SPOT Tutorial

Jenny Bellik (jbellik@ucsc.edu) & Nick Kalivoda
95th Annual (Virtual) Meeting of the Linguistic Society of America, 2021

SPOT stands for Syntax-Prosody in Optimality Theory. It is a JavaScript application (Bellik, Bellik, & Kalivoda 2015–2021) that automates candidate generation (GEN) and constraint evaluation (CON) for work on the syntax-prosody interface.

- Website: http://spot.sites.ucsc.edu
- App: https://people.ucsc.edu/~jbellik/spot/interface1.html
- Codebase: https://github.com/syntax-prosody-ot

Currently supported by NSF Grant #1749368, awarded to Junko Ito and Armin Mester.

Why automate?

- In order to create a valid analysis in Optimality Theory (OT), we must formally define the *OT system S* that we are studying.
  - S.GEN: the set of input and output candidates (algorithmically defined) in *S*
  - S.CON: the set of constraints in *S*
  - For information on systems, consult Alan Prince's OT Check List. For a more verbose introduction, see Prince's What is OT?

- In the view of syntax-prosody mapping assumed by SPOT, inputs are syntactic trees consisting of heads (X^0), phrases (XP), and clauses (CP). Outputs are prosodic trees consisting of prosodic words (ω), phonological phrases (φ), and intonational phrases (ι).

- 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).
  - Most of these will ultimately be harmonically bounded, yet all must be considered lest an omitted candidate invalidate the final analysis (Bane & Riggle 2012, Karttunen 2006).

Figure 1: Number of prosodic parses with different

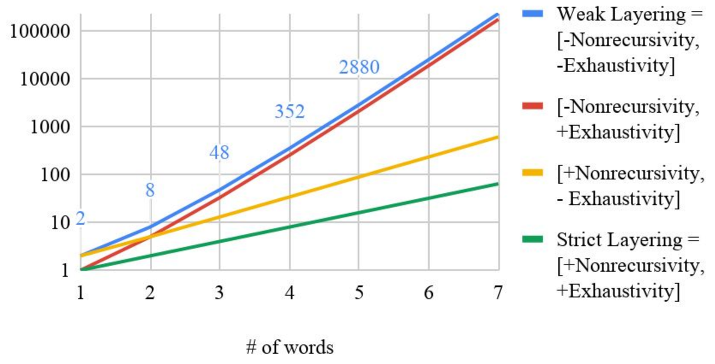
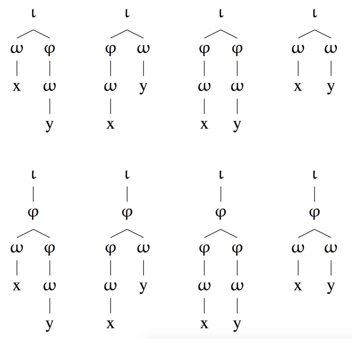


Figure 2: Parses of x, y with weak layering



- These huge numbers of candidates are the primary motivation for automation via SPOT.
  - 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 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.

Today, we will build a simplified version of a system that analyzes syntax-prosody matches and mismatches, inspired by data from Japanese. We will call this system **Match(Syntax → Prosody), Align(Syntax → Prosody) = Msp.Asp**, and it will feature in Nick Kalivoda's talk immediately after this tutorial.

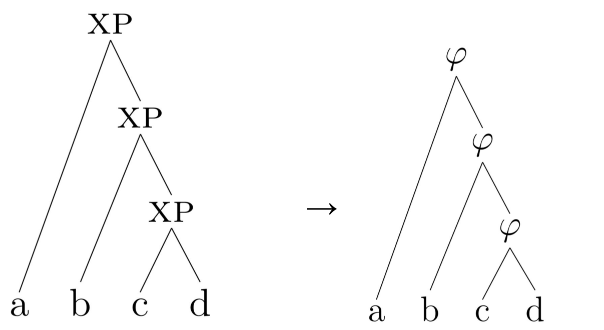
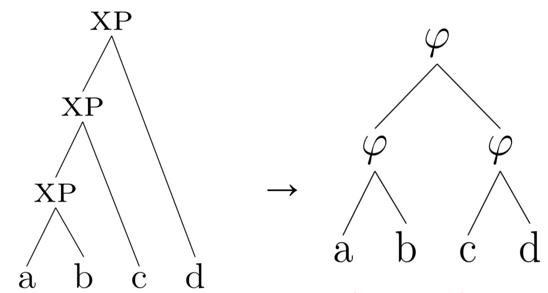
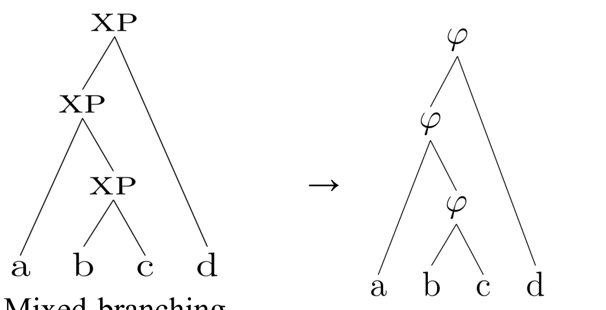
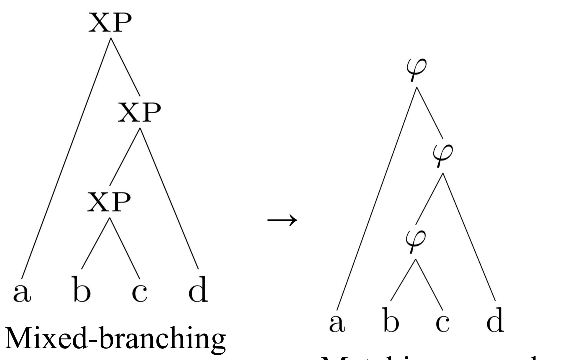
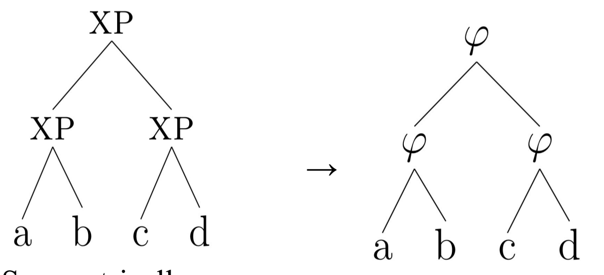
Step 0: Getting to the SPOT interface

1. Navigate to the SPOT web interface.
   a. Open the SPOT website in Chrome or Firefox: spot.sites.ucsc.edu
   b. At the top (or bottom) of the page, click "SPOT webapp". This will take you to the SPOT interface.
   c. *Alternatively*: For local access, click "SPOT codebase" to download the codebase from Github, and follow the instructions in the README.md.

Step 1: Defining inputs

The inputs to Msp.Asp are inspired by data on Japanese phrasing in Kubozono 1989 (Table 1).

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

 Right-branching syntax → Matching prosody	 Left-branching syntax → **Mismatching prosody**
 Mixed-branching syntax (Left + Right) → Matching prosody	 Mixed-branching syntax (Right + Left) → Matching prosody
 Symmetrically branching syntax → Matching prosody	

We define the set of inputs for our system Msp.Asp (1) to include two three-word syntactic trees in addition to the four-word trees in Table 1. We must build, at minimum, the three trees in (2), which form a universal support for Msp.Asp.

(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, non-terminal nodes are syntactic phrases XP.
   (c) The orthographic representation of the bracketing structure imposes a linear order on the leaves.

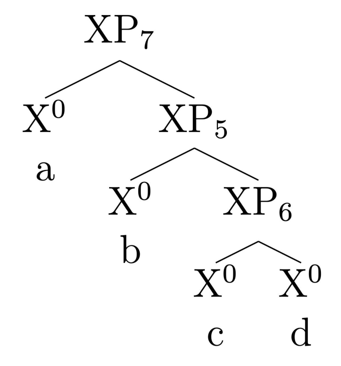
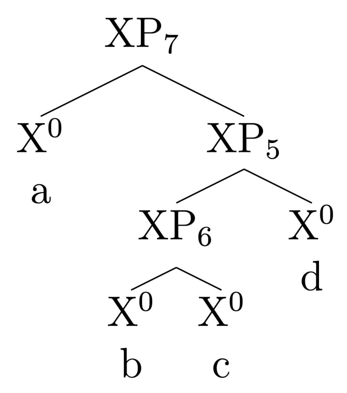
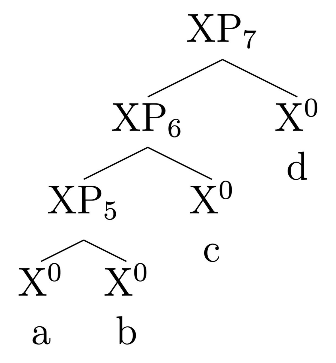
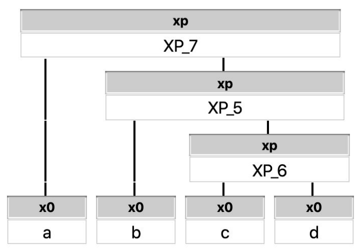
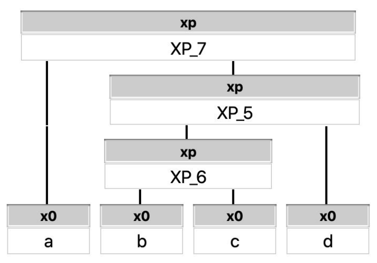
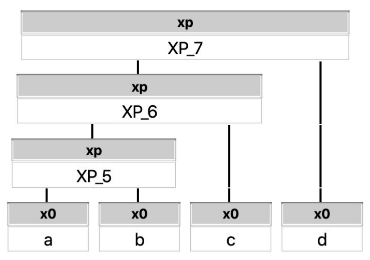
For the full set of inputs, see the appendix.

(2) A universal support for the system Msp.Asp:

(a)	[a [b [c d]]]	Four words, right-branching = 4wR
(b)	[a [[b c] d]]	Four words, mixed-branching = 4wM
(c)	[[[a b] c] d]	Four words, left-branching = 4wL

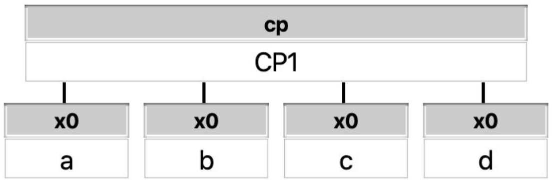
The labels of the words are arbitrary, but must be distinct. In SPOT, we will give arbitrary labels to the XPs as well, simply so SPOT can tell them apart.

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

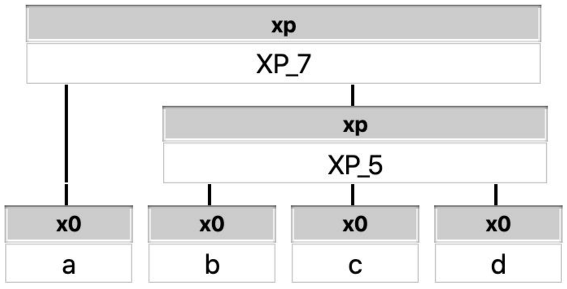
	4wR	4wM	4wL
Traditional tree view	 XP7 X0 a XP5 X0 b XP6 X0 c X0 d	 XP7 X0 a XP5 XP6 X0 d X0 b X0 c	 XP7 XP6 X0 d XP5 X0 c X0 a X0 b
SPOT tree view	 xp XP_7 xp XP_5 xp XP_6 x0 a x0 b x0 c x0 d	 xp XP_7 xp XP_5 xp XP_6 x0 a x0 b x0 c x0 d	 xp XP_7 xp XP_6 xp XP_5 x0 a x0 b x0 c x0 d

Building syntactic trees manually in SPOT

2. In the **Manual** tab under **GEN: Input parameters**, type "a b c d" into "String of terminals" and hit "Build syntax". This will open the tree builder, which will include the seed of your syntactic tree:



In this tree representation, every node has two attributes, a category and an id (identifier).
   a. The category, shown with a grey background, must be cp (complementizer phrase), xp (syntactic phrase) or x0 (syntactic head), if the node is to be visible to the syntax-prosody mapping constraint.
   b. The id (identifier) can be any alphanumeric string (no spaces or hyphens, please), and must be unique.
   c. In this handout we'll use the notational convention cat–id to refer to nodes.
3. The root node of this tree currently has the id "CP1" and the category "cp". We need to change these to "XP_7" (or some other id of your choosing) and "xp", respectively.
4. To create the XP containing "b c d":
   a. 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.
   b. Now do the same to select x0–c and x0–d.
   c. Click "Add Mother". This will create the node xp–XP_5. The tree should now be:



5. Now create an XP above x0–c and x0–d. This will be labelled XP_6, and will complete the tree.

6. Now click "Done! Add trees to analysis." A message will appear saying "The trees in the analysis are up-to-date".
    a. You can see the JavaScript code for the trees by clicking "Show code".
7. 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 in (3-7), to create the other two trees (4wM and 4wL).

Building syntactic trees automatically in SPOT

Input syntactic trees can also be created algorithmically, instead of manually. This is much quicker!

8. Under **GEN: Input parameters**, click on the **Automatic** tab.
9. Leave everything under **Syntax Parameters** on its default settings.
10. Click on **Visibility to phonology** to expand it, and check "Treat non-branching XPs as X^0s."
    a. This excludes non-branching XPs from the syntactic representation as an analytical simplification; cf. the subset of a system without this assumption in which BinMin-phi >> MatchXP.
11. 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."
12. Click the orange button marked **Generate trees.** You should see the following in the text area below:

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

Step 2: Defining outputs

Outputs in Msp.Asp

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
   (a) Every syntactic word X^0 in the input is mapped to an output phonological word ω. Linear order is preserved.
   (b) All non-terminal nodes are of category φ (represented by parentheses).
   (c) Output representations are trees with ordered leaves.
   (d) The child of a unary-branching φ must be a ω.

Generating prosodic trees in SPOT

13. Scroll down to **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.

14. Click **Prosodic categories** to show additional GEN options.
    a. For "Root prosodic tree in", select φ. This will ensure that all output prosodic trees are rooted in φs (phonological phrases), rather than the default ι (intonational phrase).
    b. Leave "Intermediate nodes are" and "Prosodic terminals are" at their default values. These should be φ and ω, respectively.

Step 3: Constraints

For reasons that will be discussed in the next talk, we’ll define CON in Msp.Asp as in (4):

(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.

Step 3.1 Selecting mapping constraints

15. Under **Mapping constraints**, 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.
    a. Select Match(Syntax→Prosody). We will keep the default category, "XP".

16. Click **Align/Wrap** to view alignment constraints. Select two constraints:
    a. Align-Left(Syntax→Prosody)
    b. Align-Right(Syntax→Prosody).
    c. Keep the default category "XP" for these too.

Step 3.2 Selecting markedness constraints

17. Under **Markedness constraints**, click **Binarity** to view the binarity constraints.
    a. Under *...counting branches,* select BinMin(branches) and BinMax(branches).
    b. Under *...counting leaves,* select BinMax(leaves).
    c. Leave the default category settings at "φ".

Step 4+: Violation tableaux and typology

18. 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.
19. You can also click on the **Save** button to download these settings as a .spot file, in case you want to load them again later.
20. Open OTWorkplace (which can be downloaded and installed on Windows from https://sites.google.com/site/otworkplace/) and import the .csv into Excel. (Or 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.)
    a. Click "Add ins" to display the OTWorkplace menu.
    b. Under OTWorkplace, click "Project start…" and name the new project.
    c. To calculate the factorial typology, under OTWorkplace, click Factorial typology > FacType Calculate.

Conclusion

- SPOT makes it easy to quickly create complete violation tableaux for an OT analysis of syntax-prosody interactions, which can be further analyzed using other OT software.
- Questions about how to use SPOT for a particular analysis? Put them in the Q&A box, or contact me later via email (jbellik@ucsc.edu) or by submitting an issue on Github (https://github.com/syntax-prosody-ot/main/issues).

Supplementary materials

Built-in system of Msp.Asp

Shortcut: This system and a number of other related systems can be found in SPOT in the **Built in systems** menu at the top of the page. Msp.Asp is listed there as ✓ *Japanese: Match SP, Align SP (Bellik, Ito, Kalivoda & Mester to appear)*.

System Msp.Asp at a glance

(5) Msp.Asp.GEN

(a) **Inputs**: Binary-branching syntactic trees on terminal strings of 1-4 nodes, where terminal nodes are syntactic words X^0, non-terminal nodes are syntactic phrases XP. The orthographic representation of the bracketing structure imposes a linear order on the leaves.

(b) **Outputs**: All possible phonological phrases for which

(i) Every syntactic word X^0 in the input is mapped to an output phonological word ω. Linear order is preserved.

(ii) All non-terminal nodes are of category φ (represented by parentheses).

(iii) Output representations are trees with ordered leaves.

(iv) The child of a unary-branching φ must be a ω.

(6) 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) BinMax(φ,branches)
Assign one violation for every node of category φ in the prosodic tree that has more than two children.

(ii) BinMin(φ,branches)
Assign one violation for every node of category φ in the prosodic tree that has less 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.

What about the 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:

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]

Note: the number of inputs on *n* words is Catalan number *n*–1. The *n*th Catalan number is (2*n*)!/(*n*+1)!*n*! for n≥0, and 1 for n=0 (Wikipedia: Catalan numbers).

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:

Redundant 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 its counterpart [a [[b c] d]]. *Matching case*: If [a [[b c] d]]→(a ((b c) d)) is optimal, then [[a [b c]] d]→((a (b c)) d) is optimal. *Squishing case*: If [a [[b c] d]]→(a (b c) d) is optimal, then [[a [b c]] d]→(a (b c) d) is optimal. *Rebracketing case*: If [a [[b c] d]]→((a b) (c d)) is optimal, then [[a [b c]] d]→((a b) (c d)) is optimal.

References

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

Bellik, Jennifer, Ozan Bellik, and 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, and Armin Mester. To appear. Matching and Alignment. In Haruo Kubozono, Junko Ito, and Armin Mester: *Prosody and Prosodic Interfaces.* Oxford University Press.

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.

Prince, Alan. 2017. What is OT? Slides. http://roa.rutgers.edu/content/article/files/1513_prince_4.pdf

–. 2017. OT Checklist. http://roa.rutgers.edu/content/article/files/1615_prince_2.pdf

Prince, Alan, Nazarré Merchant, and Bruce Tesar. 2020. OTWorkplace. https://sites.google.com/site/otworkplace/