinyambo exhibits a phrasal binarity effect (Bickmore 1989; Bickmore 1990, as well as Zec and Inkelas 1990 on the general relevance of prosodic binarity). Its prosodic structure is not fully isomorphic to syntactic structure.

Our case study will use one of the constraint sets explored in Bellik and Kalivoda (2016), which captures the phrasing of the Kinyambo sentences above in Optimality Theory:

(10) *Constraint ranking for Kinyambo:*
EqualSisters, Match(φ, XP) ≫ BinMax ≫ BinMin ≫ Match(XP, φ)

(11) *Constraint definitions*

a. EqualSisters-Adjacent(φ):
Assign a violation for every sequence of adjacent sister nodes *x,y*, such that one of *x,y* is of category φ and the other is not (Myrberg 2013).

b. Match(φ, XP):
Assign a violation for every φ in the prosodic tree (p-tree) that lacks a corresponding XP in the syntactic tree (s-tree), where two nodes p and s correspond to each other iff they dominate the same set of lexical terminals (Selkirk 2011; Elfner 2012).

c. BinMax-Branches(φ):
Assign a violation for every node of category φ that is supra-binary branching, i.e., immediately dominates more than two children.

d. BinMin(φ):
Assign a violation for every node of category φ that is sub-binary branching, i.e., immediately dominates less than two children.

e. Match(XP, φ):
Assign a violation for every XP in the s-tree that lacks a corresponding φ in the p-tree (Selkirk 2011; Elfner 2012).

To derive the ranking for these constraints that would predict the phrasal binarity effects observed above, we first used SPOT to create and evaluate all the possible candidates according to the GEN function described in (4) (with the most Weakly Layered GEN), yielding a total of 456 prosodic output candidates for the four input structures we analyzed. (For details, refer to Bellik and Kalivoda 2016). We then put the resulting violation tableaux into OTWorkplace (Prince et al. 2017) in order to identify all possible optima (presented here) and harmonically bounded candidates (excluded).

The following two tableaux illustrate our basic analysis of Kinyambo (Bellik and Kalivoda 2016). In (12), the analysis for (8), the choice is simply between the mono-phrasal (12a), the intended winner, and the perfectly matching (12b). BinMin prefers (12a) over (12b), and Match-XP has the opposite preference. Therefore, BinMin ≫ Match-XP.

(12) *Tableau for Kinyambo 2-word intransitive*

$[_{TP}\ [_{NP}$ N$]\ [_{VP}$ V$]]$	**EqSis**	**Match-φ**	**BinMax**	**BinMin**	**Match-XP**
a. ⟶ $(_{\varphi}$ N V$)$					2
b. $(_{\varphi}\ (_{\varphi}$ N$)\ (_{\varphi}$ V$))$				W_2	L_0

The tableau in (13) for the data in (9) provides further ranking information for Kinyambo. Here, the subject should be a single φ and the VP should be a single φ, as in (13a). As in (12), BinMin ≫ Match-XP accounts for the preference for (13a) over the perfectly matching candidate (13b). But a complete analysis must consider two additional candidates. In (13c), the entire sentence is flattened to one φ, and in (13d), the verb adjoins to the subject φ to avoid a BinMin violation. The impossibility of (13c) reveals that BinMax ≫ BinMin. That is, on this ranking it is more important to avoid a ternary-branching φ than it is to avoid a unary-branching φ. Since the intended loser (13d) is perfectly binary branching, the constraint EqualSisters is needed to rule it out [the offending part of the structure is $(_{\varphi}\ \varphi\ \omega_V)$], establishing the ranking EqualSisters ≫ BinMin.

(13) *Tableau for Kinyambo 3-word intransitive*

$[_{TP}\ [_{NP}\ [_{NP}$ N$]\ [_{AP}$ A$]]\ [_{VP}$ V$]]$	**EqSis**	**Match-φ**	**BinMax**	**BinMin**	**Match-XP**
a. ⟶ $(_{\varphi}\ (_{\varphi}$ N A$)\ (_{\varphi}$ V$))$				1	2
b. $(_{\varphi}\ (_{\varphi}\ (_{\varphi}$ N$)\ (_{\varphi}$ A$))\ (_{\varphi}$ V$))$				W_3	L_0
c. $(_{\varphi}$ N A V$)$			W_1	L_0	W_4
d. $(_{\varphi}\ (_{\varphi}$ N A$)$ V$)$	W_1			L_0	W_3

The inclusion of candidate (13d) is necessary to establish the ranking of EqualSisters with respect to the other constraints, underscoring the importance of exhaustive candidate generation. A common heuristic for generating plausible optima is to imagine which candidate each constraint would consider optimal. Under this heuristic, BinMax might bring to mind (13a); Match-XP, (13b); and BinMin, (13c). In fact, (13d) satisfies BinMin as well as (13c) does, and performs better on Match-XP, but this could easily be overlooked if candidates were being generated by hand, with disastrous effects for the validity of the ranking and any typology derived from it (cf. Karttunen 2006; Bane and Riggle 2012).

Bellik and Kalivoda (2016) also discuss another candidate that is likely to be omitted by accident: ($_{\varphi}$ N ($_{\varphi}$ A V)), which splits apart the noun phrase – a non-cohering parse of the noun+adjective sequence. When Match is sensitive to both upper and lower segments of adjunction structures, as in the analysis here, then the non-cohering parse is harmonically bounded. However, if only the lower segment of an adjunction structure is visible to the mapping constraints (Truckenbrodt 1995; Truckenbrodt 1999; Selkirk 2011), then Match cannot distinguish the non-cohering parse from a cohering parse as in candidate (13d) ($_{\varphi}$ ($_{\varphi}$ N A) V). However, the non-cohering parse is likely to be omitted in manual candidate generation, since it is unintuitive to imagine splitting apart a syntactic constituent in this manner.

5.2 Using SPOT through the GUI

The remainder of this paper explains how to use SPOT to arrive at the type of analysis shown above. The application is most easily accessed through the graphical user interface (GUI), which is available online (https://people.ucsc.edu/~jbellik/research/spot/interface1.html). The entire SPOT codebase is available on Github at <https://github.com/syntax-prosody-ot/main>, and can be downloaded in order to have a local copy of SPOT. To access the GUI locally, interface1.html can be opened with a browser, such as Chrome, Firefox, or Safari.[5] The browser will display an interface as in Figure 7.

5.2.1 Constraint selection

The SPOT interface lists all the available constraints with checkboxes next to their names. A constraint can be added to CON by clicking its checkbox. The radio buttons to the right of the constraint names allow the user to choose the prosodic or syntactic category to use as the argument or parameter for the selected constraint. In the case study here, we would select the constraints from (10) (reordered here to correspond to their positions in the SPOT interface): Match(XP, φ), Match(φ, XP), EqualSisters-φ (adjacent), BinMax(branches)-φ, BinMin-φ (Figure 7).

5.2.2 Building the syntactic input tree

Once all the desired constraints have been selected, the next step is to input the desired syntactic tree. This can be done using the tree-building GUI in the section of the page headed “Input syntactic tree.” Click the button labeled “Generate JS tree.” Then enter the string of lexical terminals for the syntactic tree of interest in the textbox marked “String of terminals” and click “Go!”. For the Kinyambo sentence in (9), we could enter “abak**o**zi bak**ú**ru bákajúna” or the English equivalent “workers mature helped.” After clicking “Go!”, the interface will display the seed of a syntactic tree structure (Figure 8).

This syntactic tree can be extended and modified according to the desired syntactic assumptions. Labels with grey backgrounds represent the syntactic category of the node, and can be edited. If a node is intended to be visible to the mapping constraints (Match or Align/Wrap), then its category must belong to the set of syntactic categories defined in prosodicHierarchy.js. In the standard version of SPOT, these syntactic categories are: “cp” (for Complementizer Phrase), “xp” (for Noun Phrase, Verb Phrase, Prepositional Phrase, etc.), and

5 Not all functions of SPOT work in Internet Explorer.

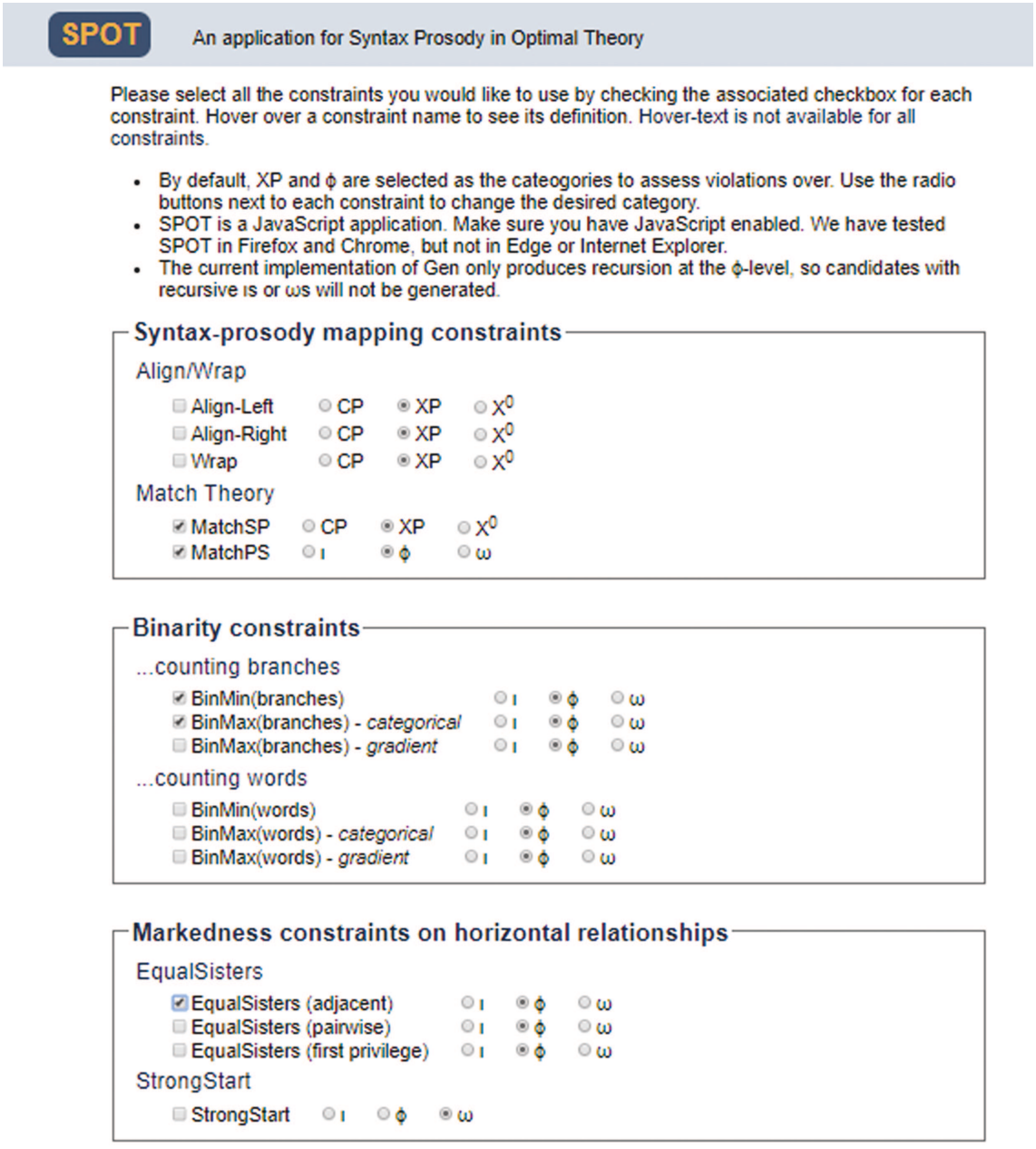

Figure 7: Constraint selection.

"x0" (for syntactic words).[6] To add nodes to the tree, select one or more existing nodes by clicking on them (one at a time), then click the button "Add Mother" above the tree (left-hand button in Figure 9). To delete a node or nodes, select them by clicking on them, then click "Delete" (middle button in Figure 9).

When the tree is finished, click "Done! Convert tree to code" to convert it into a JavaScript tree, which will appear in the textbox below. Each node in the tree is a JavaScript object with three attributes: cat (the category), id (the label), and children (an array of nodes immediately below it in the tree). Further

6 Although "XP" usually indicates a maximal projection of any category, including C (complementizer), the SPOT category "cp" does not count as "xp", because CPs alone are meant to map to ι. Selkirk (2011) and others distinguish between CPs with illocutionary force (e.g., matrix CPs) and those that lack illocutionary force (e.g., certain embedded CPs). A CP that lacks illocutionary force, and is therefore intended to map to φ rather than ι, can be represented in SPOT as "xp".

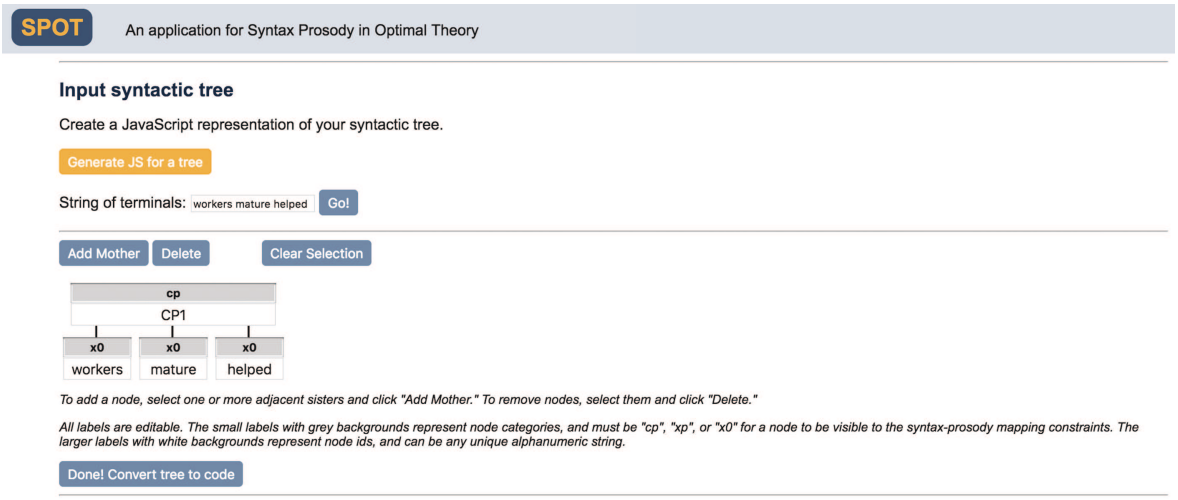

Figure 8: Seed of a syntactic tree.

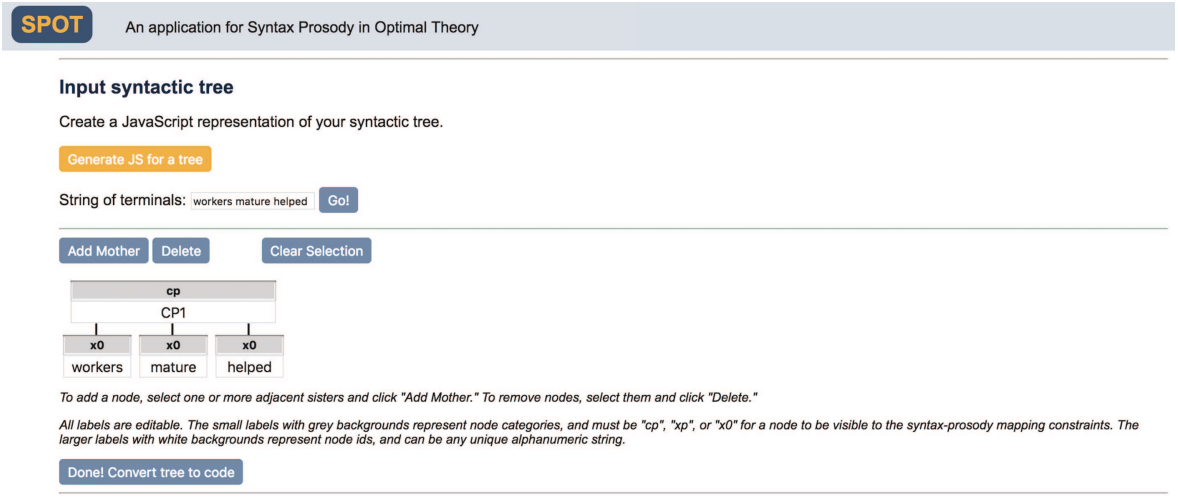

Figure 9: Selecting nodes.

attributes can be added simply by modifying the tree in the textbox, as long as JavaScript syntax is adhered to. This JavaScript tree will be the input to SPOT; additional changes to the tree in the textbox can affect the analysis.

Note that SPOT does not make any assumptions about the shape of syntactic trees. Defining the syntax is the prerogative of the analyst.

5.3 Using SPOT without the GUI

Users who are familiar with JavaScript and HTML can bypass the provided GUI and write their own HTML file to call on SPOT's functionality. This can be useful for rapidly evaluating several different syntactic trees (using a for-loop), or for building custom constraints. Existing constraint definitions and SPOT's implementation of GEN are all written in JavaScript (JS); the HTML files call the relevant JS functions, which perform the actual computations. Custom HTML files must load build.js to be able to access all the constraints. Custom constraints can be written directly in the HTML file (inside a pair of <script> tags), or saved in a separate JS file and loaded into the HTML file.

The codebase for SPOT is accessible at <https://github.com/syntax-prosody-ot/main>. A local copy can be downloaded using Github's download button. The organization of the codebase is described in Appendix C.

5.4 Analyzing SPOT's output using other OT tools

The output of SPOT is a violation tableau (VT), which is both displayed graphically in the browser and downloadable as a tab-separated table (as a .csv file). The downloaded VT is formatted for compatibility with OTWorkplace (Prince et al. 2017), which we used in conjunction with SPOT to arrive at the constraint ranking for Kinyambo phrasing, as well as to distinguish possible optima from harmonically bounded candidates and explore the different typological consequences of slightly different OT systems (Bellik and Kalivoda 2016). With slight modifications, SPOT's .csv VT file can also serve as the input to other OT software packages, such as OTSoft (Hayes et al. 2013), or PyPhon (Bane and Riggle 2010). SPOT's VT can also be viewed in any spreadsheet application or text editor, or loaded as a table or dataframe in R (R Core Team 2017) or another programming language.

6 Conclusion

We have presented SPOT, a tool for work on the Syntax-Prosody interface situated in the theoretical framework of Optimality Theory. SPOT is designed to interface with typology and ranking calculating software, such as OTWorkplace (Prince et al. 2017). A graphical user interface enables easy generation of the violation tableau for a given syntactic input, and the project source code is freely available on Github for analysts who wish to develop extensions or use the application locally. SPOT facilitates consideration of all possible candidates and comparison of different implementations of the same conceptual constraint. It is our hope that SPOT will encourage refinements to the constraints in syntax-prosody mapping, as well as exploration of their typological consequences. Indeed, SPOT has already been fruitfully used to investigate phrasing in Kinyambo (Bellik and Kalivoda 2016) and Basque (Hedding 2017); compound word prosody in Danish (Bellik and Kalivoda 2017) and Japanese (Kalivoda 2018); and the phrasings of ditransitives cross-linguistically (Kalivoda 2018). SPOT is still under active development to enable modeling of other prosodic phenomena, such as prosodic movement (Agbayani et al. 2011; Agbayani et al. 2015; Bennett et al. 2016), differential treatment of functional and lexical items (Selkirk 2003), and recursion at multiple levels of the prosodic hierarchy (Ito and Mester 2007).

Acknowledgements: This material is based upon work supported by the National Science Foundation under Grant No. 1749368 (to Junko Ito and Armin Mester). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

Agbayani, Brian, Chris Golston & Dasha Henderer. 2011. Prosodic movement. In Mary Byram Washburn, Katherine McKinney-Bock, Erika Varis, Ann Sawyer & Barbara Tomaszewicz (eds.), *Proceedings of the 28th West Coast Conference on Formal Linguistics*, 231–239. Somerville, MA: Cascadilla Proceedings Project.

Agbayani, Brian, Chris Golston & Toru Ishii. 2015. Syntactic and prosodic scrambling in Japanese. *Natural Language & Linguistic Theory* 33(1). 47–77.

Bane, Max & Jason Riggle. 2010. PYPHON 1.0. Software package. http://code.google.com/p/clml/.

Bane, Max & Jason Riggle. 2012. Consequences of candidate omission. *Linguistic Inquiry* 43(4). 695–706.

Bellik, Jennifer & Nick Kalivoda. 2016. Adjunction and branchingness effects in syntax-prosody mapping. In Gunnar Ólafur Hansson, Ashley Farris-Trimble, Kevin McMullin & Douglas Pulleyblank (eds.), *Supplemental proceedings of the 2015 Annual Meeting on Phonology* Article 2, 1–11. Washington, DC: Linguistic Society of America.

Bellik, Jennifer & Nick Kalivoda. 2017. Danish stød in recursive prosodic words. Poster presented at Northwestern Phon{etics, ology} Conference, University of British Columbia, May 19–21.

Bellik, Jennifer, Ozan Bellik & Nick Kalivoda. 2017. Syntax prosody in OT (SPOT). JavaScript application. https://github.com/syntax-prosody-ot.

Bennett, Ryan. 2012. *Foot-conditioned phonotactics and prosodic constituency*. Santa Cruz, CA: University of California, Santa Cruz dissertation.

Bennett, Ryan, Emily Elfner & James McCloskey. 2016. Lightest to the right: An apparently anomalous displacement in Irish. *Linguistic Inquiry* 47(2). 169–234.

Bickmore, Lee. 1989. *Kinyambo prosody*. Los Angeles, CA: University of California, Los Angeles dissertation.

Bickmore, Lee. 1990. Branching nodes and prosodic categories. In Sharon Inkelas & Draga Zec (eds.), *The phonology-syntax connection*, 1–18. Stanford, CA: CSLI Publications and University of Chicago Press.

Elfner, Emily. 2012. *Syntax-prosody interactions in Irish*. Amherst, MA: University of Massachusetts dissertation.

Hayes, Bruce, Bruce Tesar & Kie Zuraw. 2013. OTSoft 2.5. Software package. http://www.linguistics.ucla.edu/people/hayes/otsoft/.

Hedding, Andrew. 2017. Phonological phrasing of ditransitives in Arrasate Basque. Ms. University of California, Santa Cruz.

Ito, Junko & Armin Mester. 1992. Weak layering and word binarity. Santa Cruz, CA: Linguistic Research Center, LRC-92-09, University of California, Santa Cruz. [A slightly revised version appeared in *Festschrift for Shosuke Haraguchi*, 2003.]

Ito, Junko & Armin Mester. 2007. Prosodic adjunction in Japanese compounds. *MIT Working Papers in Linguistics* 55(Formal Approaches to Japanese Linguistics 4). 97–111.

Ito, Junko & Armin Mester. 2009. The extended prosodic word. In Barış Kabak & Janet Grijzenhout (eds.), *Phonological domains: Universals and deviations*, 135–194. Berlin & New York: Mouton de Gruyter.

Ito, Junko & Armin Mester. 2013. Prosodic subcategories in Japanese. *Lingua* 124. 20–40.

Kalivoda, Nick. 2018. *Syntax-prosody mismatches in Optimality Theory*. Santa Cruz, CA: University of California, Santa Cruz dissertation.

Karttunen, Lauri. 2006. The insufficiency of paper-and-pencil linguistics: The case of Finnish prosody. In Miriam Butt, Mary Dalrymple & Tracy Holloway King (eds.), *Intelligent linguistic architectures: Variations on themes by Ronald M. Kaplan*, 287–300. Stanford, CA: CSLI Publications.

Ladd, D. Robert. 1986. Intonational phrasing: the case for recursive prosodic structure. *Phonology* 3. 311–340.

Ladd, D. Robert. 1988. Declination "reset" and the hierarchical organization of utterances. *Journal of the Acoustical Society of America* 84. 530–544.

Langendoen, D. Terence. 1987. On the phrasing of coordinate compound structures. In Brian Joseph & Arnold Zwicky (eds.), *A festschrift for Ilse Lehiste* 186–196. Columbus, OH: Ohio State University.

Lee, Seunghun J. & Elisabeth Selkirk. 2015. Constituency in sentence phonology: An introduction. *Phonology* 32(1). 1–18.

McCarthy, John J. 2008. The serial interaction of stress and syncope. *Natural Language & Linguistic Theory* 26. 499–546.

Myrberg, Sara. 2013. Sisterhood in prosodic branching. *Phonology* 30(1). 73–124.

Nespor, Marina & Irene Vogel. 1986. *Prosodic phonology*. Dordrecht: Foris.

OEIS A054726. *Online encyclopedia of integer sequences*. https://oeis.org/A054726 (accessed 5 April, 2018).

Prince, Alan & Paul Smolensky. 2004[1993]. *Optimality Theory*: Constraint interaction in generative grammar. Malden, MA: Blackwell. [Revision of 1993 technical report, Rutgers University Center for Cognitive Science. Available on Rutgers Optimality Archive, ROA-537.]

Prince, Alan, Bruce Tesar & Nazarré Merchant. 2017. OTWorkplace. https://sites.google.com/site/otworkplace.

R Core Team. 2017. R: A language and environment for statistical computing. Vienna, Austria: The R Foundation for Statistical Computing. https://www.r-project.org/.

Schiering, René, Balthasar Bickel & Kristine A. Hildebrandt. 2010. The prosodic word is not universal, but emergent. *Journal of Linguistics* 46. 657–709.

Selkirk, Elisabeth. 1986. On derived domains in sentence phonology. *Phonology Yearbook* 3. 371–405.

Selkirk, Elisabeth. 1995. The prosodic structure of function words. In Jill Beckman, Laura Walsh Dickey & Suzanne Urbanczyk (eds.), *Papers in Optimality Theory*, 439–470. Amherst, MA: GLSA.

Selkirk, Elisabeth O. 1984. *Phonology and syntax: the relation between sound and structure*. Cambridge: MIT Press.

Selkirk, Elisabeth. 2000. The interaction of constraints on prosodic phrasing. In Merle Horne (ed.), *Prosody: Theory and experiment: Studies presented to Gösta Bruce*, 231–261. Dordrecht & Boston: Kluwer Academic Publishers.

Selkirk, E. 2003. The prosodic structure of function words. In McCarthy J, (ed.), *Optimality theory in phonology: A reader*, Chapter 25. Oxford, UK: Blackwell

Selkirk, Elisabeth. 2011. The syntax-phonology interface. In John Goldsmith, Jason Riggle & Alan C. L. Yu (eds.), *The handbook of phonological theory*, 435–484. Cambridge, MA: Blackwell.

Selkirk, Elisabeth & Gorka Elordieta. 2010. The role for prosodic markedness constraints in phonological phrase formation in two pitch accent languages. Handout of paper presented at Tone and Intonation in Europe (TIE) 4, Stockholm University, Department of Scandinavian Languages. http://www.hum.su.se/pub/jsp/polopoly.jsp?d=13236anda=73666 (accessed 15 August 2015).

Truckenbrodt, Hubert. 1995. *Phonological phrases: Their relation to syntax, focus, and prominence*. Cambridge, MA: Massachusetts Institute of Technology dissertation.

Truckenbrodt, Hubert. 1999. On the relation between syntactic phrases and phonological phrases. *Linguistic Inquiry* 30(2). 219–255.
Wagner, Michael. 2010. Prosody and recursion in coordinate structures and beyond. *Natural Language & Linguistic Theory* 28. 183–237.
Zec, Draga. 2005. Prosodic differences among function words. *Phonology* 22(1). 77–112.
Zec, D. & Inkelas, S. (eds.) 1990. *The phonology-syntax connection*, Chicago, IL, USA: CSLI Publications and the University of Chicago Press.

Appendix A

The code for GEN can be found in candidategenerator.js. The function takes as its input a list of syntactic terminals (leaves) and a list of options (obeysNonrecursivity, obeysExhaustivity, obeysHeadedness).

(14) GEN(leaves) = gen(leaves) + phi_wrap(gen(leaves)

(15) gen(leaves)
  a. Base case: If there are 0 leaves in the input, return the list of candidates and terminate.
  b. Recursive case: If there is at least one leaf in the input, then:
    i. For i from 1 to the number of leaves in leaves,
      1. Case 1: The first i leaves in leaves attach directly to the parent.
        a. Let Leftside = the first i elements in leaves
        b. Let Rightsides = all the outputs of gen(the remaining leaves) that do not have w at their left edge (i.e., outputs that are either an empty tree, or a subtree with a phi boundary at its left edge)
        c. Combine Leftside with each element in Rightsides, and add the results to the candidate list
      2. Case 2: The first i leaves do not attach directly to the parent (they are wrapped in a phi).
        a. Leftsides = phi_wrap(gen(the first i elements in leaves))
        b. Rightsides = gen(the remaining elements in leaves)
        c. Cross Leftsides and Rightsides with each other. Add the results to the candidate list.
      3. Return the candidate list.

(16) phi_wrap(tree_list): Wrap each element of tree_list in a phi and return the resulting list. By calling GEN recursively, all the possible sub-trees are also incorporated into the candidate set.

Appendix B: Markedness constraint families implemented in SPOT

(17) Constraints on Binarity (Maximal and Minimal):
  a. Counting the number of branches a node has (Elfner 2012; Bellik and Kalivoda 2016):
    iii. BinMin-branches
    iv. BinMax-branches (categorical)
    v. BinMax-branches (gradient)
  b. Counting the number of leaves a node dominates (Ito and Mester 2007, to appear):
    iii. BinMin-leaves
    iv. BinMax-leaves (categorical)
    v. BinMax-leaves (gradient)

(18) Constraints on horizontal relationships:
  a. EqualSisters (Myrberg 2013)

iii. EqualSisters (adjacent)
iv. EqualSisters (pairwise)
v. EqualSisters (first privilege)
b. StrongStart (Selkirk 2011)

(19) Constraints on vertical relationships:
a. Exhaustivity
b. NonRecursivity
iii. Non-recursivity, assessed by dominated node
iv. Non-recursivity, assessed by non-overlapping leaves (Truckenbrodt 1999)

(20) Constraints on accent (developed for analyses of Japanese and Basque, Selkirk and Elordieta 2010, Ito and Mester 2013):
a. Accent-As-Head
b. NoLapse

Appendix C

The most important elements of the codebase are organized as follows (ordered by relevance):
- main/interface1.html: Open this file in a browser to use SPOT's graphical user interface (GUI), which can be used without any programming. Note that this is not the same as interface1.js, which is the JavaScript that accompanies the GUI and does not need to be opened directly.
- main/prosodicHierarchy.js: This defines the prosodic and syntactic categories and their mappings that SPOT assumes, and can be modified locally by analysts who wish to include other prosodic categories. The default hierarchy includes three categories, ι (intonational phrase), φ (phonological phrase), and ω (phonological word). Note that adjusting the prosodic hierarchy will not automatically alter GEN to incorporate those new prosodic categories into the prosodic candidates.
- main/candidateGenerator.js: This file defines SPOT's GEN function (described in Section 3).
- main/constraints: This folder contains all constraint definition files. Conceptually related constraints, such as different variations on the Binarity constraints, are defined in the same JS file (binarity.js, in the case of Binarity constraints). Look at these JS files to see the details of the algorithms SPOT uses to evaluate constraint violations. Users wishing to add their own constraints should save them to this folder, and then run build.sh to incorporate them into main/build to make them more accessible to the HTML files.
- main/build: The build folder contains concatenations of all the JavaScript files so that they can be easily imported into an HTML file for performing an analysis. The contents of main/build are created automatically by the bash shell script jsbuild.sh, located in main.
- main/lib: This folder contains utilities – currently just jszip.min, a library that converts a collection of csv files into a single zip file for convenient downloading.