Phonokit: a Typst package for phonology

Guilherme D. Garcia • UNIVERSITÉ LAVAL

DOI 10.5281/zenodo.17971032

Last updated on December 20, 2025

1. Introduction

I have used LaTeX (Knuth, 1984; Lamport, 1994) for over a decade, and it has been a central part of my workflow: slides, articles, posters, handouts, letters, statements, books. If you have custom-made templates and macros (especially when paired with snippets), it is difficult to beat the quality and efficiency of LaTeX, especially if you are attentive to detail and enjoy typography overall. However, for all its pros, LaTeX certainly has its cons.

LaTeX has a steep learning curve, especially for students who have zero experience with coding — it doesn't help that its error messages are often cryptic. LaTeX also has slow compilation times. Typst is a new language that addresses all these shortcomings: it compiles nearly instantaneously, it is more intuitive and user-friendly, and it's *light*. On top of that, it is a proper scripting language, which means complex tasks that require dozens of lines of LaTeX can frequently be accomplished in just a few lines of Typst.

The transition from LaTeX to Typst within my workflow is the reason why Phonokit exists. As I went down my list of must-haves for my typesetting needs, I started to create different functions and templates. The package is thus a collection of those functions. Its goal is very simple: to simplify and speed up all the essential typesetting elements a phonologist needs when producing documents, including phonetic transcription, tables of phonemic inventories, prosodic representations, rewrite rules, and optimality-theoretical tableaux. These represent about 90% of everything I need when preparing class notes and slides, or when writing up papers.

This vignette introduces Phonokit and its functions. Comments and suggestions are welcome, as are bug reports. The GitHub repository for the package can be found at `guilhermegarcia/phonokit`. The main goals of Phonokit are to MINIMIZE EFFORT and MAXIMIZE QUALITY.

2. IPA module

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

2.1. Phonemic transcription

As can be seen in Code example 1, 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. Note that all functions in Phonokit require the Charis SIL font (SIL International, 2025), regardless of the font used in the document.

`#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")`	pʰɪkˈ t͡ʃæt pl̥iz	*Pick, chat, please*
`#ipa("'lIt \\v l \\s 'b2R \\schwar , flaI")`	ˈlɪtl̩ ˈbʌɾɚˌflaɪ	*Little butterfly*

Code example 1: Transcriptions using `#ipa()`

2.2. Phonemic inventories

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 (Brazilian) Portuguese, 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 /ɯ/ is not present, in which case /w/ appears only under bilabial).

Aspirated consonants are shown when `aspirated: true`, which allows for aspirated affricates in Mandarin to be displayed, for example (Figure 2). When neither `affricates` nor `aspirated` are set to `true`, the function will omit both groups (e.g., Figure 1) and fewer rows will be printed.

	Bilabial	Labiodental	Dental	Alveolar	Postalveolar	Retroflex	Palatal	Velar	Uvular	Pharyngeal	Glottal
Plosive	p b			t d				k g			
Nasal	m			n			ɲ				
Trill											
Tap or Flap				ɾ							
Fricative		f v		s z	ʃ ʒ			x			
Lateral fricative											
Approximant	w						j	w			
Lateral approximant				l			ʎ				

Figure 1: `#consonants("portuguese")`

The user can either input a language (see caption of Figure 1) or a string of consonants to create a custom inventory — the input follows the same format used by the `#ipa()` function discussed in Section 2.1, so `#ipa("\\* r")` generates "ɹ". Finally, the function also allows for flexible sizing with the `scale` argument, shown in Figure 2.

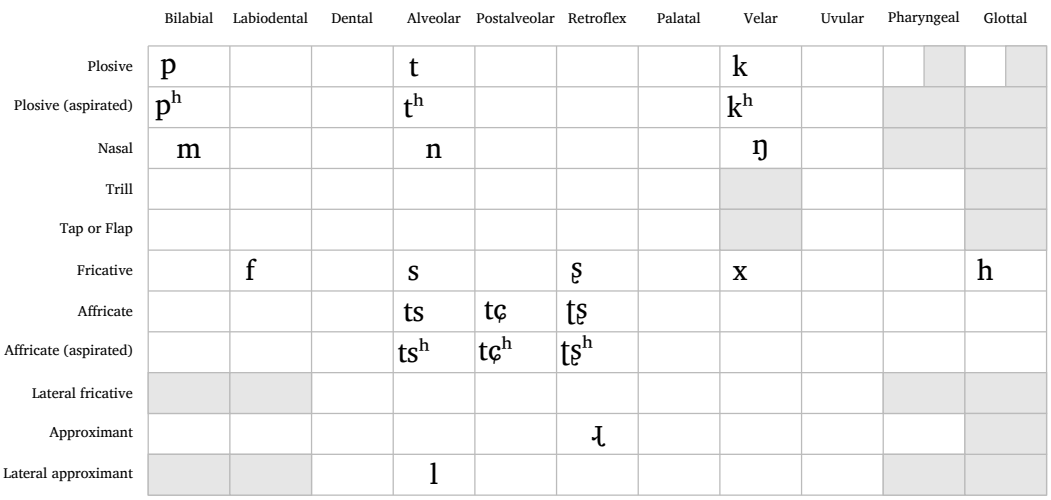

Figure 2: `#consonants("mandarin", aspirated: true, affricates: true, scale: 0.6)`

Besides the function `#consonants()`, the package also 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.

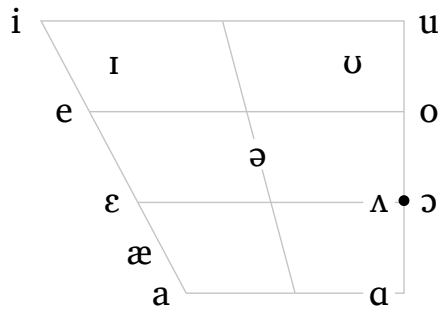

Figure 3: `#vowels("english", scale: 0.6)`

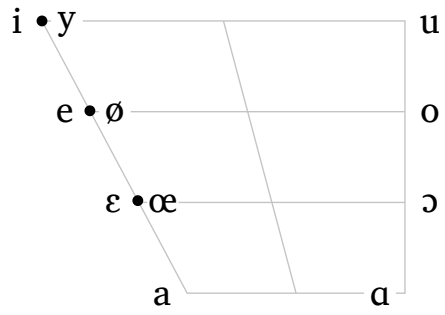

Figure 4: `#vowels("french", scale: 0.6)`

3. Prosody module

3.1. Sonority

When discussing the sonority principle in introductory phonology courses, it is often useful to illustrate relative sonority with a visual representation. The function `#plot-son()`, 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), but the user is free to adjust the scale as needed in the source code.

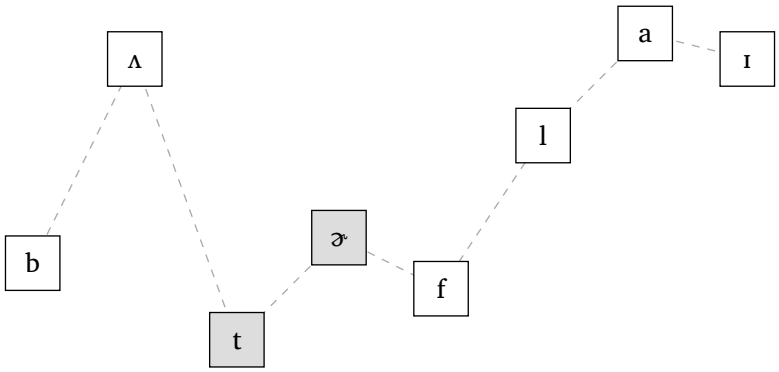

Figure 5: `#plot-son("b2.t \\schwar . flaI")`

Figure 5 shows an example for the word “butterfly”. 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.

3.2. Syllables

We start with an essential representation, namely the syllable. Two options are available: `#syllable()` for a classic onset-rhyme representation (Figure 6), and `#mora(.., coda: true)` for a moraic representation (Figure 7). 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 2.1.

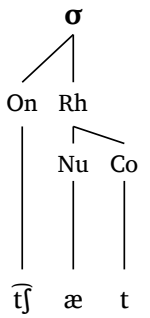

Figure 6: `#syllable("\\t tS \\ae t")`

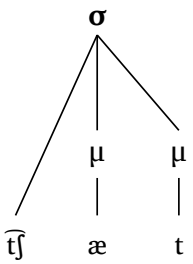

Figure 7: `#mora("\\t tS \\ae t", coda: true)`

Vowel length is also represented in both `#syllable()` and `#mora()`, as can be seen in Figure 8 and Figure 9, respectively. The crucial element here is the use of `:`, which triggers the : symbol for both representations. In the moraic representation, two moras branch out of the vowel, as expected.

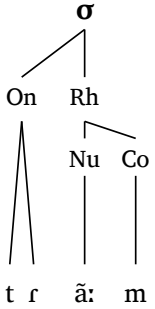

Figure 8: `#syllable("tR \\~ a:m")`

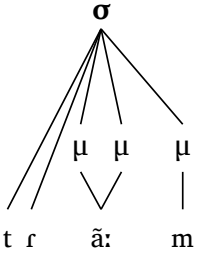

Figure 9: `#mora("tR \\~ a:m", coda: true)`

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 10 illustrates this with an extreme example.

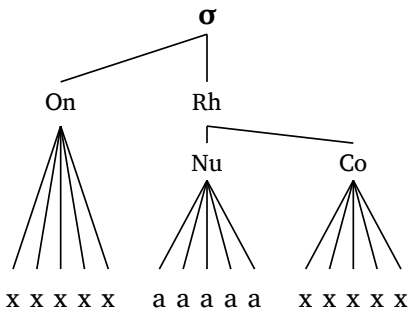

Figure 10: How spacing is managed

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 11, Figure 12 and Figure 13.

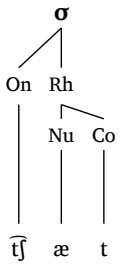

Figure 11: Scale 0.75

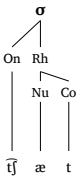

Figure 12: Scale 0.5

Figure 13: Scale 0.25

3.3. Feet

Next, we examine metrical feet (Figure 14 and Figure 15). 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 3.4). A period `.` is used 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 16 and Figure 17).

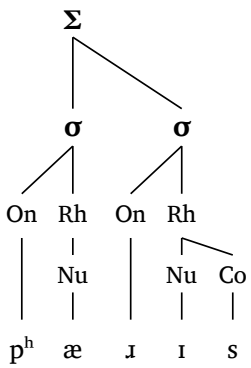

Figure 14: `#foot("'p \\h \\ae.\\*r Is")`

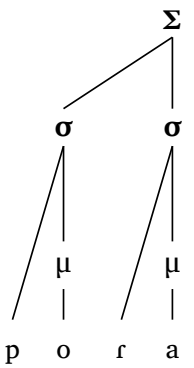

Figure 15: `#foot-mora("po.'Ra", coda: true)`

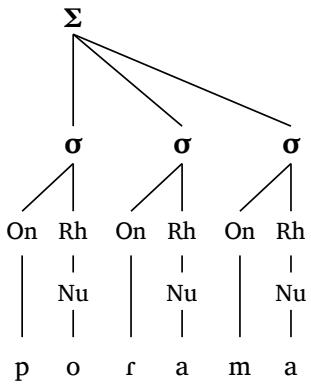

Figure 16: `#foot("'po.Ra.ma")`

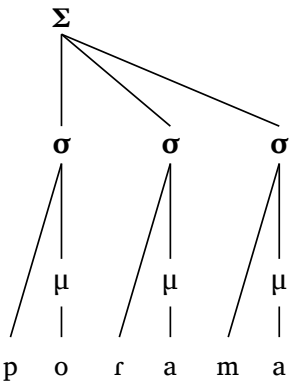

Figure 17: `#foot-mora("'po.Ra.ma")`

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 18 and Figure 19 show a disyllabic word containing a geminate in both onset-rhyme and moraic representations. The two figures alternate the stress position to illustrate right- and left-headed feet — it goes without saying that the function does not evaluate the plausibility of a metrical representation.

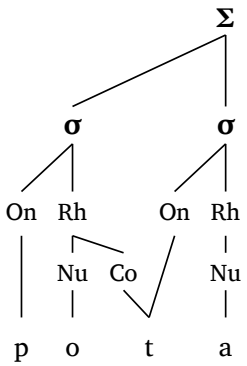

Figure 18: `#foot("pot.'ta")`

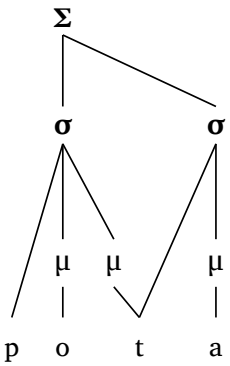

Figure 19: `#foot-mora("'pot.ta", coda: true)`

Extreme cases are important to test how adaptable the function is when it comes to line crossings, a key problem in prosodic representations. When a head domain is at an edge of a long string, it is challenging to avoid crossings. As can be seen in Figure 20, the height of Σ is proportional to the width of the representation to avoid superposition of lines.

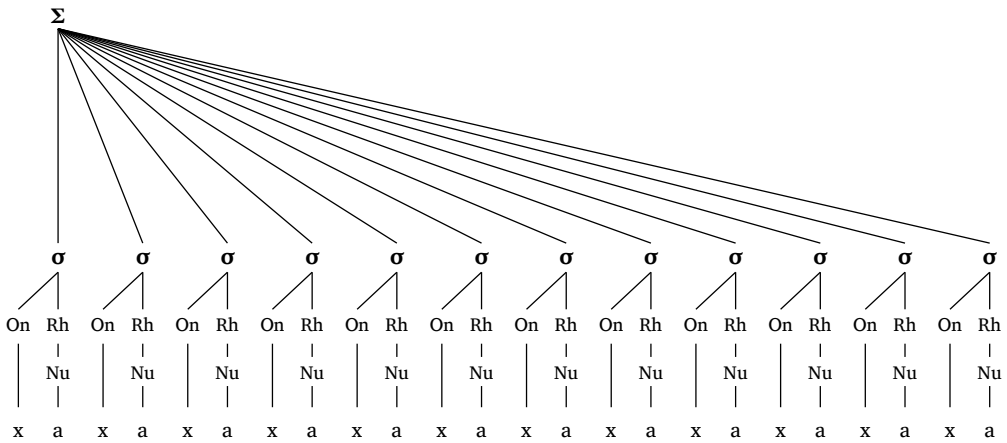

Figure 20: An extreme case

3.4. 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).

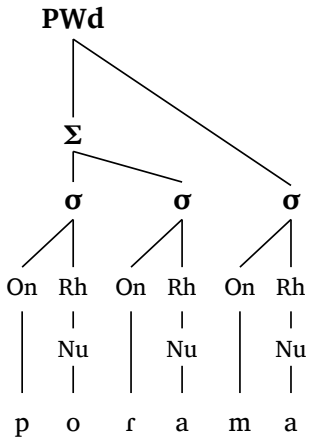

Figure 21: `#foot("('po.Ra).ma")`

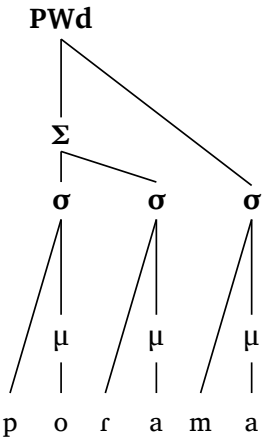

Figure 22: `#foot-mora("('po.Ra).ma")`

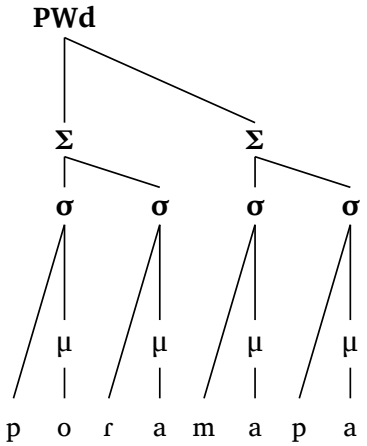

Figure 23: When `foot: "L"` (default)

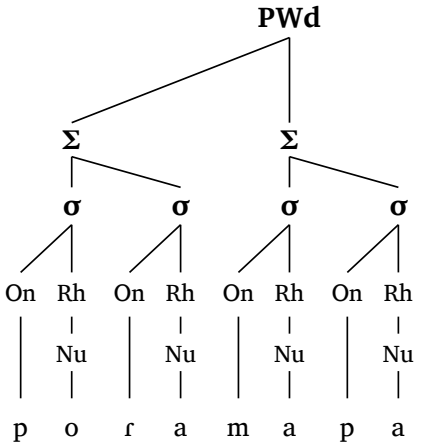

Figure 24: When `foot: "R"`

3.4.1. 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 far away from the head foot of a given PWd, the height of the representation will be proportional. This will inevitably create tall figures, as already mentioned — see Figure 25.

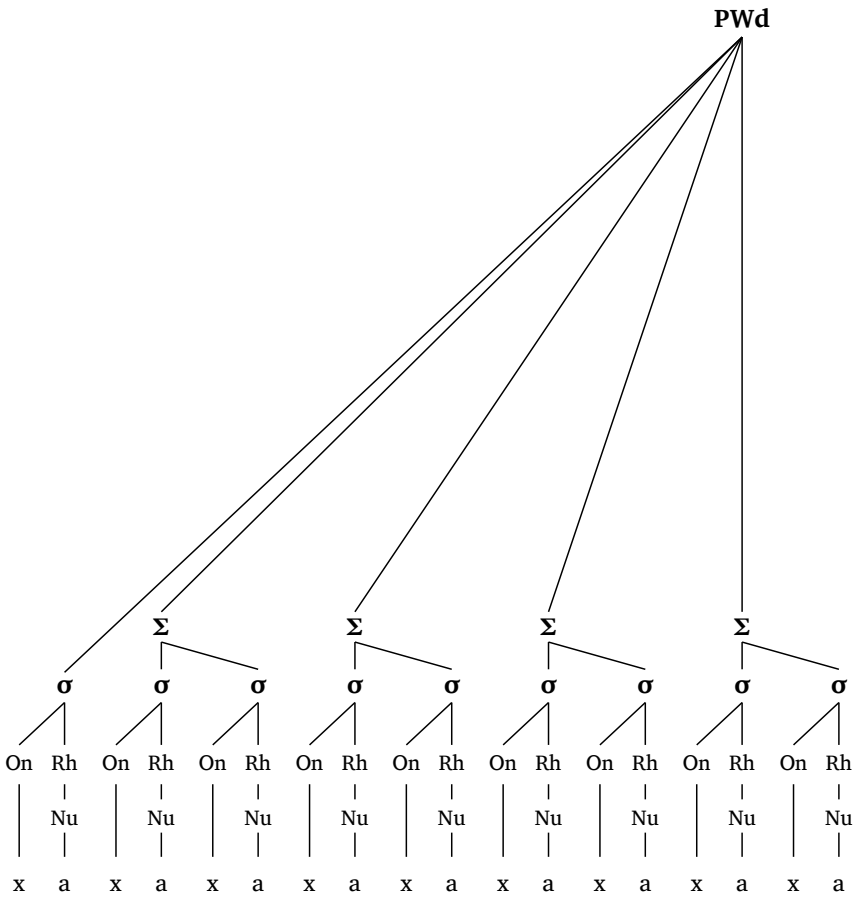

Figure 25: Unfooted syllable far away from head foot.

3.5. 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 as as intuitive and parsimonious as possible. Figure 26 and Figure 27 show grids for the word *butterfly*.

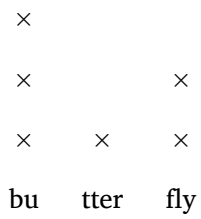

Figure 26: Input as string

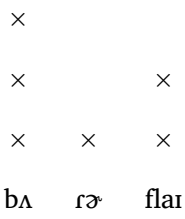

Figure 27: Input as tuple (IPA-compatible)

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

Code example 2: Metrical grids shown in Figure 26 and Figure 27

4. SPE

Rewrite rules can be very complex, and an excellent package already exists to deal with their complexity in Typst (`linphon`). The problem is that too many degrees of freedom exist in SPE-like representations, not to mention the variation across scholars when it comes to symbols, brackets, etc. For that reason, Phonokit only has two primitive functions to help create feature matrices, which in turn can be combined to form SPE-style rules (Chomsky & Halle, 1968)

The first function is `#feat-matrix()` , shown in Code example 3. It simply 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 does not produce *minimal* matrices, but it can be used in sequence to represent matrices for a given word, for example — see Figure 28. The user is free to adjust the inventory of features and their values, since there's variation in the literature. The function in question is based on the features in (Hayes, 2009).

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

Code example 3: Generating matrices for the phonemes in "patchy", shown in Figure 28

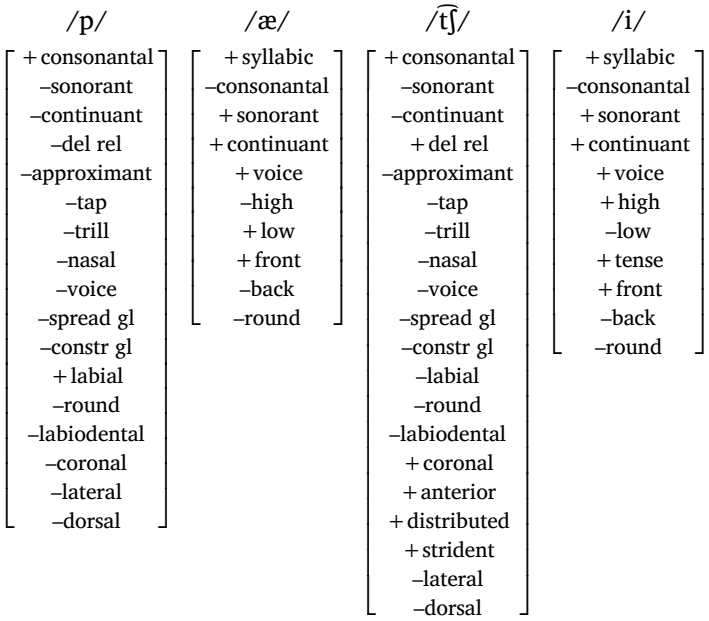

Figure 28: Matrices for the phonemes in the word “patchy”

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 29 is achieved with the code shown in Code example 4 — notice that α notation requires a specific syntax, i.e., `sym.X + "feat"` or `sym.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, `#ar` adds an arrow using New Computer Modern (other arrows are available, such as `#al` ←, `#au` ↑, `#ad` ↓, `#alr` ↔, `#asr` ⇝).

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

Code example 4: Nasal place assimilation using `#feat()`

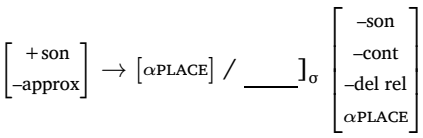

Figure 29: A nasal place assimilation rule

5. Optimality theory

Unlike SPE rules, tableaux in optimality theory (OT; Prince & Smolensky (1993)) are more predictable and constrained. As a result, a function can do a bit more. Phonokit 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: ()`.

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

/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.

6. Maximum Entropy grammars

Finally, the package goes one step further and produces a MaxEnt tableau (Goldwater & Johnson, 2003; Hayes & Wilson, 2008) with the function `#maxent()`. Tableau 2 illustrates a scenario where the data in Tableau 1 is variable, i.e., all candidates in question have a non-zero probability of being observed given a specific input x. The column $H(y)$ displays the Harmony score of each candidate y, calculated as the weighted sum of all constraint violations. Next, the column $e^{-H(y)}$ 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 $P(y|x)$, 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.

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

The function `#maxent()` calculates $H(y)$, $e^{-H(y)}$ and $P(y|x)$ automatically given the weights provided. Tableau 2 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 example 5), 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	$H(y)$	$e^{-H(y)}$	$P(y\|x)$
[kra.θa]	0	0	1	1	0.368	0.598
[ka.θa]	1	0	0	2.5	0.082	0.133
[ka.ra.θa]	0	1	0	1.8	0.165	0.269

Tableau 2: A MaxEnt tableau

In Code example 5, 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, 1),
  violations: (
    (0, 0, 1),
    (1, 0, 0),
    (0, 1, 0),
  ),
  visualize: true  // Show probability bars (default)
)
```

Code example 5: Code to generate a MaxEnt tableau

7. Future directions

Phonokit 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. In future updates, I intend to focus on phonological processes such as spreading, which will build upon the representations presented above.

Typst is still a young language, and most linguists do not know about it yet (as of 2025). But as the language expands into linguistics (which I think will be inevitable), there is *a lot* potential for significant advances in our workflows. I hope this package will make document preparation quicker and more enjoyable to the phonologists out there.

References

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

Fukui, R. (1996). TIPA: A system for processing phonetic symbols in LaTeX. *Tugboat*, *17*(2), 102–114.

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

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

Knuth, D. E. (1984). *The TeXbook*. Addison-Wesley.

Lamport, L. (1994). *LaTeX: A Document Preparation System* (2nd ed.). Addison-Wesley.

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. (1993). *Optimality Theory: constraint interaction in Generative Grammar*. Blackwell.

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.

A. IPA symbol reference

Input	Output	Input	Output	Input	Output	Input	Output
Plosives							
`p`	p	`b`	b	`t`	t	`d`	d
`\\:t`	ʈ	`\\:d`	ɖ	`c`	c	`\\barredj`	ɟ
`k`	k	`g`	g	`q`	q	`\\;G`	ɢ
`?`	ʔ						
Fricatives							
`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
`\\textcrh`	ħ	`Q`	ʕ	`h`	h	`H`	ɦ
`\\textbeltl`	ɬ	`\\l3`	ɮ				
Nasals							
`m`	m	`M`	ɱ	`n`	n	`\\:n`	ɳ
`\\nh`	ɲ	`N`	ŋ	`\\;N`	ɴ		
Approximants & trills							
`V`	ʋ	`\\*r`	ɹ	`j`	j	`\\darkl`	ɬ
`l`	l	`L`	ʎ	`\\:l`	ɭ	`\\;L`	ʟ
`r`	r	`R`	ɾ	`\\:r`	ɽ	`\\;R`	ʀ
`\\textturnmrleg`	ɰ						
Vowels							
`i`	i	`I`	ɪ	`y`	y	`Y`	ʏ
`1`	ɨ	`0`	ʉ	`W`	ɯ	`u`	u
`U`	ʊ	`e`	e	`\\o`	ø	`9`	ɵ
`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						
Diacritics, suprasegmentals, archiphonemes							
`'ta`	ˈta	`,ta`	ˌta	`u:`	uː	`\\~ a`	ã
`\\r i`	i̥	`\\v n`	n̥	`t \\h`	tʰ	`p \\*`	p˞
`\\t ts`	t͡s	`k \\labial`	kʷ	`p \\velar`	pˠ	`t \\palatal`	tʲ
`\\dental t`	t̪	`\\C`	ɕ	`\\V`	ʌ	`\\N`	ɴ

Table 1: TIPA-to-IPA Reference Guide

B. Representing processes

While there are no functions dedicated to phonological processes *per se*, prosodic representations can be linearized and concatenated with arrows, which is frequently enough to show many processes on slides and handouts. Figure 30 shows one simple example generated with Code example 6 — the command `#aR` creates a visually appropriate arrow for this type of figure.

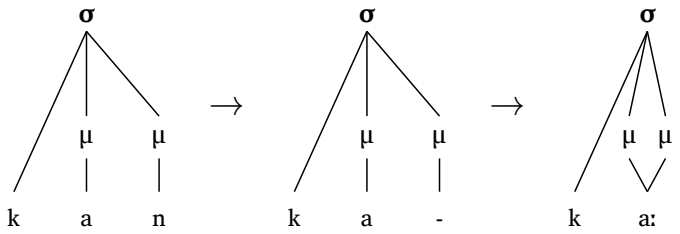

Figure 30: Compensatory lengthening

```
#mora("kan", coda: true) #aR #mora("ka-", coda: true) #aR #mora("ka:")
```

Code example 6: Transcriptions using `#ipa()`