

Syntax Prosody in Optimality Theory (SPOT) app tutorial

Jennifer Bellik & Nick Kalivoda*

Abstract. SPOT (Syntax Prosody in Optimality Theory app; <http://spot.sites.ucsc.edu>) automates candidate generation and evaluation for work on the syntax-prosody interface. SPOT is intended to facilitate the creation and comparison of multiple versions of an analysis, in service of refining constraint definitions and theory development. The codebase is available at <https://github.com/syntax-prosody-ot>. This paper briefly explains the motivation for the SPOT app, then walks through the process of creating an analysis in SPOT. We show how to create input syntactic trees, either manually or automatically; how to select a constraint set; and how to generate tableaux with candidates, constraints, and violation counts. Finally, we show how to use the output of SPOT to calculate rankings and typologies using an OT application.

Keywords. Optimality Theory; syntax-prosody interface; prosody; typology; SPOT; OTWorkplace

1. Introduction. In order to create a valid analysis in Optimality Theory (OT), we must formally define the OT system that we are studying (Alber et al. 2016, Prince 2016, Prince 2017). A system *S* consists of two components: *S.GEN* and *S.CON*. *S.GEN* is the set of input and output candidates (algorithmically defined) in *S*, while *S.CON* is the set of constraints in *S*.

Under the widely assumed Prosodic Hierarchy Theory (Selkirk 1978, 1980, 1984; Nespor & Vogel 1986; adopted by many others), inputs to phrasal phonology are syntactic trees, while outputs are prosodic trees. Syntactic trees consist of heads (X^0), phrases (XP), and clauses (CP). Prosodic trees consisting of prosodic words (ω), phonological phrases (ϕ), and intonational phrases (ι).¹ Because candidates are pairs of trees, even the strictest GEN function yields an exponential increase in the number of output structures as the number of words in the sentence grows (Figure 1). Under Strict Layering (Selkirk 1984, Nespor & Vogel 1986), when level-skipping and level-repeating are banned, the number of prosodic trees rooted in an intonational phrase and with prosodic words for terminals is $2^{(n-1)}$, where *n* is the number of terminals in the tree (dashed grey line in Figure 1). This exponential function grows relatively slowly, such that six terminals can be parsed into 32 strictly layered trees—a candidate set that would be tedious, but possible, to generate and evaluate by hand.

With the inclusion of non-exhaustive parses (Weak Layering, Ito & Mester 2003/1992), words can be parsed directly into the intonational phrase, skipping the phonological phrase level.² This increases the number of possible parses (compare [-Exhaustive] black lines to

*Our thanks to Junko Ito, Armin Mester, SPOT co-creator Ozan Bellik, and RAs Colin Chen, Timothy Gee, Iva Petkov, Ed Shingler, Max Tarlov, and Su Zin, as well as participants at the 2020 meeting of the Society for Typological Analysis. This work was supported of the National Science Foundation Grant #1749368, awarded to Junko Ito and Armin Mester. Authors: Jennifer Bellik, University of California, Santa Cruz (jbellek@ucsc.edu) & Nick Kalivoda, Lund University (nicholas.kalivoda@ling.lu.se)

¹ SPOT assumes the minimal prosodic hierarchy employed in Selkirk 2011, Ito & Mester 2013, and other works of Match Theory.

² SPOT has a distinct GEN parameter to enforce Headedness (proposed as an inviolable requirement by Ito & Mester 2013), ensuring that every node of category *k* has at least one child of category *k*-1. The numbers for [-Exhaustive] black lines in Figure 1 diminish by one if Headedness is enforced on GEN, which excludes the single candidate in each set in which all words are parsed directly into the intonational phrase.

[+Exhaustive] grey lines). When prosodic recursion is also allowed (Ito & Mester 2013), the number of trees increases even faster (compare [-Nonrecursive] solid lines to [+Nonrecursive] dashed lines of the same color).

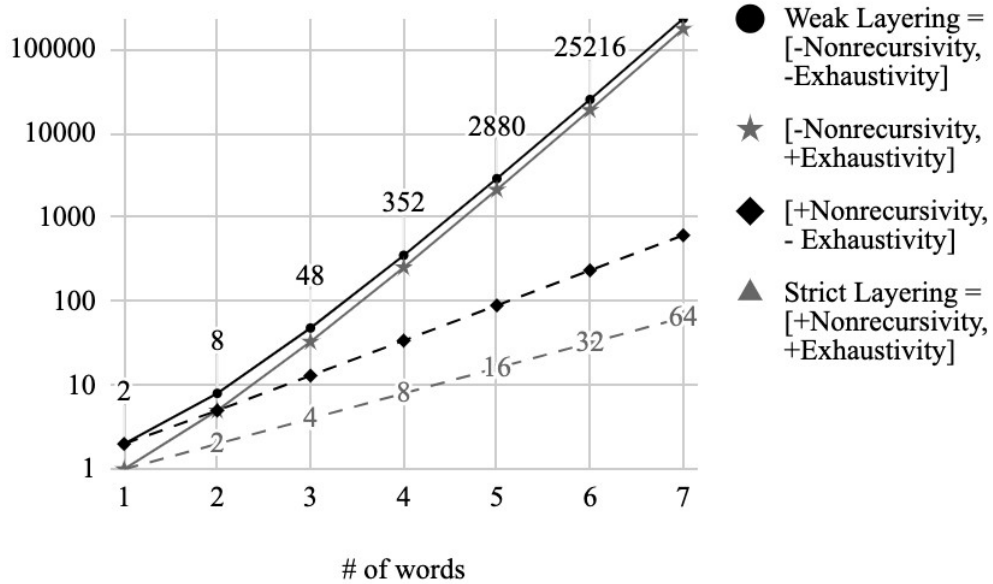


Figure 1. Number of prosodic trees rooted in τ with n words as terminals

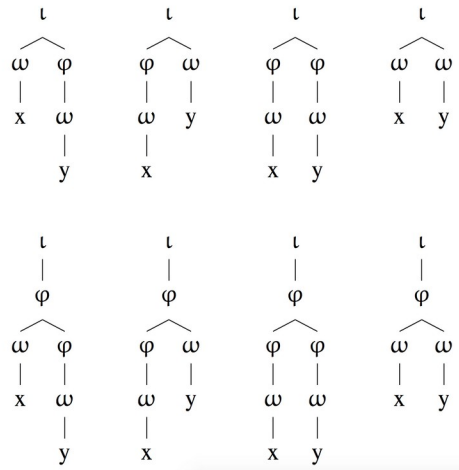


Figure 2. The eight parses of two terminals with weak layering

When both level-skipping and level-repeating are allowed, there are already eight possible parses of two terminals (depicted in Figure 2). With six terminals, over twenty-five thousand weakly layered parses become possible (Figure 1). Most of these candidates will ultimately be harmonically bounded, yet all must be considered lest an omitted candidate invalidate the final analysis (Bane & Riggle 2012, Karttunen 2006).

These huge numbers of candidates demand an automated tool for candidate generation and evaluation in syntax-prosody analyses. Obviously it is impractical to build and evaluate

thousands of prosodic trees by hand. It is also impossible to do this automatically using OTWorkplace (Prince et al. 2020) or other existing OT software, because their GEN capacity is limited to regular expressions, which cannot handle recursion.

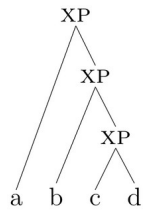
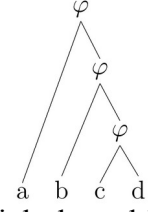
This is where SPOT comes in. SPOT stands for Syntax-Prosody in Optimality Theory. It is a JavaScript application under development since 2015 (Bellik, Bellik, & Kalivoda 2015–2021) that automates candidate generation (GEN) and constraint evaluation (CON) for work on the syntax-prosody interface. SPOT news and information are available at the SPOT website (<http://spot.sites.ucsc.edu>), which links to the webapp and codebase. SPOT was designed specifically to handle tree structures. It produces violation tableaux which can be viewed in the browser, or imported into the OT tool of your choice for further analysis.

This paper explains, step by step, how to use SPOT to build a simplified version of a system that analyzes syntax-prosody matches and mismatches, inspired by data from Japanese. We will call this system Msp.Asp, which stands for Match(Syntax→Prosody), Align(Syntax→Prosody). We also explain how to begin analyzing SPOT's violation tableaux in OTWorkplace.

2. Getting to the SPOT interface. To use SPOT online, navigate to the SPOT web interface by opening the SPOT website, preferably in Chrome or Firefox: spot.sites.ucsc.edu. At the top of the page, click “SPOT webapp.” There is also a link at the bottom of all pages on the website. This will take you to the SPOT web interface.

It is also possible to install SPOT locally so that it can be used without an internet connection later. To take this route, click “SPOT codebase” on the website to navigate to the Github repository, where you can follow the instructions in README.md to download the codebase from Github, build locally, and access the app in your browser from your local copy of the SPOT interface.

3. Defining inputs. The inputs to Msp.Asp are abstracted from data on Japanese phrasing in Kubozono 1989 (Table 1). Of the five possible binary-branching syntactic trees with four words, four are mapped to matching prosodic structures in Japanese, while one, the fully left-branching input, is rebracketed into a balanced prosodic structure. These structures are diagnosed by %LH rises in pitch at the left edge of the ϕ , and by downstep that applies to non-initial rises throughout the maximal ϕ . In the mismatching prosody in (b), an unexpected rise appears on the third word, indicating that it is at the left edge of a ϕ that does not correspond to any XP in the input. No such rebracketing occurs in its mirror image, the fully right-branching syntax in (a), which features rises on the first three words. If (a) were also rebracketed, it would have the same pitch contour as the left-branching input (b) and the symmetrically-branching input (e).

Syntax		Prosody	Match/Mismatch
 a. Right branching	→	 Right-branching	Match

b. Left branching	→	Symmetric	Mismatch! (Unexpected %LH rise on c)
c. Mixed branching (R + L)	→	Mixed branching	Match
d. Mixed branching (L + R)	→	Mixed branching	Match
e. Symmetric	→	Symmetric	Match

Table 1. Inspirational data on Japanese phrasing (abstracted from Kubozono 1989)

We include the five four-word trees from Table 1 in the set of inputs for our system Msp.Asp (1). Msp.Asp.GEN will also include two three-word syntactic trees. (For the full set of inputs, see the appendix.)

- (1) Msp.Asp.GEN Inputs = Binary-branching syntactic trees
 - a. on terminal strings of 3-4 nodes,
 - b. where terminal nodes are syntactic words X^0 and non-terminal nodes are syntactic phrases XP.
 - c. The orthographic representation of the bracketing structure imposes a linear order on the leaves.

As it turns out, the trees in (2) form a support for this system, meaning that as long as those three inputs are included in the calculation of the typology, it will contain the exact same languages as it would with the inclusion of all or some additional trees drawn from the set in (1). Thus, to analyze this system, we must build at least the trees 4wR, 4wM, and 4wL in (2), although we can also build the other trees specified by (1). The labels of the words are arbitrary, but must be distinct. We will give arbitrary labels to the XPs as well, simply so SPOT can

distinguish them. Table 2 shows traditional tree depictions of 4wR, 4wM, and 4wL, as well as the graphical depictions that SPOT's tree builder uses.

- (2) A universal support for the system Msp.Asp
- [a [b [c d]]] Four words, right-branching = 4wR
 - [a [[b c] d]] Four words, mixed-branching = 4wM
 - [[[a b] c] d] Four words, left-branching = 4wL

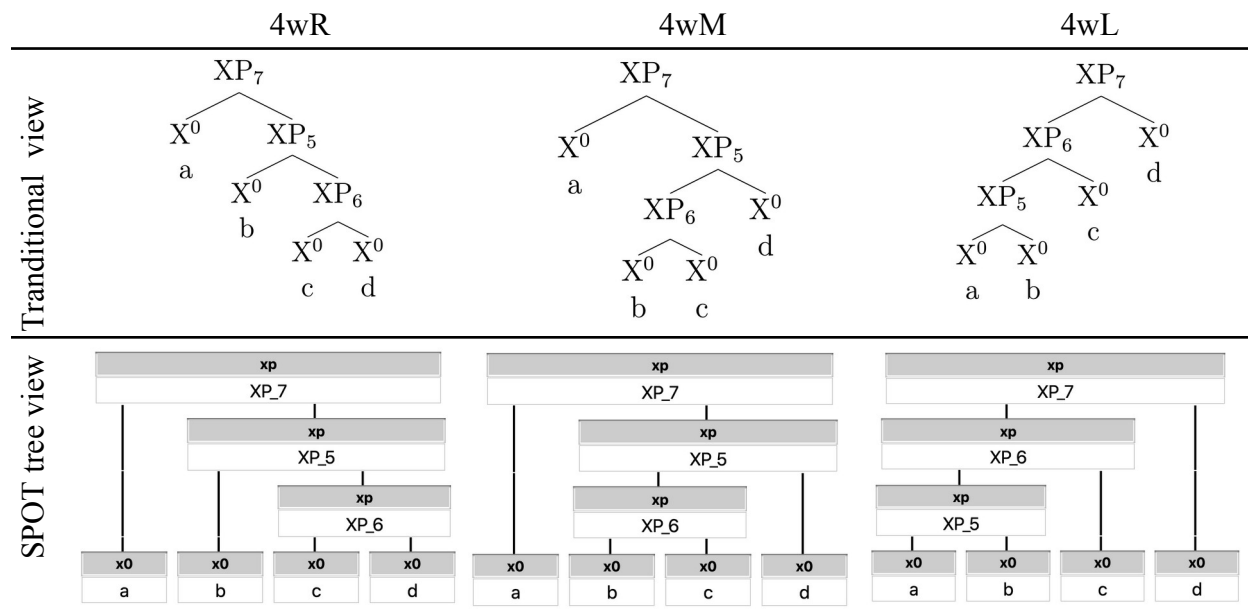


Table 2. Tree diagrams of the universal support for Msp.Asp

3.1. BUILDING SYNTACTIC TREES MANUALLY IN SPOT. Input trees in SPOT can be constructed in two different ways. We will first cover the manual approach, which allows the analyst to decide exactly what trees to include, and provides a graphical tree depiction, rather than only a bracketed representation of the trees. To use this approach, start in the **Manual** tab under **GEN: Input parameters**. Type “a b c d” into “String of terminals” and click on the button “Build syntax.” This will open the tree builder, with the seed of the syntactic tree (Figure 3).

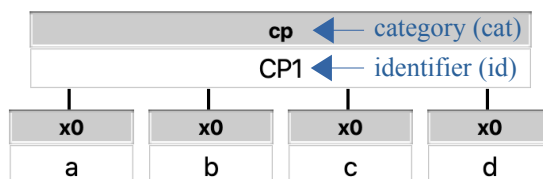


Figure 3. Seed of a four-terminal tree

In SPOT's graphical tree representation, every node has two attributes, a category and an identifier. If the node is to be visible to the syntax-prosody mapping constraints, its category, shown with a grey background, must be “cp” (complementizer phrase), “xp” (syntactic phrase) or “x0” (syntactic head). The identifier can be any alphanumeric string (no spaces or hyphens, please), and must be unique. We use the notational convention *cat-id* to refer to specific nodes. The root node of this tree currently has the identifier “CP1” and the category “cp.” We need to

change these to “XP_7” (or some other identifier of your choosing) and “xp,” respectively. To do this, click in the text fields for *cat* and *id* and change the text there.

Next, to create the XP containing *x0-b*, *x0-c*, and *x0-d*, hover over the space immediately surrounding node *x0-b*, and click on the grey area. This area will then turn light blue, indicating that you have selected that node. Now do the same to select *x0-c* and *x0-d*. Finally, click “Add Mother.” This will create the node *xp-XP_5*. The tree should now look like Figure 4.

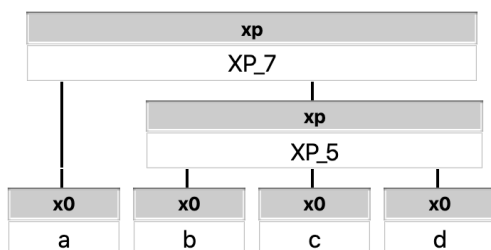


Figure 4. With XP_5 containing *x0-b*, *x0-c*, and *x0-d*

Next, create an XP above *x0-c* and *x0-d* by selecting those nodes and clicking “Add Mother” again. This XP will be labelled *XP_6*, and will complete the tree. It should look like 4wR in Table 2. When the tree is complete, click the button labeled “Done! Add trees to analysis” under the tree-builder. A message will appear next to it, saying, “The trees in the analysis are up-to-date.” This means SPOT has translated that graphical tree representation into a JavaScript tree object. Its code can be viewed using the “Show code” toggle switch below the “Done!” button.

To create another tree, put the terminal string for your tree in the “String of terminals” box, and click on “Build syntax” again. The seed of a second tree will appear below the one you already built. Add additional structure using the same procedures as above, to create the other two trees (4wM and 4wL).

It is also possible to add additional information to the trees, such as labeling some nodes as being functional projections, having silent heads, being accented, or bearing semantic focus. Click on the info button immediately above the tree builder, next to the “Clear selection” button, to access information about how to add these features to a tree you have created.

Another feature available in this part of the SPOT interface is “Trim syntactic tree,” which will prune an elaborated syntactic tree down to only the structure that is visible to the mapping constraints in the version of MATCH proposed by Elfner (2012). Further details about this can be accessed on the SPOT interface using the info button next to “Trim syntactic tree” immediately above the tree-builder. Trimming will not alter the trees in Msp.Asp, however, since they do not contain any phonologically empty elements.

3.2. BUILDING SYNTACTIC TREES AUTOMATICALLY IN SPOT. Input syntactic trees can also be created algorithmically, instead of manually. This is much quicker—an obvious practical advantage. The primary theoretical advantage to this approach is that the analyst can be sure of having formally defined the space of inputs. However, algorithmically generating input trees offers less fine-grained control over input representations, since it is not possible to adjust individual trees, nor is it possible to label projections as functional or silent.

To use SPOT's automated syntactic tree generation, click on the **Automatic** tab under **GEN: Input parameters**. Leave everything under **Syntax Parameters** on its default settings for this

system. These parameters can be adjusted to exclude adjunction structures, root syntactic trees in the CP level rather than XP, restrict head alignment, and so on.

Click on **Visibility to phonology** to expand it, and check “Treat non-branching XPs as X^0 s.” This excludes non-branching XPs from the syntactic representation as an analytical simplification; it is not meant as a theoretical claim. It is equivalent to generating the subset of a system without this assumption in which $\text{BINMIN-}\phi \gg \text{MATCHXP}$.

Next, under **Specify terminals**, type “a b c” in the box labeled “String of terminals 1.” Then click **Add terminal string** and type “a b c d” in the box labeled “String of terminals 2.” Finally, click the orange button marked **Generate trees**. In the text area below, a table that looks like Figure 5 should appear.

1.	[a [b c]]	
2.	[[a b] c]	
1.	[a [b [c d]]]	= 4wR
2.	[a [[b c] d]]	= 4wM
3.	[[a b] [c d]]	
4.	[[a [b c]] d]	
5.	[[[a b] c] d]	= 4wL

Figure 5. Algorithmically generated inputs for Msp.Asp

These are bracket representations of the syntactic trees for Msp.Asp. Square brackets [] stand for XP boundaries, and the terminals a, b, c, d stand for X^0 s.³

4. Defining outputs. We formally define the output structures of the system Msp.Asp in (3).

- (3) Msp.Asp. GEN Outputs = All possible phonological phrases ϕ for which
- Every syntactic word X^0 in the input is mapped to an output phonological word ω .
Linear order is preserved.
 - All non-terminal nodes are of category ϕ (represented by parentheses).
 - Output representations are trees with ordered leaves.
 - The child of a unary-branching ϕ must be a ω .

This set of prosodic trees can be generated in SPOT by adjusting the settings in the next section, **GEN: Output parameters**. Leave the first checkbox for “No prosodic recursion” unchecked, to allow recursive prosodic structures. (You can check “Enforce headedness” and/or “No level-skipping (enforce exhaustivity)” if you like, but these will be rendered redundant by the next step.)

Next, click **Prosodic categories** to show additional GEN options. This allows the user to determine the prosodic category labels for the root, intermediate, and terminal nodes. For Msp.Asp, we will select ϕ in the first row, under “Root prosodic tree in.” This will ensure that all output prosodic trees are rooted in ϕ s (phonological phrases), rather than the default ι (intonational phrase).⁴ Leave “Intermediate nodes are” and “Prosodic terminals are” at their

³ Since the input syntactic trees of Msp.Asp are rooted in XPs, there are no curly braces { }, which SPOT uses to represent CP boundaries in bracketed representations of syntactic trees; changing the root category under Syntax parameters from XP to CP would alter this.

⁴ This halves the number of output candidates (since the first node to contain all the terminals in the tree can only be

default values. These should be ϕ and ω , respectively. With these settings, only two prosodic categories (ϕ and ω) are included in each tree, so all structures will necessarily satisfy EXHAUSTIVITY, since every word will be parsed into a phonological phrase. Likewise, all candidates will necessarily satisfy HEADEDNESS, because every ϕ will certainly contain at least one ω .

The prosodic trees whose characteristics can be determined in this section will not be viewable until the final violation tableaux are visible, unlike the syntactic trees produced in **GEN: Input Parameters**. That is because the prosodic trees are so numerous (see Figure 1).

5. Selecting constraints. Because of the asymmetries in Japanese phonological phrasing, which were noted in the discussion of Table 1 (see Bellik et al. to appear), we define CON in Msp.Asp as in (4). This CON includes both MATCH and ALIGN constraints, as well as three constraints on BINARITY.

- (4) Msp.Asp.CON
- a. Mapping constraints
 - (i) MATCH(XP, ϕ) Assign one violation for every node of category XP in the syntactic tree such that there is no node of category ϕ in the prosodic tree that dominates all and only the same terminal nodes.
 - (ii) ALIGN(XP,L, ϕ ,L) Assign a violation for every syntactic node of category XP whose left edge is not aligned with the left edge of a prosodic node of category ϕ .
 - (iii) ALIGN(XP,R, ϕ ,R) Assign a violation for every syntactic node of category XP whose right edge is not aligned with the right edge of a prosodic node of category ϕ .
 - b. Markedness constraints:
 - (i) BINMIN(ϕ ,branches) Assign one violation for every node of category ϕ in the prosodic tree that has less than two children.
 - (ii) BINMAX(ϕ ,branches) Assign one violation for every node of category ϕ in the prosodic tree that has more than two children.
 - (iii) BINMAX(ϕ , ω)
Assign one violation for every node of category ϕ in the prosodic tree that dominates more than two nodes of category ω .

In the Japanese phrasing pattern that this constraint set is designed to account for, a uniformly left-branching input syntax is rebracketed into a balanced, symmetric prosody (5a). This rebracketing has previously been analyzed with combinations of Match and Binarity constraints (Ishihara 2014, Ito and Mester 2013, Kalivoda 2018, Selkirk 2011). However, as detailed in Bellik et al. (to appear), existing analyses fall short when other four-word syntactic structures are admitted to the candidate set. Both MATCH and BINARITY constraints are symmetric, so they cannot distinguish between the uniformly left-branching input that should be rebracketed, and the uniformly right-branching input that should be matched (5b).

- (5) Asymmetry in Japanese syntax-prosody mapping
- a. Left-branching mismatch: $[[[a\ b]\ c]\ d] \rightarrow ((a\ b)(c\ d))$
 - b. Right-branching match: $[a\ [b\ [c\ d]]] \rightarrow (a\ (b\ (c\ d)))$

a ϕ , rather than either a ϕ or an ι), and is equivalent to examining the subset of the typology in which MATCH-XP_{Max} (Ishihara 2014) is undominated.

Existing asymmetric markedness constraints such as STRONGSTART cannot solve this problem, so we introduce asymmetric mapping constraints in the form of Alignment to resolve the matter. We refer the reader to Bellik et al. (to appear) for a full explanation of the typology and findings from this system.

5.1. SELECTING MAPPING CONSTRAINTS. To add the mapping constraints to Msp.Asp on the SPOT interface, scroll down to **Mapping constraints**, then click **Match** to view the Match Theory constraints. All constraint definitions can be seen on the interface by clicking on the info buttons next to each constraint name. Select “Match(Syntax→Prosody).” We will keep the default category, “XP.” Although we will not use them in Msp.Asp, other MATCH constraints are available on SPOT, and can be accessed by clicking “Show more” at the bottom of the **Match** section.

Next, click **Align/Wrap** to view alignment constraints. Select two constraints: “Align-Left(Syntax→Prosody),” “Align-Right(Syntax→Prosody).” Keep the default category “XP” for these too. As with Match, other versions of Align become available by clicking “Show more.”

5.2. SELECTING MARKEDNESS CONSTRAINTS. Markedness constraints can be added in the same manner. Under **Markedness constraints**, click **Binarity** to view the binarity constraints. Under ...*counting branches*, select “BinMin(branches)” and “BinMax(branches).” Under ...*counting leaves*, select “BinMax(leaves).” Leave the default category settings at “φ.” Numerous other versions of both Binarity and other prosodic well-formedness constraints, such as EQUALSISTERS, STRONGSTART, and ACCENTASHEAD are available on SPOT as well,⁵ but do not feature in Msp.Asp.CON.

6. Violation tableaux and typology. At this point, both GEN and CON are defined, so the system definition on SPOT is complete. Scroll to the bottom of the interface and click the **Get results** button to download and view the violation tableaux. A dialog box will appear asking if you want to save the VT as a .csv file; save it for importing into OTWorkplace or another OT tool of your choice.

Another option is to click on the **Save** button to download these settings as a .spot file, in case you want to load them again later. (Since .spot files are plain text, they can be viewed in any text editor.) You can use the “Load” button on SPOT to upload this analysis again at another time to regenerate the violation tableaux or revise the analysis.

Once you have downloaded the violation tableaux, then continue the investigation by opening OTWorkplace⁶ and import the .csv into Excel. (Alternatively, open the .csv in Excel and then open OTWorkplace, or open the .csv in another editor, select all, and copy-paste to add it to OTWorkplace.) Click “Add ins” to display the OTWorkplace menu. Under OTWorkplace, click “Project start...” and name the new project. To calculate the factorial typology, under OTWorkplace, click “Factorial typology > FacType Calculate.” OTWorkplace will generate the typology predicted by the Msp.Asp, which has fourteen languages; one contains the Japanese phrasing pattern shown in Table 1. For more on typologies and languages in OT, see Prince (2016), Alber et al. (2016).

7. Conclusion. SPOT makes it easy to quickly create complete violation tableaux for an OT analysis of syntax-prosody interactions. SPOT implements many of the commonly used

⁵ To request a constraint or feature currently missing from SPOT, email the first author or submit an issue on Github (<https://github.com/syntax-prosody-ot/main/issues>)

⁶ OTWorkplace can be downloaded and installed on Windows from <https://sites.google.com/site/otworkplace/>.

constraints in syntax-prosody mapping, including constraints from both Match Theory and Align Theory, so that analyses couched in both frameworks can be created and compared directly. In fact, the system created here (Msp.Asp) is but one of several that we compared in Bellik et al. (to appear), a selection of which can be accessed via the Built-in systems menu at the top of the SPOT interface. Msp.Asp is represented there as “Japanese ✓: MatchSP, AlignSP (Bellik, Ito, Kalivoda, & Mester to appear)”; selecting that menu item is a shortcut to bringing up the entire system. Others can also be selected to view alternative analyses that do (✓) or do not (✗) capture the asymmetric Japanese phrasing pattern.

The violation tableaux created by SPOT can be analyzed by hand, but ideally, they should be imported into other software for Optimality Theory analyses, such as OTWorkplace or OTSoft, which can quickly generate the typologies that those systems predict. By combining SPOT with OT software, syntax-prosody analysts can create more theoretically rigorous analyses of phenomena at the syntax-prosody interface.

References

- Alber, Birgit, Natalie DelBusso & Alan Prince. 2016. From intensional properties to universal support. *Language* 92(2). e88–e116. <https://doi.org/10.1353/lan.2016.0029>.
- Bane, Max & Jason Riggle. 2012. Consequences of candidate omission. *Linguistic Inquiry* 43(4). 695–706. https://doi.org/10.1162/ling_a_00112.
- Bellik, Jennifer, Ozan Bellik & Nick Kalivoda. 2015–2021. Syntax-prosody in Optimality Theory (SPOT). Javascript application. <http://spot.sites.ucsc.edu>. Codebase at <https://github.com/syntax-prosody-ot>.
- Bellik, Jennifer, Junko Ito, Nick Kalivoda & Armin Mester. To appear. Matching and alignment. In Haruo Kubozono, Junko Ito & Armin Mester (eds.), *Prosody and prosodic interfaces*. Oxford, UK: Oxford University Press.
- Ishihara, Shinichiro. 2014. Match theory and the recursivity problem. In Shigeto Kawahara & Mika Igarashi (eds.), *Proceedings of FAJL 7: Formal Approaches to Japanese Linguistics* (MIT Working Papers in Linguistics 73), 69–88.
- Ito, Junko & Armin Mester. 2003/1992. Weak layering and word binarity. In Takeru Honma, Masao Okazaki, Toshiyuki Tabata and Shin-ichi Tanaka (eds.), *A new century of phonology and phonological theory. A Festschrift for Professor Shosuke Haraguchi on the occasion of his sixtieth birthday*, 26–65. Tokyo: Kaitakusha.
- Ito, Junko & Armin Mester. 2013. Prosodic subcategories in Japanese. *Lingua* 124. 20–40. <https://doi.org/10.1016/j.lingua.2012.08.016>.
- 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.
- Kubozono, Haruo. 1989. Syntactic and rhythmic effects on downstep in Japanese. *Phonology* 30(1). 39–67.
- Nespor, Marina & Irene Vogel. 1986. *Prosodic phonology*. Reprinted 2008 by Mouton de Gruyter. Dordrecht: Foris Publications. <https://doi.org/10.1515/9783110977790>.
- Prince, Alan. 2017. What is OT? Slides. http://roa.rutgers.edu/content/article/files/1513_prince_4.pdf.
- Prince, Alan. 2017. OT Checklist. http://roa.rutgers.edu/content/article/files/1615_prince_2.pdf.
- Prince, Alan, Nazarré Merchant & Bruce Tesar. 2020. OTWorkplace.

<https://sites.google.com/site/otworkplace/>.

- Selkirk, Elisabeth. 1978. On prosodic structure and its relation to syntactic structure. In *Nordic prosody II* ed. Fretheim T, 111–40. Trondheim: TAPIR .
- Selkirk, Elisabeth. 1980. Prosodic domains in phonology: Sanskrit revisited. In Mark Aronoff & M.-L. Kean (eds.), *Juncture* (vol. 7), 107–29. Saratoga, CA: Anma Libri.
- Selkirk, Elisabeth. 1984. *Phonology and syntax: The relation between sound and structure*. Cambridge, MA: MIT Press.

Appendix: Other inputs in Msp.Asp. Msp.Asp as defined above has 7 inputs, those in the 3w and 4w columns. Logically, we could also expand its input set to include analogous trees with one and two terminals, those in the 1w and 2w columns below.⁷

1w	2w	3w	4w
[a]	[a b]	[a [b c]] [[a b] c]	[a [b [c d]]] [a [[b c] d]] [[a b] [c d]] [[a [b c]] d] [[[a b] c] d]

Table 3. Extended inputs of Msp.Asp

We have calculated Msp.Asp with all of the inputs, and its FacTyp contains the 14 languages we just encountered with only three inputs (4wR, 4wM, 4wL). By removing inputs, we show that [a [b [c d]]], [[[a b] c] d], and one of either [a [[b c] d]] or [[a [b c]] d] are a universal support for Msp.Asp. As for the other inputs, the reasons they are redundant are explained in Table 4.

Input	Comment
[a]	Msp.Asp.GEN admits only one candidate for this cset: [a]→(a)
[a b]	Only 1 candidate in the cset is an optimum: [a b]→(a b)
[a [b c]]	Only 1 candidate in the cset is an optimum: [a [b c]]→(a (b c))
[[a b] c]	Only 1 candidate in the cset is an optimum: [[a b] c]→((a b) c)
[[a b] [c d]]	Only 1 candidate in the cset is an optimum: [[a b] [c d]]→((a b) (c d))
[[a [b c]] d]	In every language, this input behaves the same way as [a [[b c] d]]. <i>Matching case:</i> If [a [[b c] d]]→(a ((b c) d)) is optimal, [[a [b c]] d]→((a (b c)) d) is optimal. <i>Squishing case:</i> If [a [[b c] d]]→(a (b c) d) is optimal, [[a [b c]] d]→(a (b c) d) is optimal. <i>Rebracketing case:</i> If [a [[b c] d]]→((a b) (c d)) is optimal, [[a [b c]] d]→((a b) (c d)) is optimal.

Table 4. Redundant inputs

⁷ The number of inputs on n words is Catalan number n–1. The nth Catalan number is (2n)!/(n+1)!n! for n≥0, and 1 for n=0 .