phono*kit*

v 0.4.1

A toolkit to create phonological representations in Typst

Guilherme D. Garcia

Pavillon Charles-De Koninck, 1030, Bureau DKN-3267
Département de langues, linguistique et traduction, Université Laval
Avenue des Sciences-Humaines, Québec QC, Canada
`gdgarcia.ca` `guilhermegarcia` `guilherme.garcia@lli.ulaval.ca`

DOI 10.5281/zenodo.17971032

Abstract

Creating high-quality phonological representations in academic documents is often time-consuming and requires juggling multiple tools and LaTeX packages (most notably `TikZ`). This vignette introduces **phono*kit***, an open-source Typst package designed to streamline the creation of phonological structures while maintaining typographical precision. The package provides intuitive functions for IPA transcription (supporting `tipa` input conventions), prosodic representations (syllables, moras, feet, prosodic words, metrical grids), sonority profiles, IPA vowel charts and consonant tables with built-in language inventories, autosegmental phonology (feature spreading, tone, delinking), multi-tier representations for complex non-linear structures such as Government Phonology, CV phonology, SPE feature matrices, Optimality Theory tableaux, Harmonic Grammar, Noisy Harmonic Grammar, Maximum Entropy grammars, and Hasse diagrams for constraint rankings. The package also includes numbered linguistic examples and a collection of helper symbols for Greek letters and arrows. All functions prioritize minimal input syntax while automatically handling typographical challenges such as dynamic spacing and proper alignment. The font used by the package can be customized globally. **phono*kit*** aims to reduce cognitive load in document preparation, allowing phonologists to focus on content rather than formatting. The package is freely available through Typst's package repository and is under active development.

How to keep track of updates? See table of contents on the next page.

● = new feature ● = recent update or change

Version history

- `0.4.5` - Vowel trapezoids accept arrows and shifted vowels
- `0.4.1` - Consonants table has abbreviation argument; alignment fix in `#consonants()`
- `0.4.0` - Multi-tier representations; sorted MaxEnt candidates; helper functions for Greek letters
- `0.3.7` - Font and prosodic symbols can now be chosen; aesthetic improvements to prosodic functions
- `0.3.6` - Improved IPA symbols, distances in `#syllable()`, and `README.md`
- `0.3.5` - Numbered examples; minor improvements in IPA symbols and several fixes in documentation
- `0.3.0` - Autosegmental phonology
- `0.2.0` - Sonority, tableaux and metrical grids
- `0.0.1` - Initial release with prosodic representations and IPA module

Table of contents

1. Introduction

This vignette introduces **phono*kit*** and its functions. If you haven't used Typst yet and want to follow along, go to Typst.app to use their online editor. You may want to check their own tutorial too (I have an introductory YouTube series here). This is **not** an introduction to Typst, but see Section C, Section D and Section E in the appendix for some useful info. The GitHub repository for the package can be found at `guilhermegarcia/phonokit`. Comments and suggestions are welcome, as are bug reports (open an issue in the package's repository). The main goals of **phono*kit*** are to MINIMIZE EFFORT and MAXIMIZE QUALITY.

1.1. Installation

Typst packages are always loaded the same way: using the `#import` function at the top of your `typ` document, as shown in Code 1. Replace `X.X.X` with the version you wish to import. The `*` simply states that we want to import all functions from the package in question. See package's page on Typst's website here.

```
#import "@preview/phonokit:X.X.X": *
```

Code 1: Loading a package in Typst using the official repository

Alternatively, if you want the most up-to-date version and this version is ahead of the published version, download/clone the repository here and load the package locally. This is shown in Code 2. You simply need to import the `lib.typ` file, which is present in any package.

```
#import "phonokit/lib.typ": *
```

Code 2: Loading a package in Typst using a local package

You may need to create a symlink depending on how you structure your files. See Section D for more information on Typst packages in general, as they work slightly differently from what you may be used to if you use LaTeX.

2. Font

All functions in **phono*kit*** require the Charis SIL font (SIL International, 2025) to work as intended out of the box. Thus, if you don't have the font, you will need to install it (unless you're ok with the fallback option in Typst). However, as of version `0.3.7`, the user can set a global font to be used by the package throughout the document. Code 3 shows how to set New Computer Modern as the main font for the package.

```
#import "@preview/phonokit:0.3.7": *
#phonokit-init(font: "New Computer Modern") // ← add to the top of your document
```

Code 3: Choosing a different font

3. IPA module

IPA transcription is likely the most commonly used feature when typesetting documents in phonology. **phono*kit*** accomplishes that with the `#ipa()` function, which takes a string as input. Crucially, the function uses the familiar `tipa` input (Rei, 1996), with a few exceptions (e.g., secondary stress is represented by a comma `,`, not by two double quotes `""`).

3.1. Phonemic transcription

As can be seen in Code 4, symbols introduced by two backslashes `\\` must not have adjacent characters. For archiphonemes, you can use `\\` followed by a capital letter of your choice. This is nice because you don't need to leave the function to render an archiphoneme, so the font will be automatically consistent across phonemes and archiphonemes. Thus, while `#ipa("N")` maps to "ŋ", `#ipa("\\N")` maps to "N" — see examples.

`#ipa("DIs \\s Iz \\s @ \\s sEn.t@ns")`	ðɪs ɪz ə sɛn.təns	*This is a sentence*
`#ipa("N \\N R \\R \\I I Z \\Z")`	ŋNɾRIɪʒZ	*Escaped characters*
`#ipa("p \\h I k \\* \\s \\t tS \\ae t \\s p \\r l iz")`	pʰɪk̚ t͡ʃæt pl̥iz	*Pick, chat, please*
`#ipa("'lIt \\v l \\s 'b2R \\schwar ,flaI")`	ˈlɪtl̩ ˈbʌɾɚˌflaɪ	*Little butterfly*

Code 4: Transcriptions using `#ipa()`

3.2. Phonemic inventories

3.2.1. Consonants

Two additional functions allow users to quickly create consonant tables and vowel trapezoids given a string of phonemes. Figure 1 shows the consonant inventory for Italian, for example. The function mirrors the pulmonic consonants table in the IPA chart with some minor changes. For example, affricates are shown when `affricates: true`, and /w/ is shown in the approximant row under both bilabial and velar columns (when /ɰ/ is not present, in which case /w/ appears only under bilabial). The argument `abbreviate: true` shortens labels for both rows and columns.

	Bilab	Labdent	Dent	Alv	Postalv	Retro	Pal	Vel	Uvu	Phar	Glot
Plos	p b			t d				k g			
Nas	m			n			ɲ				
Trill				r							
Tap/Flap											
Fric		f v		s z	ʃ						
Affr				ts dz	tʃ dʒ						
Lat fric											
Approx							j				
Lat approx				l			ʎ				

Figure 1: Italian consonants - `#consonants("italian", affricates: true, abbreviate: true)`

Aspirated consonants are shown when `aspirated: true`. In cases where neither `affricates` nor `aspirated` are set to `true`, the function will omit both groups and fewer rows will be printed.

The user can either input a language[1] (see caption of Figure 1) or a string of consonants to create a custom inventory (Figure 2) — the input follows the same format used by the `#ipa()` function discussed in Section 3.1, so `#ipa("\\*r")` generates "ɹ". Note, however, that both affricates and aspirated consonants require curly braces around them as well as `affricates: true` and `aspirated: true`, as shown in the caption of Figure 2. Finally, the function also allows for flexible sizing with the `scale` argument.

	Bilabial	Labiodental	Dental	Alveolar	Postalveolar	Retroflex	Palatal	Velar	Uvular	Pharyngeal	Glottal
Plosive	p			t				g			
Plosive (aspirated)								kʰ			
Nasal											
Trill											
Tap or Flap											
Fricative				s	ʃ						
Affricate				ts	tʃ						
Affricate (aspirated)											
Lateral fricative											
Approximant				ɹ							
Lateral approximant											

Figure 2: `#consonants("ts{ts}psS \\*r g{tS} {k \\h}", affricates: true, aspirated: true)`

3.2.2. Vowels ●

Besides the function `#consonants()`, the package has a function to print vowel inventories. The function `#vowels()` also accepts either a pre-defined language or a string as input. Figure 3 and Figure 4 show the inventories for English and French, respectively. The argument `scale` is also available here, so the user can adjust the size of the trapezoid as needed.

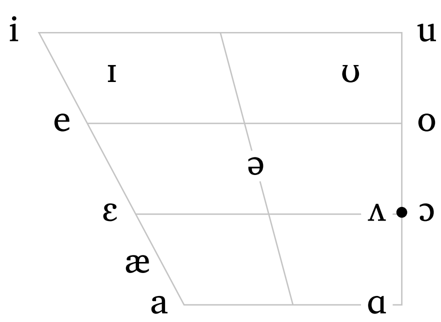


Figure 3: `#vowels("english", scale: 0.6)`

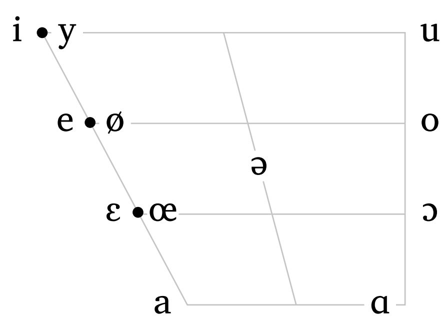


Figure 4: `#vowels("french", scale: 0.6)`

[1] Available languages: Arabic, English, French, German, Italian, Japanese, Portuguese, Russian and Spanish (all language names are lowercase in the function). You can also use `all` to display all consonants. The same applies to the function `#vowels()`.

As of version `0.4.5`, the function `#vowels()` also accepts a range of arguments to include arrows, shifted vowels, and highlights to a trapezoid. Figure 5, for example, illustrates some of the available arguments to create a trapezoid showing diphthongs in English. The arguments are intuitive and self-explanatory: you can use `arrows` to draw arrows between any pair of vowels in the trapezoid. You can choose if you want arrows to be curved (`curved: true`) or dashed (`arrow-style: dashed`), for example. You can also choose arrow colors (`arrow-color`), which is set to a shade of gray in Figure 5 using the `luma()` function (you can naturally simply use `gray` or `gray.darken(X%)` if you prefer).

```
#vowels(
    "english",
    arrows: (
      ("a", "U"),
      ("a", "I"),
      ("e", "I"),
      ("O", "I"),
      ("o", "U"),
    ),
    arrow-color: luma(200),
    curved: true,
    highlight: ("a", "e", "o", "O"),
    highlight-color: blue.lighten(80%),
  )
```

Code 5: Code to generate Figure 5

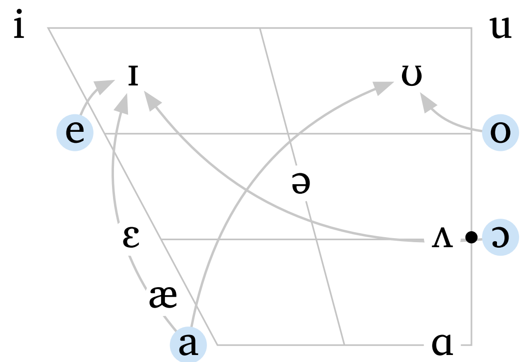


Figure 5: Diphthongs in North American English

The function `#vowels()` unsurprisingly is based on the assumption that vowels occupy a predetermined location in the trapezoid. As of version `0.4.5`, you can also shift said location, which allows for a much higher degree of flexibility when illustrating dialectal variation, for example. The argument `shift`, shown in Figure 6, allows you the specify a vowel and its shift from the original position of said vowel. Shifted vowels can then also be independently targeted by the `highlight` argument. In addition, you can customize arrows to target shifted vowels by using the same shifted values you defined for each vowel. Finally, shifted vowels can have their color and size adjusted with `shift-color` and `shift-size`, respectively. In the example shown in Figure 6, a custom vowel inventory is illustrated with three shifted vowels (in dark blue). The trapezoid also shows how to change the color and style of arrows.

As is often the case, a figure becomes too crowded. Once we start adding arrows and shifted vowels, a trapezoid may be less clear given the amount of information displayed. Figure 6 (bottom) shows how you can simplify the figure by removing grid lines (`rows` and `cols`). Because `#vowels()` allows you to add *a lot* of information to a trapezoid, this option can be useful. Last but not least, notice that the trapezoid at the bottom in Figure 6 is scaled down (by default, `#vowels()` uses `scale: 0.7`) to `0.4`. Everything in the trapezoid scales down accordingly, as expected.

```
#vowels(
    "aeiouIU",
    arrows: (
      ("a", ("U", -0.6, -0.3)),
      ("a", ("U", -0.3, -0.7)),
    ),
    arrow-color: blue.lighten(40%),
    arrow-style: "dashed",
    curved: true,
    shift: (
      ("a", 0.6, 0.3),
      ("U", -0.6, -0.3),
      ("U", -0.3, -0.7),
    ),
    shift-size: 1.5em,
    shift-color: blue.darken(20%),
    highlight: ("a", ("U", -0.6, -0.3)),
    highlight-color: blue.lighten(90%),
    rows: 0, // Fig at the bottom with
    cols: 0, // no grid lines
  )
```

Code 6: Code to generate Figure 6

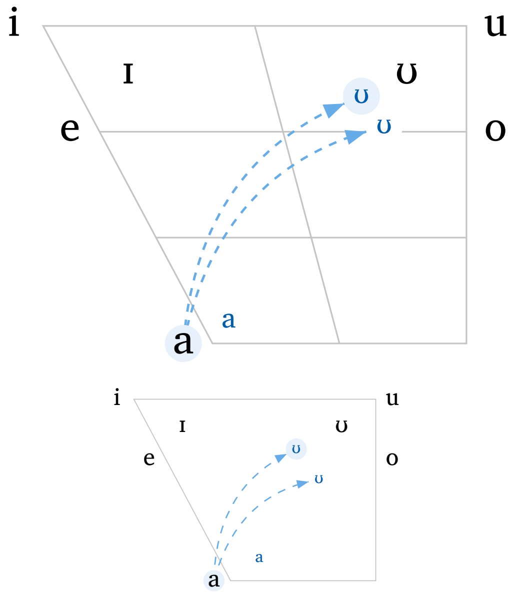


Figure 6: Flexible arrows and shifted vowels

4. Prosody module

4.1. Sonority

When discussing the sonority principle in introductory phonology courses, it is often useful to illustrate relative sonority with a visual representation. The function `#sonority()`, based on the Fonology package for R (Garcia, 2025), plots phonemes and their relative sonority profiles. The function is based on the sonority scale in Parker (2011, p. 18). You may want to customize the sonority scale in question to your needs and preferences. In that case, download or clone the package and adjust the scale as needed in the source code (`sonority.typ`).

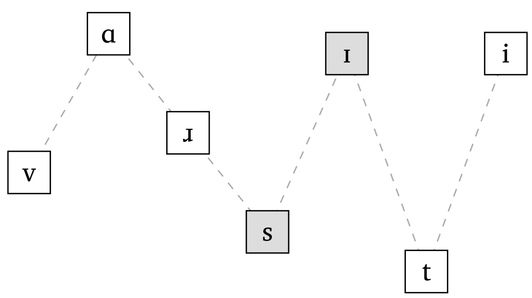


Figure 7: `#sonority("vA \\*r .sI.ti", scale: 0.7)`

Figure 7 shows an example for the word “varsity”. If syllable boundaries are detected in the input, the function alternates between white and gray fills to distinguish each syllable. If no boundaries are detected, all boxes will be white by default.

4.2. Syllables

We start with an essential representation, namely the syllable. Two options are available: `#syllable()` for a classic onset-rhyme representation (Figure 8), and `#mora(.., coda: true)` for a moraic representation (Figure 9). The latter option allows you to define whether or not codas project a mora (`coda: true`). These two functions are used for single-syllable representations only. As can be seen in the figures, these functions take as input a string that should be familiar given the discussion about `#ipa()` in Section 3.1.

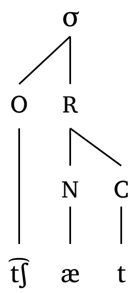


Figure 8: `#syllable("\\t tS \\ae t")`

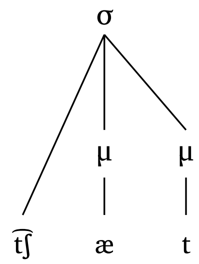


Figure 9: `#mora("\\t tS \\ae t", coda: true)`

Vowel length is also represented in both `#syllable()` and `#mora()`, as can be seen in Figure 10 and Figure 11, respectively. The crucial element here is the use of `:`, which triggers the ː symbol for both representations. In the moraic representation, two moras branch out of the vowel, as expected.

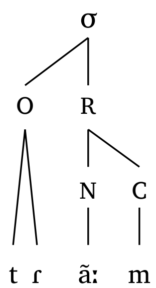


Figure 10: `#syllable("tR \\~ a:m")`

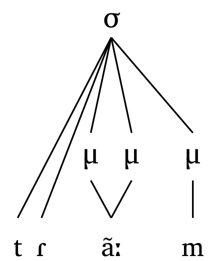


Figure 11: `#mora("tR \\~ a:m", coda: true)`

The dimensions of each representation adjust as a function of how many segments are found in the input. As such, more complex onsets, nuclei or codas result in wider representations that respect a safe and consistent between-segment distance. Figure 12 illustrates this with an extreme example.

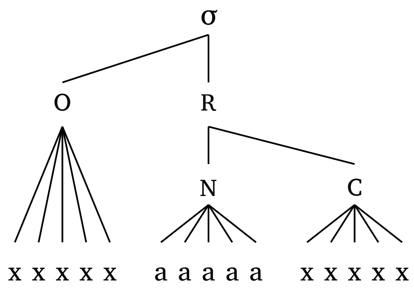


Figure 12: How spacing is managed

We often need to adjust the size of a representation as a whole. But doing so can be problematic if the text and the representation itself behave independently. Here, however, representations can be easily adjusted with the argument `scale`, which takes care of both line width and text size. Examples are shown in Figure 13, Figure 14 and Figure 15.

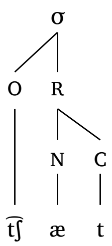


Figure 13: Scale 0.75

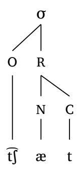


Figure 14: Scale 0.5

Figure 15: Scale 0.25

4.3. Feet

Next, we examine metrical feet (Figure 16 and Figure 17). These functions are designed to deal with a single foot where all syllables are footed by definition, since unfooted syllables have nowhere to attach to (see Section 4.4). A period `.` is used in the code to indicate syllabification and a single apostrophe `'` is used to indicate which syllable is the head of the foot. This allows us to easily create trochees and iambs. Naturally, you are free to generate non-binary feet, as the function can handle them as well (dactyls in Figure 18 and Figure 19).

Can I choose my own symbols? *Yes*. If you prefer "Ft" instead of Σ, for example, see Section B.

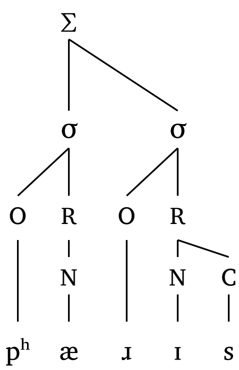


Figure 16: `#foot("'p \\h \\ae.\\*r Is")`

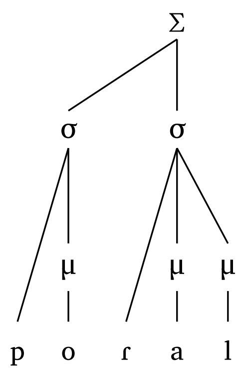


Figure 17: `#foot-mora("po.'Ral", coda: true)`

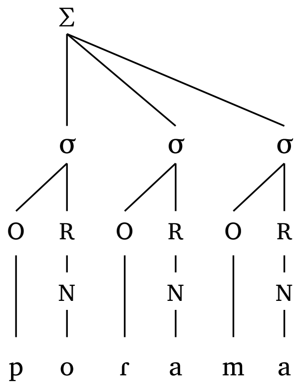


Figure 18: `#foot("'po.Ra.ma")`

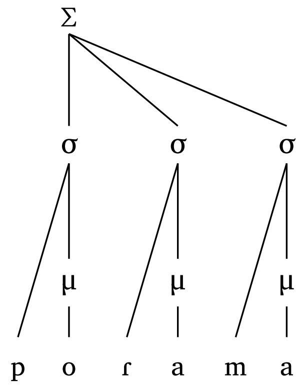


Figure 19: `#foot-mora("'po.Ra.ma")`

Geminates are also represented by the functions `#foot()` and `#foot-mora()`. In onset-rhyme representations, a geminate will be linked to the coda and the following onset, as expected. In moraic representations, the user will probably want to define `coda: true` to represent geminates in a traditional fashion. Figure 20 and Figure 21 show a disyllabic word containing a geminate in both onset-rhyme and moraic representations.

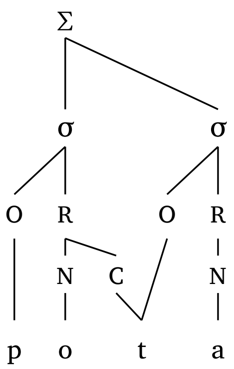


Figure 20: `#foot("'pot.ta")`

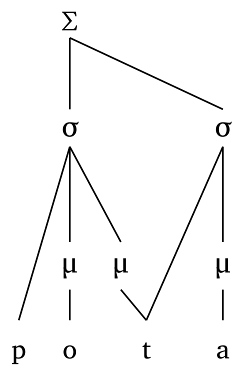


Figure 21: `#foot-mora("'pot.ta", coda: true)`

Extreme cases are important to test how adaptable the function is when it comes to line crossings, a key problem in prosodic representations. When the head of a domain (the foot here) is at an edge of a long string, it is challenging to avoid crossing or overlapping lines. As can be seen in Figure 22, the height of Σ is proportional to the width of the representation to avoid superposition of lines.

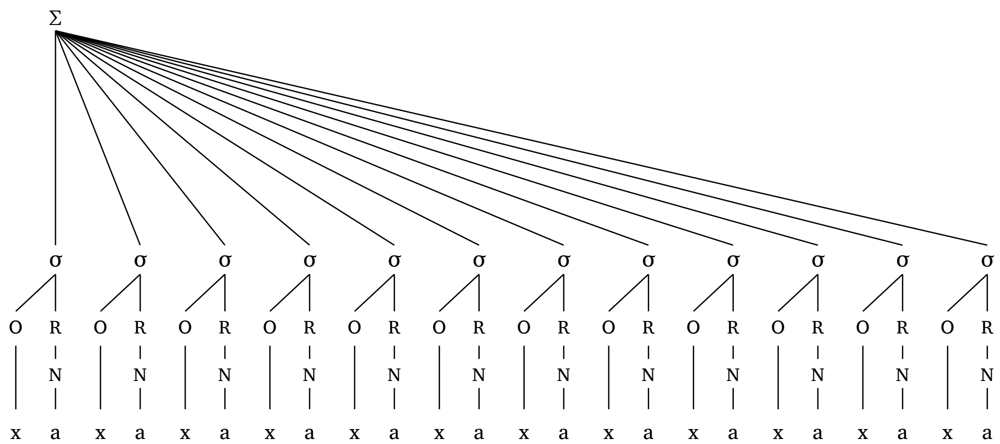


Figure 22: An extreme case

4.4. Prosodic words

Finally, we arrive at prosodic words (PWd), which bring together syllables and feet. This is where the user has more options, given the metrical parameters involved. Parentheses `()` are used to define feet, which means that any syllable *outside* the foot will be linked directly to the PWd. Next, an apostrophe `'` symbolizing stress (both primary *and* secondary) is used to indicate the head of each foot. Finally, the argument `foot: "R"` or `foot: "L"` is used to determine which foot in the PWd contains the primary stress in the word (in cases where more than one foot is present in a given PWd).

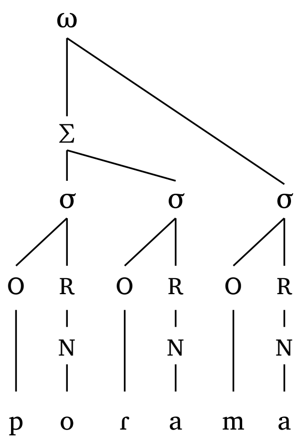


Figure 23: `#word("('po.Ra).ma")`

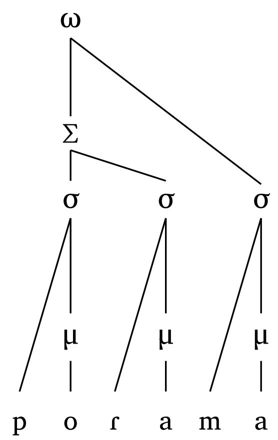


Figure 24: `#word-mora("('po.Ra).ma")`

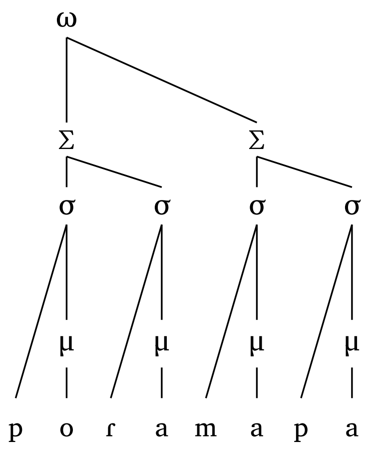


Figure 25: When `foot: "L"` (default)

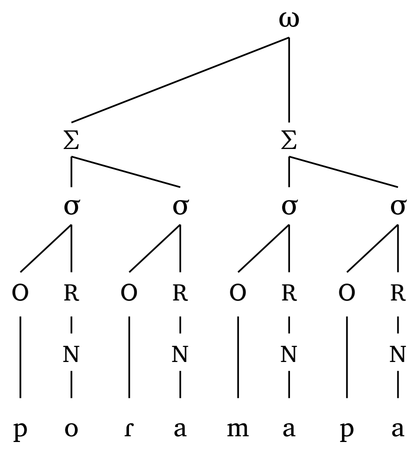


Figure 26: When `foot: "R"`

4.4.1. Extreme scenarios

It is worth noting that *all* lines are straight in the prosody module (this is by design), so curved lines are not a possibility. Consequently, in extreme scenarios, e.g., where an unfooted syllable is *very* far away from the head foot of a given PWd, the height of the representation will be adjusted accordingly to avoid line crossings. This will inevitably create taller figures, as already mentioned. Figure 27 presents a hypothetical scenario to demonstrate how the function deals with extreme cases.

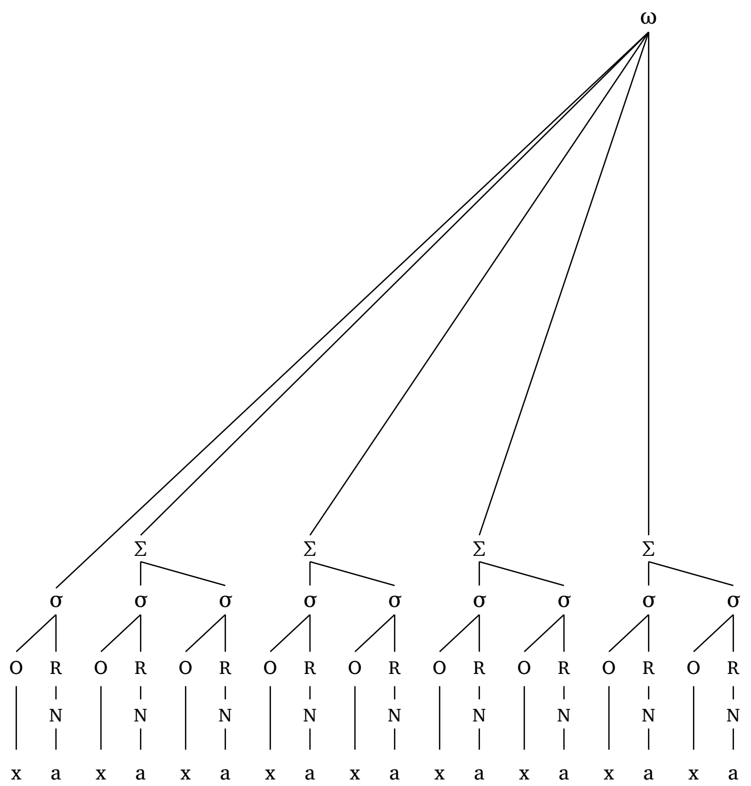


Figure 27: Unfooted syllable far away from head foot

4.5. Metrical grid

The function `#met-grid()` allows you to create a metrical grid using × to indicate prominence (e.g., Zsiga (2024, p. 373)). As with all functions in the package, the goal here is to create high-quality output with *minimal effort*, that is to say, with functions that are as intuitive and parsimonious as possible. Figure 28 and Figure 29 show grids for the word *butterfly*.

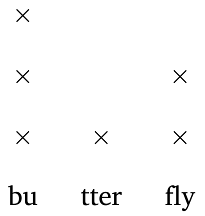


Figure 28: Input as string

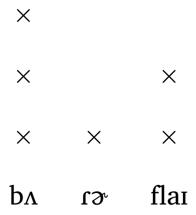


Figure 29: Input as tuple (IPA-compatible)

```
#met-grid("bu3.tter1.fly2") // Input as string
#met-grid(("b2", 3), ("R \\schwar", 1), ("flaI", 2)) // Input as tuple
```

Code 7: Metrical grids shown in Figure 28 and Figure 29

4.6. Autosegmental phonology

Another key function in **phono*kit*** is `#autoseg()`, introduced in version 0.3.5. The function allows you to represent either features or tones on a separate tier. More importantly, the function is able to show linking and delinking, floating tones, as well as contour tones. Figure 30 demonstrates how `#autoseg()` works in a simple scenario of feature spreading.

```
#autoseg(
    ("k", "\\ae", "n", "t"),
    features: ("", "", "[+nas]", ""),
    links: ((2, 1),),
    spacing: 1.0,
    arrow: false,
  )
```

Code 8: Code to generate Figure 30

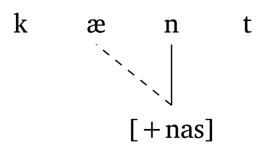


Figure 30: Feature spreading

As can be seen in the figure, `#autoseg()` accepts the same inputs as `#ipa()`. Here, inputs are arrays, so each phoneme must be entered individually. This results in more flexibility, as you can add empty spaces, symbols for domain boundaries, etc. The argument `features` works the same way. In this particular case, only one feature is present (in position 2, since we count from zero). That position is aligned with the segment “n”: this automatically draws a vertical line linking segment to feature. Indices are key because the argument `links` depends on them. For the spreading to be represented here, `links` is defined as **an array of tuples** `((2,1),)`, hence the commas. Read this as “draw a link from index 2 to index 1”. Finally, `spacing` allows you to conveniently set the inter-segmental spacing (lines and links are adjusted accordingly). The argument `arrow` simply asks whether an arrow head should be added to the linking line (`true` or `false`).

Multiple links are easy to implement. Figure 31 shows an example of metaphony in Brazilian Veneto (Garcia & Guzzo, 2023) where two vowels are targeted by the spreading of [+high]. The source of the process is position 6 (“i”) and the targets are positions 3 (“e”) and 1 (“o”), hence the array `((6,3), (6,1))` in the code. Finally, notice that the argument `gloss` allows for quick annotation.

```
#autoseg(
    ("z", "o", "v", "e", "n", "-", "i"),
    features: ("", "", "", "", "", "", "[+hi]"),
    links: ((6, 3), (6, 1)),
    spacing: 0.5, // better spacing
    arrow: true,
    gloss: [_young_.#smallcaps("pl")],
  )
```

Code 9: Code to generate Figure 31

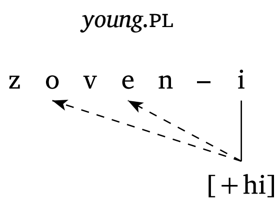


Figure 31: Metaphony in Brazilian Veneto

4.6.1. Tones

One of the arguments in `#autoseg()` is `tone` (`true` or `false`). This allows us to “flip” the representation vertically to show tones above the segmental tier. Take a look at Figure 32, which shows a case of low tone spreading without delinking in Nupe (Zsiga, 2024). We can easily add multiple instances of the function to represent processes with relatively concise code.

```
#autoseg(
  ("e", "b", "e"),
  features: ("L", "", "H"),
  spacing: 0.5,
  tone: true,
  gloss: [],
)
#a-r // arrow
#autoseg(
  ("e", "b", "e"),
  features: ("L", "", "H"),
  links: ((0, 2),),
  spacing: 0.5,
  tone: true,
  gloss: [èbě _pumpkin_],
)
```

Code 10: Code to generate Figure 32

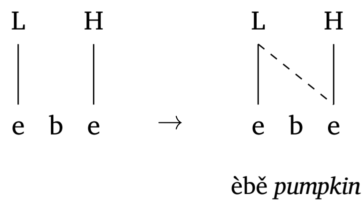


Figure 32: Tone spreading

Floating tones can be added with the `float` argument. Code 11 demonstrates how to add such a tone and how to draw a circle around it with the `highlight` argument, which can be used with any tone, of course.

```
#autoseg(
      ("i", "@", "N", "k", "a"),
      features: ("H", "", "L", "", "H"),
      highlight: (2,),
      spacing: 1.0,
      tone: true,
      float: (2,),
    )
```

Code 11: Floating tone

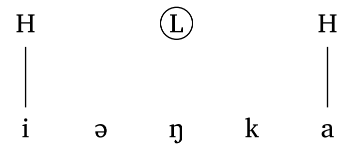


Figure 33: Floating tone

Spreading from a contour tone involves a more complex scenario. In Figure 34, the vowel in position 2 is linked to two tones. To create this representation, the element in position 2 inside `features` must be itself a tuple with the two tones in question. This will already add the solid lines connecting the vowel to the two tones. The `links` argument then takes care of the dashed line: `(((2, 0), 1),),`. This merely says "look at position 2 and pick the first element there, i.e., `(2,0)`, then draw a dashed line from that element to position 1". Tuples inside tuples are essential here.

```
#autoseg(
  ("m", "1", "A", "u"),
  features: ("", "", ("H", "L"), ""),
  links: (((2, 0), 1),),
  tone: true,
  highlight: ((2, 0),),
  spacing: 1.0,
  arrow: true,
),
```

Code 12: Contour tone

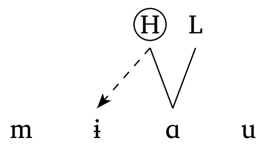


Figure 34: Contour tone

Finally, let's take a look at a three-step process (Zsiga, 2024). In this example, we have three instances of `#autoseg()` (one for each stage in the process). Here, we have delinking and segments that share a tone, which require some branching. Delinking is an essential process to represent, and that's what the argument `delinks` does. If you examine Code 13, you will notice that the third function specifies `delinks: ((3, 3),)`. This merely states that the link created between position 3 and itself must be delinked. Both `links` and `delinks` work as arrays of tuples.

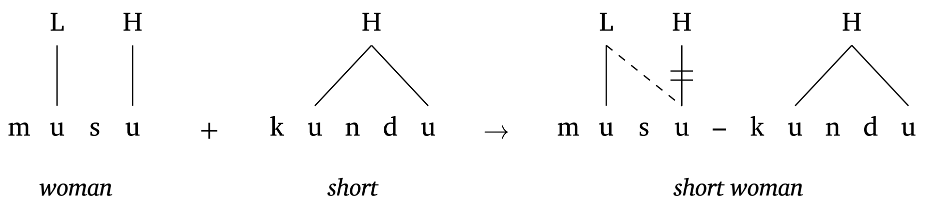


Figure 35: OCP effects in Vai (Zsiga, 2024)

To create a tone that is underlyingly linked to more than one segment, we use the `multilinks` argument. Unsurprisingly, this will require tuples inside tuples once again. For example, the result of the process in question is represented by the third `#autoseg()` in Code 13. There, we see `multilinks: ((6, (6, 9)),)`, which means "search for position 6 (this is the 'u' in 'kun'), then link that to positions 6 and 9 *at the same time*". The function automatically centers the tone between the two segments in question (it will center the tone regardless of how many segments it is linked to), and it also "erases" the vertical line inserted by the `features` argument.

```
#autoseg(
    ("m", "u", "s", "u"),
    features: ("", "L", "", "H"),
    tone: true,
    spacing: 0.5, // keep consistent
    baseline: 37%,
    gloss: [_woman_],
  ) +
  #autoseg(
    ("k", "u", "n", "d", "u"),
    features: ("", "H", "", "", ""), // H at position 1, but will be repositioned
    tone: true,
    float: (1,), // Mark H as floating so it doesn't draw vertical stem
    multilinks: ((1, (1, 4)),), // H links to segments at positions 1 and 4
    spacing: 0.5,
    baseline: 37%,
    arrow: false,
    gloss: [_short_],
  ) #a-r
  #autoseg(
    ("m", "u", "s", "u", "-", "k", "u", "n", "d", "u"),
    features: ("", "L", "", "H", "", "", "H", "", "", ""),
    links: ((1, 3),), // Link between L and H
    delinks: ((3, 3),), // Delink line between position 3 and itself
    arrow: false,
    multilinks: ((6, (6, 9)),), // Automatically removes vertical line in position 6
    tone: true,
    baseline: 37%, // Adjust as needed to customize vertical position
    spacing: 0.50,
    gloss: [_short woman_],
  )
```

Code 13: Code to generate Figure 35

Autosegments are difficult to typeset because there are many degrees of freedom involved, which means functions start to become too convoluted for the benefit they provide (see Section 4.7). There is probably a sweet spot, a point beyond which a function of this type becomes impractical because there are simply too many arguments. `#autoseg()` attempts to be intuitive while covering spreading, delinking, one-to-many and many-to-one relationships, floating tones, and highlights with circles. I believe these cover the vast majority of scenarios with precision and symmetry. The function also provides convenient arguments for horizontal (`spacing`) and vertical (`baseline`) positioning, which can be handy in processes where you may want to adjust the position of the representation relative to an arrow or any other symbols you may want to add outside the function *per se*.

Finally, the function does not and will not cover intonation, as its sole goal is to represent autosegmental phonology. In the future, an additional function might be added to cover intonational patterns and a wider range of tone-related representations.

4.7. Multi-tier representations

Thus far, the functions we've seen are single-purpose: `#syllable()` only creates syllables. The advantage of such an approach is that the function can offer minimal syntax, which makes it easy to use and very user-friendly. The disadvantage is that the user can only produce one type of output. Certain phonological structures, however, have too many degrees of freedom for a single-purpose function to be enough. This is why the function `#multi-tier()` exists: it gives you the freedom to create a wide range of non-linear structures. It is much more flexible, but that comes with more complicated syntax and more arguments, by definition.

Let us look at three use cases. First, let's reproduce a figure from Booij (2012, p. 159) on the interface of phonology and morphology. This is a good scenario to test the function because the figure requires multiple tiers. The figure is shown in Figure 36, and the code needed is shown next to Figure 37, which also displays the grid to help you see the position of each element. The main argument is `levels`, which should be relatively familiar given the discussion on `#autoseg()` earlier. Think of the function as a big grid where you position elements relative to rows (i.e., levels) and columns.

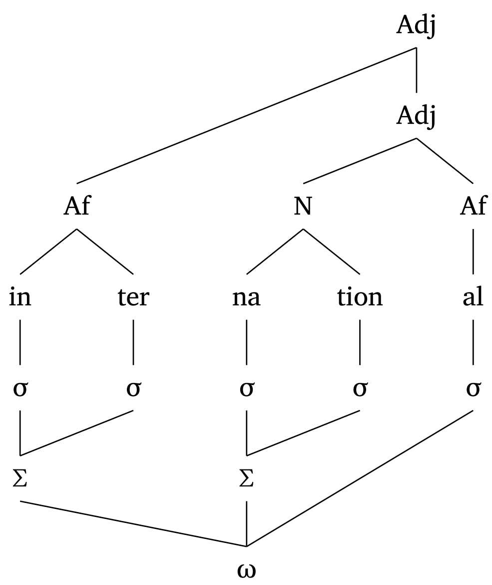


Figure 36: Morphology and phonology

```
#multi-tier(
  show-grid: true, // ← To help you see the grid
  levels: (
    ("", "", "", "", ("Adj", 3.5)),
    ("", "", "", "", ("Adj", 3.5)),
    ("", ("Af", 0.5), "", ("N", 2.5), "Af"),
    ("in", "ter", "na", "tion", "al"),
    ("sigma", "sigma", "sigma", "sigma", "sigma"),
    ("Sigma", "", "Sigma", "", ""),
    ("", "", "omega", "", ""),
  ),
  links: (
    ((0, 4), (2, 1)), // Adj → Af
    ((1, 4), (2, 3)), // Adj → N
    ((2, 1), (3, 0)), // Af → in
    ((2, 3), (3, 2)), // N → na
    ((5, 0), (4, 1)), // Ft → Syl
    ((5, 2), (4, 3)), // Ft → Syl
    ((6, 2), (5, 0)), // PWd → Ft
    ((6, 2), (4, 4)), // PWd → Ft
  ),
),
```

Code 14: Code to create Figure 37

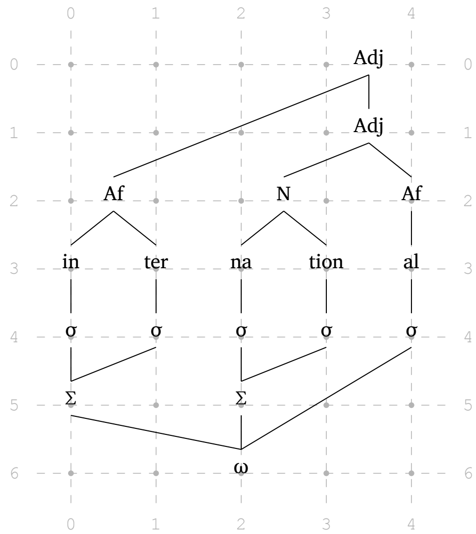


Figure 37: Morphology and phonology

You can think of `#multi-tier()` as a more powerful version of `#autoseg()`, and you will recognize many common elements between the two. For example, `float` is again an argument here, which allows you to have elements in the representation that aren't linked to anything. This is important because `#multi-tier()` will automatically link an element (with a single link) to the element on the next level (row). In Figure 37, you can comment out the contents in the `links` argument to see what the figure looks like in its bare form. Likewise, `highlight` can also be used if you need to circle a particular element in the representation.

By default, elements are automatically placed on the grid, but you may want to specify a different position. In Figure 37, for example, four elements are placed at fractional positions on the grid (both "Adj" at column 3.5, "Af" at 0.5, and "N" at 2.5). This is made possible by defining the position for the element in question as a tuple. Therefore, "N" is specified as `("N", 2.5)`, which places "N" in between "na" and "tion", found in the level below. You can also add a third argument for the vertical position if you ever want to place an element in between levels (not shown here). The argument `levels` will also render numbers as subscripts automatically.

Finally, you may have noticed that the function automatically detects Greek letters, so if you add "sigma", it will render as σ.[2]

You can use `#multi-tier()` to create a wide range of representations that are not easily generated by single-purpose functions. Structures used in Government Phonology or CV Phonology, for instance, can be easily generated. Figure 38 is from Goad (2012, p. 355).

[2] **phono*kit*** also provides a list of the most common Greek letters used in phonology (`extras.typ`): `#sigma`, `#mu`, `#omega`, etc. These will always render without italics. See Section G.2.

```
#multi-tier(
  levels: (
    ("O", "R", "", "O", "R", "O", "R"),
    ("", "N1", "", "", "N2", "", "N3"),
    ("", "x", "x", "x", "x", "x", "x"),
    ("", "", "s", "t", "E", "m", ""),
  ),
  links: (
    ((0, 1), (2, 2)),
  ),
  ipa: (3,),
  arrows: (
    ((3, 3), (3, 2)),
    ((0, 4), (0, 1)),
  ),
  arrow-delinks: (
    (1,)
  ),
  spacing: 1,
)
```

Code 15: Code to create Figure 38

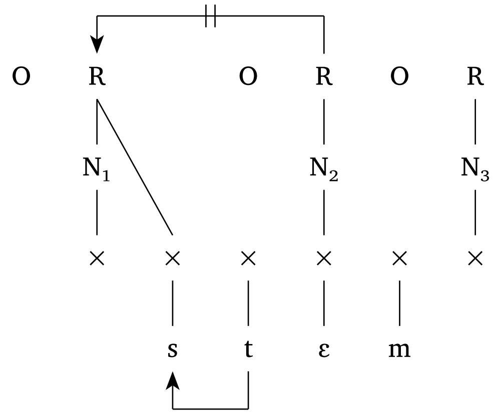


Figure 38: Government Phonology

As we examine Figure 38 and its code, some arguments are worth discussing in more detail. First, `ipa` allows you to specify which level contains IPA symbols that follow the conventions of **phono*kit*** (Section A). This makes it easy to add IPA symbols to the representation. Next, the argument `arrows` is responsible for the arrows you see in the figure. Arrows follow right-angle paths, and can be "cut off" with the `arrow-delinks` argument. Finally, `#multi-tier()`, like many other functions discussed here, also allows for sizing adjustments with arguments such as `scale`, `spacing`, `level-spacing`, `stroke-width` (which is used in Figure 39 below). As already mentioned, the flexibility offered by the function results in a much richer set of arguments, which in turn require a bit more time to get used to.

Finally, let's reproduce another figure, this time adapted from Carvalho (2017, p. 7). This representation also shows another argument, `tier-labels`, which is very practical if you ever want to label levels in your representation. Figure 39 also shows how to draw a dashed link between two elements, and how to `highlight` a particular element.

```
#multi-tier(
  levels: (
    ("T", "", "R", ""),
    ("O1", "n1", "O2", "n2"),
    ("x", "", ("x", 2.5), ""),
    ("o", "", ("N", 2.5), ""),
    ("", "", ("V", 2.5), ""),
  ),
  links: (
    ((1, 3), (2, 2)),
  ),
  dashed: (
    ((2, 0), (3, 2)),
  ),
  level-spacing: 1.2,
  highlight: (
    (1, 0),
  ),
  spacing: 1,
  stroke-width: 0.7pt,
  tier-labels: (
    (1, "C-plane"),
    (2, "skeleton"),
    (3, "V-plane"),
  ),
  scale: 1,
)
```

Code 16: Code to create Figure 39

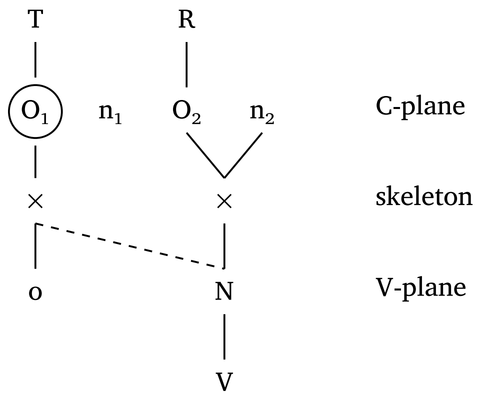


Figure 39: CV phonology

5. SPE

Rewrite rules can be very complex, and an excellent package already exists to deal with their complexity in Typst (Breit, 2025, `linphon`). The problem is that, like autosegmental representations, too many degrees of freedom exist in SPE-like representations, not to mention the variation across scholars when it comes to symbols, brackets, etc. On the plus side, you can do a lot simply by employing primitives that already exist (e.g., matrices and arrows), so SPE rules are not as challenging to typeset as some other non-linear structures in phonology (at least simpler rules…). For that reason, **phono*kit*** only has two primitive functions to help create feature matrices, which in turn can be combined to form SPE-style rules (Chomsky & Halle, 1968).

```
#feat-matrix("p") #feat-matrix("\\ae") #feat-matrix("\\t tS") #feat-matrix("i")
```

Code 17: Generating matrices for the phonemes in "patchy", shown in Figure 40

The first function is `#feat-matrix()`, shown in Code 17. It outputs the maximal feature matrix for a given phoneme (with the option for 0 values if `all: true`). This can be useful in introductory courses, where students are introduced to the notion of distinctive features. The function is not able to compute *minimal* matrices, i.e., it can't figure out that feature A entails feature B given inventory I,[3] but it can be used in sequence to represent matrices for a given word, for example — see Figure 40. The user is free to adjust the inventory of features and their values, since there's variation in the literature. The function is based on the features in Hayes (2009).

$$
\begin{array}{cccc}
/\text{p}/ & /\text{æ}/ & /\widehat{\text{tʃ}}/ & /\text{i}/ \\
\begin{bmatrix} +\text{consonantal} \\ -\text{sonorant} \\ -\text{continuant} \\ -\text{del rel} \\ -\text{approximant} \\ -\text{tap} \\ -\text{trill} \\ -\text{nasal} \\ -\text{voice} \\ -\text{spread gl} \\ -\text{constr gl} \\ +\text{labial} \\ -\text{round} \\ -\text{labiodental} \\ -\text{coronal} \\ -\text{lateral} \\ -\text{dorsal} \end{bmatrix}
&
\begin{bmatrix} +\text{syllabic} \\ -\text{consonantal} \\ +\text{sonorant} \\ +\text{continuant} \\ +\text{voice} \\ -\text{high} \\ +\text{low} \\ +\text{front} \\ -\text{back} \\ -\text{round} \end{bmatrix}
&
\begin{bmatrix} +\text{consonantal} \\ -\text{sonorant} \\ -\text{continuant} \\ +\text{del rel} \\ -\text{approximant} \\ -\text{tap} \\ -\text{trill} \\ -\text{nasal} \\ -\text{voice} \\ -\text{spread gl} \\ -\text{constr gl} \\ -\text{labial} \\ -\text{round} \\ -\text{labiodental} \\ +\text{coronal} \\ +\text{anterior} \\ +\text{distributed} \\ +\text{strident} \\ -\text{lateral} \\ -\text{dorsal} \end{bmatrix}
&
\begin{bmatrix} +\text{syllabic} \\ -\text{consonantal} \\ +\text{sonorant} \\ +\text{continuant} \\ +\text{voice} \\ +\text{high} \\ -\text{low} \\ +\text{tense} \\ +\text{front} \\ -\text{back} \\ -\text{round} \end{bmatrix}
\end{array}
$$

Figure 40: Matrices for the phonemes in the word "patchy"

Next, the function `#feat()` creates a matrix given a set of features. This is the function used in a rewrite rule, for example. The assimilation rule in Figure 41 is achieved with the code shown in Code 18 — notice that `alpha` notation requires a specific syntax, i.e., `X + "feat"` or `X + [#smallcaps("feat")]` if you prefer to use small caps. A helper function, `#blank()`, adds a long underline for the context of application in the rule. Likewise, `#a-r` adds an arrow using New Computer Modern (other arrows are available, such as `#a-l` ←, `#a-u` ↑, `#a-d` ↓, `#a-lr` ↔, `#a-sr` ⇝); see Section G.1.

```
#feat("+son", "-approx") #a-r
  #feat(sym.alpha + [#smallcaps("place")]) / #blank()\]#sub[#sym.sigma]
  #feat("-son", "-cont", "-del rel", sym.alpha + [#smallcaps("place")])
```

Code 18: Nasal place assimilation using `#feat()`

$$
\begin{bmatrix} +\text{son} \\ -\text{approx} \end{bmatrix} \rightarrow [\alpha\text{PLACE}] \; / \; \underline{\qquad}]_{\sigma} \begin{bmatrix} -\text{son} \\ -\text{cont} \\ -\text{del rel} \\ \alpha\text{PLACE} \end{bmatrix}
$$

Figure 41: A nasal place assimilation rule

[3] But see the Fonology package for R (Garcia, 2025).

6. Optimality theory

Unlike SPE rules, tableaux in optimality theory (OT; Prince & Smolensky (2004, originally circulated in 1993)) are more predictable and constrained. They're also one of the easiest representations to typeset, but if you use LaTeX they're still time-consuming to create (even though you can find tools online to streamline the task). **phono*kit*** includes two constraint-related functions, the first of which is `#tableau()`, shown in Tableau 1. The function takes six arguments: `input`, `candidates`, `constraints`, `violations`, `winner`, and `dashed-lines`. The accompanying code shown in Tableau 1 provides an example of how each argument works. For example, the `violations` argument requires a nested structure. Likewise, `dashed-lines` requires a comma if you want a given column to have a dashed line. If no dashed lines are needed, you can simply specify `dashed-lines: ()`. The `winner` candidate counts from zero, so that is the index/position for the first candidate, as shown in the example. These conventions are the same as those presented for autosegmental representations in Section 4.6.

```
#tableau(
  input: "kraTa",
  candidates: ("kra.Ta", "ka.Ta",
"ka.ra.Ta"),
  constraints: ("Max", "Dep", "*Complex"),
  violations: (
    ("", "", "*"),
    ("*!", "", ""),
    ("", "*!", ""),
  ),
  winner: 0, // ← Position of winning
cand
  dashed-lines: (1,) // ← Note the comma
  shade: true, // ← true by default
)
```

Code 19: Code to generate Tableau 1

/kraθa/	MAX	DEP	*COMPLEX
☞ [kra.θa]			*
[ka.θa]	*!		
[ka.ra.θa]		*!	

Tableau 1: A typical OT tableau

One nice feature of `#tableau()` is that the function automatically shades cells once a fatal violation is entered (`!`). Likewise, it adds the "☞" symbol for the winner, whose position is extracted from the `winner` argument shown in the accompanying code next to Tableau 1.

It is often useful to present a ranking using a Hasse diagram. These diagrams can be generated in **phono*kit*** using the `#hasse()` function. In a nutshell, the function takes tuples with n elements. In the simplest case, $n = 1$, which produces a floating constraint. Hasse diagram 1 shows a basic scenario. The third element in the first tuple indicates the "stratum" in the diagram — this is especially important in more complex cases, which require better control over the vertical position of different constraints. Optional arguments exist to give the user more flexibility (e.g., `scale` and `node-spacing`).

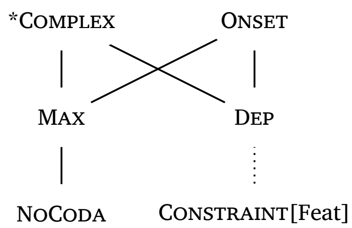
```
#hasse(
   (
     ("C1", "C2", 0),
     ("C3",), // floating constraint
   ),
   node-spacing: 2, // optional
   level-spacing: 2, // optional
   scale: 0.9, // optional
 )
```

Hasse diagram 1: Basic scenario

A more realistic scenario is shown in Hasse diagram 2. Notice that constraint names are automatically rendered in small caps (except features inside square brackets; see figure). 4-element tuples are also possible, in which case the fourth element allows for different line types between two constraints (`"dashed"` or `"dotted"`).

The top level of a diagram occupies position 0, which includes four partial rankings here. The second level (position 1) has two partial rankings in the diagram in question, namely Max $\gg$ NoCoda and Dep $\gg$ Constraint. You may want to play around with this position: change it from 1 to 2 and then to 3 to see how this affects the diagram — you may need to manually adjust the `node-spacing` accordingly to avoid constraint overlapping.

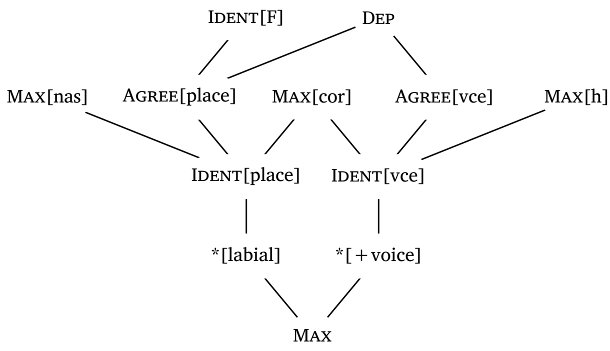
```
#hasse(
    (
      ("*Complex", "Max", 0),
      ("*Complex", "Dep", 0),
      ("Onset", "Max", 0),
      ("Onset", "Dep", 0),
      ("Max", "NoCoda", 1),
      ("Dep", "Constraint[Feat]", 1,
"dotted"),
    ),
    node-spacing: 3,
  )
```



Hasse diagram 2: Dotted lines and small caps

Finally, let's see a more complex scenario (notice that diagrams are automatically scaled according to their complexity). Hasse diagram 3 is adapted from Lamont (2021).

```
#hasse(
    (
      ("Ident[F]", "Agree[place]", 0),
      ("Dep", "Agree[vce]", 0),
      ("Dep", "Agree[place]", 0),
      ("Max[nas]", "Ident[place]", 1),
      ("Max[cor]", "Ident[place]", 1),
      ("Max[cor]", "Ident[vce]", 1),
      ("Max[h]", "Ident[vce]", 1),
      ("Agree[place]", "Ident[place]", 1),
      ("Agree[vce]", "Ident[vce]", 1),
      ("Ident[place]", "*[labial]", 2),
      ("Ident[vce]", "*[+voice]", 2),
      ("*[labial]", "Max", 3),
      ("*[+voice]", "Max", 3),
    ),
  )
```



Hasse diagram 3: Example from Lamont (2021)

7. Maximum Entropy grammars

Last but not least, the package goes one step further and produces a MaxEnt tableau (Goldwater & Johnson, 2003; Hayes & Wilson, 2008) with the function `#maxent()`. Tableau 1 illustrates a scenario where the data in Tableau 1 are variable, i.e., all candidates in question have a non-zero probability of being observed given a specific input x. The column h_i displays the harmony score of each candidate i, calculated as the weighted sum of all constraint violations. Next, the column e^{-h_i} provides the unnormalized probability, which is the exponential of the negated harmony score (this has also been called the *MaxEnt score*). Finally, the actual predicted probability is shown in column P_i, which is obtained by dividing the unnormalized value of a candidate by $Z(x)$ (the sum of all unnormalized scores). The formal equation for this probability is provided in Equation 1.

$$P(y|x) = \frac{e^{-\sum_{i=1}^{n} w_i C_i(y,x)}}{Z(x)} \tag{1}$$

The function `#maxent()` calculates h_i, e^{-h_i} and $P(y|x)$[4] (shown as P_i in the tableau) automatically given the weights provided. Tableau 1 lists the weights for the constraints in use at the top and prints probability bars at the right margin. These can be turned off with `visualize: false` (see Code 20), but they are printed by default as this can help students quickly visualize probabilities when many candidates are evaluated.

	$w = 2.5$	$w = 1.8$	$w = 1$			
/kraθa/	MAX	DEP	*COMPLEX	h_i	e^{-h_i}	P_i
[kra.θa]	0	0	1	1	0.368	0.598
[ka.ra.θu]	0	1	0	1.8	0.165	0.269
[ka.θa]	1	0	0	2.5	0.082	0.133

Tableau 1: A MaxEnt tableau with ordered by P_i with `sort: true`

[4]Where y is a given candidate and x is the input.

In Code 20, you can see all the necessary arguments for the function `#maxent()`. Like the function `#tableau()` discussed above, the `violations` argument in `#maxent()` requires a nested structure. Everything else is self-explanatory. As expected, the rows and columns will expand as needed for both constraint-based functions.

```
#maxent(
  input: "kraTa",
  candidates: ("[kra.Ta]", "[ka.Ta]", "[ka.ra.Ta]"),
  constraints: ("Max", "Dep", "*Complex"),
  weights: (2.5, 1.8, 0.5),
  violations: (
    (0, 0, 1),
    (1, 0, 0),
    (0, 1, 0),
  ),
  visualize: true, // Show probability bars (default)
  sort: true // Sort candidates from most to least probable
)
```

Code 20: Code to generate a MaxEnt tableau

phono*kit* also has functions for harmonic grammars (`#hg()`) and noisy harmonic grammars (`#nhg()`). These functions are very similar to `#maxent()`, so their syntax will be familiar. The Noisy Harmonic Grammar function derives probabilities by simulating a number of evaluations (by default, 1000) given the constraints and violations provided by the user. It is possible to change the number of simulations and to omit the noise column from the tableau. The noise displayed is extracted from an additional simulation, so it is shown for illustrative purposes. For the most part, the functions discussed in this section are based on conventions in the literature, e.g., Flemming (2021).

8. Numbered examples

One important feature in any phonology paper is an environment for numbered examples. We could simply use the excellent `eggs` package for Typst (Shaldov, 2025), which does a great job for syntax examples. In phonology, however, we sometimes want examples to have arrows to show a process, and this creates an alignment issue when we have multiple instances in a single example or in multiple examples throughout. **phono*kit*** has a function to deal with that: `#ex()`, which is accompanied by `#subex-label()`. The latter function allows us to add labels for each line inside an example.

> **Important.** To use this function, add this line after importing the package: `#show: ex-rules`

```
// #import "lib.typ": *
// #show: ex-rules // ⟵ this must be added to your doc
//  ...
      #ex(caption: "A phonology example")[
        #table(
          columns: 4, // ⟵ where we may specify widths
          stroke: none,
          align: left,
          [#subex-label()<ex-anba>], [#ipa("/anba/")],
[#a-r], [#ipa("[amba]")],
          [#subex-label()<ex-anka>], [#ipa("/anka/")],
[#a-r], [#ipa("[aNka]")],
        )
      ] <phon-ex>
```

(1)	a.	/anba/	→	[amba]
	b.	/anka/	→	[aŋka]

Code 21: Numbered example

We can now refer to the example as a whole using `@phon-ex`, just like any other label, as in (1) and (2). Crucially, we can easily refer to sub-examples using their labels, as in (1a) and (2b). Because `#ex()` is based on tables, it is easy to customize it with however many columns we need.

(2)	a.	/inbi/	→	[imbi]
	b.	/inki/	→	[iŋki]

Another advantage of building a function on top of tables is that we can establish spacing consistently across multiple examples by specifying the width of each column. If you look back at Code 21, you will notice that the argument `columns` specifies the number of columns needed for the example. Instead of using an integer, we can use an array with specific widths for each column. This is what is done for examples (2) and (3): `columns: (2em, 7em, 4em, 7em)`. This guarantees perfect alignment across different examples.

(3)	a.	/inbinbi/	→	[imbimbi]
	b.	/inkinki/	→	[iŋkiŋki]

You will notice that `caption` is one of the arguments in `#ex()`. This is not a typical caption (nothing is printed), since this is an example after all. Instead, this caption is there in the (unlikely) event we wish to create a table of contents *only for examples* using the `#outline()` function.[5] The outline below shows what the examples would look like in a table of contents — see Code 22.

List of Examples

[5]While it is not very common to have such tables, `#ex()` gives you the possibility.

```
#outline(
  title: [List of Examples],
  target: figure.where(kind: "linguistic-example"),
)
```

Code 22: Creating a table of examples using `#outline()`

A note on figure spacing. If your document uses a custom `#show figure` rule to add vertical spacing around figures, you may notice misalignment in sub-example labels. This happens because `#subex-label()` uses figures internally, and added spacing will disrupt table cell alignment. To fix this, exclude `linguistic-subexample` from your spacing rule. The present document, for example, does have a rule to add `1em` vertical spacing before and after each figure. To protect `#ex()` from that custom spacing, the preamble should have something along the lines of Code 23.

```
#show figure: it => {
  if it.kind == "linguistic-subexample" {
    it
  } else {
    v(1em)
    it
    v(1em)
  }
}
```

Code 23: Excluding linguistic examples from custom figure spacing specifications

9. Final remarks

phono*kit* is a *very* young project (December 2025). As stated above, its main goal is to quickly generate structures that are frequently used by phonologists when typesetting documents for teaching and research. This means that functions must be **intuitive** — but an intuitive interface cannot sacrifice typographical quality. I hope to have shown that this goal is possible. The package will certainly improve as people start using it in a wider range of scenarios than those presented in this vignette, which represent for the most part the contexts I need myself when preparing documents.

Typst is still in its infancy, and I believe most linguists do not know about it yet (as of January 2026). But as the language expands into linguistics, there is *a lot* of potential for significant advances in our workflows (both in research and teaching). I hope this package will make document preparation quicker and more enjoyable to the phonologists out there.

References

Booij, G. (2012). *The grammar of words: An introduction to linguistic morphology* (3rd ed.). Oxford University Press.

Breit, F. (2025). *linphon: Typesetting phonological rules and feature matrices in Typst*. https://github.com/thatfloflo/typst-linphon

Carvalho, J. B. d. (2017). Deriving sonority from the structure, not the other way round: A Strict CV approach to consonant clusters. *The Linguistic Review*, *34*(4), 589–614.

Chomsky, N., & Halle, M. (1968). *The sound pattern of English*. Harper & Row.

Dreamin, M., & Varner, N. (2024). *Tinymist: An integrated language service for Typst*. https://github.com/Myriad-Dreamin/tinymist

Fejfar, J., & Funke, M. (2024). *CeTZ: A library for drawing with Typst*. https://github.com/cetz-package/cetz

Flemming, E. (2021). Comparing MaxEnt and noisy harmonic grammar. *Glossa: A Journal of General Linguistics*, *6*(1), 1–42. https://doi.org/10.16995/glossa.5775

Garcia, G. D. (2025). *Fonology: Phonological Analysis in R*. https://gdgarcia.ca/fonology

Garcia, G. D., & Guzzo, N. B. (2023). A corpus-based approach to map target vowel asymmetry in Brazilian Veneto metaphony. *Italian Journal of Linguistics*, *35*(1), 115–138. https://doi.org/10.26346/1120-2726-205

Goad, H. (2012). sC clusters are (almost always) coda-initial. *Linguistic Review*, *29*(3).

Goldwater, S., & Johnson, M. (2003). Learning OT constraint rankings using a Maximum Entropy model. *Proceedings of the Stockholm Workshop on Variation within Optimality Theory*, 111–120.

Hayes, B. (2009). *Introductory phonology*. John Wiley & Sons.

Hayes, B., & Wilson, C. (2008). A Maximum Entropy Model of Phonotactics and Phonotactic Learning. *Linguistic Inquiry*, *39*(3), 379–440. https://doi.org/10.1162/ling.2008.39.3.379

Lamont, A. (2021). Optimality Theory implements complex functions with simple constraints. *Phonology*, *38*(4), 729–740. https://doi.org/10.1017/s0952675721000361

Parker, S. (2011). Sonority. In M. van Oostendorp, C. J. Ewen, E. Hume, & K. Rice (Eds.), *The Blackwell Companion to Phonology: The Blackwell Companion to Phonology* (pp. 1160–1184). Wiley Online Library. https://doi.org/10.1002/9781444335262.wbctp0049

Prince, A., & Smolensky, P. (2004). *Optimality Theory: constraint interaction in Generative Grammar*. Blackwell.

Rei, F. (1996,). *TIPA: A system for processing phonetic symbols in LATEX*. TUGBoat.

Shaldov, A. (2025). *eggs: linguistic examples with minimalist syntax*. https://github.com/retroflexivity/typst-eggs

SIL International. (2025). *Charis SIL*. https://software.sil.org/charis/

Zsiga, E. C. (2024). *The sounds of language: An introduction to phonetics and phonology* (2nd ed.). John Wiley & Sons.

Appendix

A. IPA symbol reference

Plosives															
p	p	b	b	t	t	d	d	\\:t	ʈ	\\:d	ɖ	c	c	\\barredj	ɟ
k	k	g	ɡ	q	q	\\;G	ɢ	?	ʔ	\\barredP	ʡ				
Fricatives															
F	ɸ	B	β	f	f	v	v	T	θ	D	ð	s	s	z	z
S	ʃ	Z	ʒ	\\:s	ʂ	\\:z	ʐ	C	ç	J	ʝ	x	x	G	ɣ
X	χ	K	ʁ	\\barredh	ħ	Q	ʕ	h	h	H	ɦ	\\;H	ʜ	\\barrevglotstop	ʢ
\\textbeltl	ɬ	\\l3	ɮ	*w	ʍ	\\texththeng	ɧ								
Nasals															
m	m	M	ɱ	n	n	\\:n	ɳ	\\nh	ɲ	N	ŋ	\\;N	ɴ		
Approximants & trills															
V	ʋ	*r	ɹ	j	j	\\darkl	ɫ	l	l	L	ʎ	\\:l	ɭ	\\;L	ʟ
r	r	R	ɾ	\\:r	ɽ	\\;R	ʀ	\\mw	ɰ	\\;B	ʙ	\\:R	ɻ	\\turnlonglegr	ɺ
Clicks & implosives															
\\!o	ʘ	\\doublebarpipe	ǂ	||	ǁ	\\!b	ɓ	\\!d	ɗ	\\!j	ʄ	\\!g	ɠ	\\!G	ʛ
\\textctz	ʦ														
Vowels															
i	i	I	ɪ	y	y	Y	ʏ	1	ɨ	0	ʉ	W	ɯ	u	u
U	ʊ	e	e	\\o	ø	9	ɘ	8	ɵ	7	ɤ	o	o	@	ə
E	ɛ	\\oe	œ	3	ɜ	2	ʌ	O	ɔ	\\ae	æ	\\OE	ɶ	a	a
5	ɐ	A	ɑ	6	ɒ	\\schwar	ɚ	\\epsilonr	ɝ	\\closeepsilon	ɞ				
Diacritics, suprasegmentals, archiphonemes															
'ta	ˈta	,ta	ˌta	u:	uː	\\~ a	ã	\\r i	i̥	\\v n	n̩	t \\h	tʰ	p *	p̚
\\t ts	t͡s	k \\labial	kʷ	p \\velar	pˠ	t \\palatal	tʲ	\\dental t	t̪	\\C	C	\\V	V	\\N	N

Table 1: IPA Reference Guide

B. Symbols in prosodic representations

As of version `0.3.7`, users can decide which symbols are used for prosodic words, feet, syllables and moras. This is accomplished by the argument `symbol`, which requires an array. For consistency, the default symbols are Greek letters: ω, Σ, σ, μ.

For **prosodic words**, the `symbol` array has 3 positions (or 4 for prosodic words using moras). The user can replace the numbers in the code with the symbols/strings of their choice.

```
#word("@.( \\t dZ En).d@",
  symbol: ("1", "2", "3"))
```

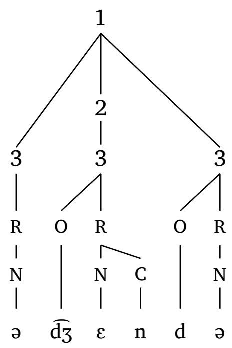


Figure 42: Array for prosodic symbols

For feet, two positions are available (foot and syllable).

```
#foot("\\t dZ En",
  symbol: ("1", "2"))
```

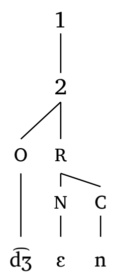


Figure 43: Array for prosodic symbols

For feet with moras, three positions can be changed should the user prefer to alter the default symbol for moras.

```
#foot-mora("\\t dZ En",
  symbol: ("1", "2", "3"))
```

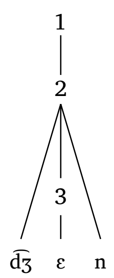


Figure 44: Array for prosodic symbols

Finally, syllables can also be changed.

```
#syllable("\\t dZ En",
  symbol: ("1",))
```

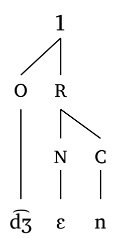


Figure 45: Array for prosodic symbols

C. How can I use Typst offline?

This vignette assumes that you know about Typst, but you may not be very familiar with it. That's why this section exists. For example, while the online app at Typst.app is very useful and practical, most of us prefer to work offline. **How can you use Typst offline then?**

One of the best IDE options out there is to use VS Code with the extension Tinymist (Dreamin & Varner, 2024) — the extension is therefore available for Positron, which is the successor to RStudio. Tinymist is also available as a plugin for NeoVim users. All these options work extremely well because Tinymist is great, and I haven't had any issues thus far: compilation is instantaneous, and `bib` files also work flawlessly.[6]

D. How do packages work in Typst?

If you've used R, Python, LaTeX, etc., you are used to installing packages and then importing them. This vignette has imported **phono*kit***, of course, which in turn imports CeTZ (Fejfar & Funke, 2024) as a dependency. As you start using Typst, you will notice that it works a bit differently, and this may not be self-evident at first. As seen in Section 1.1, there are basically two ways to load and use a package, both of which require the function `#import` inside your `typ` document — notice that you don't install a package *per se*. The traditional way is to import a package from the official Typst collection/repository, which means adding `#import "@preview/phonokit:0.3.7": *` to your `typ` document if you plan on using **phono*kit*** (assuming version `0.3.7`). The `@preview` bit indicates that the package comes from Typst's official repository. This is what you should do most of the time. Typst packages are cached once you compile a document with a given package.

Another option is to fork, clone or download a package from GitHub and import its `lib.typ` file instead: `#import "PACKAGE_DIRECTORY/lib.typ": *`. There's only one caveat: Typst restricts imports to files within the compilation root and its subdirectories (i.e., you can't load `lib.typ` if the package is in a parent directory or elsewhere in your system). Thus, you may need to use symlinks (this is the same strategy applied to `bib` files if you don't want to have a local copy of your references).

```
# From your working directory:
ln -s PATH_TO_PACKAGE_DIRECTORY package_name
```

Code 24: Creating a symlink to local package

Finally, Typst's repository contains sub-directories to keep track of each version of a given package. When a package is updated, nothing happens to the existing version of the package. Instead, a new directory is added with the updated version. That's why you specify the *version* of a package upon importing: `#import "@preview/phonokit:0.4.0": *`. If you go to Typst's repository on GitHub, you will see that the repository for **phono*kit*** has one sub-directory for each version of the package. This means that you always know which version of a package you are using. And because previous versions are not removed, backwards compatibility is not an issue. If you are used to LaTeX and you have the habit of frequently updating your packages, you will appreciate this, as there's no risk of recompiling your document and running into errors because one of the packages you use has been updated with breaking changes.

[6] Provided that they are clean and do not have any problems regarding fields, repeated entries, etc.

E. Exporting representations as images

You may want to use **phono*kit*** without necessarily adopting Typst. The easiest way to do this is go to to Typst.app to create the structure you need using the page set-up shown in Code 25. You can then download the output and voilà. You can later add to your LaTeX or Word document — this is similar to using LaTeXiT.

Fortunately, it is also easy to automate this process if you want to do it off-line. First, make sure you install Typst compiler here. Then, create a Typst file with your desired representation. One example is provided in Code 25, where we replicate Tableau 1. In the preamble of the file, notice that we load **phono*kit*** and then adjust our page settings. You will notice that `fill` is set to `none`, which ensures that our resulting PNG file has transparent background. Finally, both `height` and `width` are set to `auto`, which sets the page size dynamically according to the size of the representation you wish to create. We will call this file `maxent.typ`.

```
// Create a file called maxent.typ:
#import "@preview/phonokit:0.4.0": *
#set page(width: auto, height: auto, margin: 0.5em, fill: none)
#maxent(
  input: "kraTa",
  candidates: ("[kra.Ta]", "[ka.Ta]", "[ka.ra.Ta]"),
  constraints: ("Max", "Dep", "*Complex"),
  weights: (2.5, 1.8, 1),
  violations: (
    (0, 0, 1),
    (1, 0, 0),
    (0, 1, 0),
  ),
  visualize: true, // Show probability bars (default)
)
```

Code 25: Example `typ` file to generate a MaxEnt tableau (Tableau 1)

After having created the file in question, simply run the command shown in Code 26 from Terminal. This will generate a PNG file called `maxent.png` with 500 pixels per inch.

```
typst compile --ppi 500 maxent.typ maxent.png
```

Code 26: Compile to PNG in terminal

You could go one step further and use a convenient bash script to take a **phono*kit*** function and generate a PNG figure for any given representation. If you're not familiar with bash scripting, you can use any AI agent to create such a script. An example is provided in Code 27. The script allows you to run the function `phonokit()` from your terminal. Inside the function, you can use any **phono*kit*** function. For example, if you wanted to create a PNG figure for the syllable "ast", you would simply run `phonokit(syllable("ast"))`. The script also gives you the option to set your resolution. By default, the output is saved with a time stamp to your desktop (I'm using Mac OS for reference).

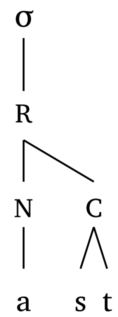


Figure 46: Syllable "ast" generated as a PNG file

```
# NOTE: Function to generate PNGs figures using Phonokit
# Adjust Phonokit version and destination as needed
phonokit() {
  local code="$1"
  local ppi="${2:-500}"
  local timestamp=$(date +%Y%m%d_%H%M%S)
  local output="$HOME/Desktop/phonokit_output_${timestamp}.png"
  local tmp=$(mktemp /tmp/phonokit-XXXXXX.typ)

  cat > "$tmp" << EOF
#import "@preview/phonokit:0.4.0": *
#set page(width: auto, height: auto, margin: 0.5em, fill: none)
$code
EOF

  typst compile --ppi "$ppi" "$tmp" "$output"
  rm "$tmp"

  echo "Saved to $output"
}
```

Code 27: Bash script to export PNGs

F. Useful NeoVim keymaps

If you use NeoVim, you may want to set some keymaps for Typst files. Code 28 shows a simplified version of the keymaps I use along with two key plugins: Tinymist (`neovim/nvim-lspconfig`), already mentioned, and `typst-preview` (`chomosuke/typst-preview.nvim`), which allows you to preview `typ` files live in the browser. In a nutshell, these commands could be useful: `\p` to preview a `typ` file on the browser and `\c` to compile the document to a `PDF`. The other two commands listed in Code 28 are also useful: `\i` generates a `PNG` file from the `typ` document (see Section E), and `\s` stops the preview.

```lua
vim.api.nvim_create_autocmd("FileType", {
  pattern = "typst",
  callback = function()
    vim.keymap.set("n", "<localleader>i", function()
      local file = vim.fn.expand("%")
      local png_name = vim.fn.expand("%:r") .. ".png"
      vim.fn.system("typst compile --format png --ppi 500 " .. vim.fn.shellescape(file))
      if vim.v.shell_error == 0 then
        print("✓ Exported: " .. png_name)
      else
        print("✗ PNG export failed!")
      end
    end, { desc = "Export Typst to PNG (500 ppi)", buffer = true })
    vim.keymap.set("n", "<localleader>p", "<cmd>TypstPreview<cr>", { desc = "Typst
Preview", buffer = true })
    vim.keymap.set("n", "<localleader>s", "<cmd>TypstPreviewStop<cr>", { desc = "Typst
Preview Stop", buffer = true })
    vim.keymap.set("n", "<localleader>c", function()
      local file = vim.fn.expand("%")
      local pdf_name = vim.fn.expand("%:r") .. ".pdf"

      vim.fn.system("typst compile " .. vim.fn.shellescape(file))

      if vim.v.shell_error == 0 then
        print("✓ Compiled: " .. pdf_name)
      else
        print("✗ Compilation failed!")
      end
    end, { desc = "Export Typst to PDF", buffer = true })
  end,
})
```

Code 28: Useful NeoVim keymaps

G. Extras

The `extras.typ` module provides convenience symbols and helper functions.

G.1. Arrows

Function	Symbol	Description
`#a-r`	→	Right arrow
`#a-l`	←	Left arrow
`#a-u`	↑	Up arrow
`#a-d`	↓	Down arrow
`#a-lr`	↔	Bidirectional arrow
`#a-ud`	↕	Vertical bidirectional arrow

`#a-sr`	⇝	Squiggly right arrow
`#a-sl`	⇜	Squiggly left arrow
`#a-r-large`	⟶	Large right arrow with spacing

G.2. Greek symbols

These render upright (non-italicized), unlike math-mode `$sigma$`.

Function	Sym.	Function	Sym.	Function	Sym.
`#alpha`	α	`#mu`	μ	`#tau`	τ
`#beta`	β	`#phi`	φ	`#omega`	ω
`#gamma`	γ	`#pi`	π	`#cap-phi`	Φ
`#delta`	δ	`#sigma`	σ	`#cap-sigma`	Σ
`#lambda`	λ			`#cap-omega`	Ω

G.3. Utilities

Function	Description
`#blank()`	Underline blank for fill-in exercises or SPE rules. Width is adjustable: `#blank(width: 4em)`
`#extra[ ... ]`	Wraps content in ⟨angle brackets⟩ for extrametricality: `#extra[tion]` → ⟨tion⟩

H. More information, questions, suggestions

If you have any questions, visit `github.com/guilhermegarcia/phonokit`, where you will find all the code for the package. If you find a bug or typo, or if you'd like to suggest a feature, please open an issue in the repository — this will help improve the package. This is an ongoing project that started in December 2025, so there is *a lot* to be improved.