How to use the Syntacticon

Raphael Finkel raphael@cs.uky.edu Daniel Kaufman dkaufman@qc.cuny.edu

August 25, 2026

Contents

1 Introduction **3**

2 Syntax patterns **4**

2.1 Format 4

2.2 Representation 5

2.3 Multiple syntax patterns and wild cards 5

3 Syntax rules **6**

3.1 Format 6

3.2 Conventions 6

3.3 Conditions 8

3.4 Flags 8

3.5 Commands 9

4 Preamble **14**

4.1 Phonology 14

4.2 Roots 16

4.3 MPSs 17

5 Morphophonological rules **17**

5.1 Overview 17

5.2 Conditions 18

5.3 Changes 19

5.3.1 Elements 20

5.3.2 Replacements 22

5.4 Disjunctive rules 23

5.5 Autosegmental phonology 23

6 User interface **26**

6.1 General 27

6.2 Quick links 27

6.3 Discussion 27

6.4 Errors and messages 27

6.5 Results 28

6.6 Metadata 29

6.7 Inputs 29

6.7.1 syntaxPatterns 29

6.7.2 syntaxRules 29

6.7.3 morphologicalRules 30

6.8 Function buttons 30

6.9 Sample rules 30

6.10 Library 30

6.11 IPA chart 31

6.12 Segments and features 32

7 Acknowledgements **32**

References **32**

Index **32**

1 Introduction

The Syntacticon is a web-accessible facility that applies rules to syntax trees to achieve a surface-form utterance. The input to the program includes a syntax tree, syntax rules to manipulate the tree, and morphophonological rules to convert the resulting tree to a surface form. This program is intended both as an aid in teaching syntax and to help researchers formalize syntax theories.

One can use the Syntacticon in several ways.

- Enter a syntax theory and have the Syntacticon compute its results.
- Choose a prepackaged syntax theory from the **Library**.
- Investigate the phonological features of IPA symbols.

The Syntacticon is the third in a series of facilities, building on the previous two: the Phonomaton (for phonological theories) and the Syntacticon (for morphological theories). Many of the features of those facilities are also available in the Syntacticon.

This document is organized according the input sections of the Syntacticon. Each section has a show/hide button that lets the user expand or contract its contents. The Syntacticon preserves the contents of input sections as it creates output, so the user can make modifications and resubmit the input.

2 Syntax patterns

This input section provides the starting tree (or rarely, trees) that the syntax theory manipulates. Each tree representation should be on a single line, or, for convenience, split into multiple lines with at the end of all but the last line. The symbol `%` introduces a comment that continues to the end of the current line.

2.1 Format

Figure 1 shows a simple syntax tree.

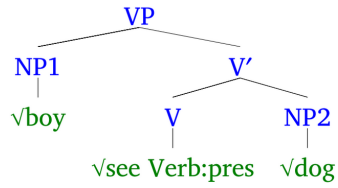

Figure 1: Syntax tree example

Syntax trees should follow a particular format. Nodes in the tree can be phrases, intermediate constituents, or heads. Each node has a **name**. All names in the tree must be unique.

The name of a **phrase** must end with `P`, optionally followed by digits to distinguish similar names. A phrase must have one or two children. Generally, one of the children is an intermediate constituent, and the other is an optional specifier, which is another phrase. It is also valid for a phrase to only contain an expression, described below.

An **intermediate constituent** must have a name that ends with an apostrophe, optionally followed by digits. (The Syntacticon accepts either the ordinary `'` or the Unicode 02b9 `′`). It must have two children, a head and a phrase, which acts as a complement. The children may be in either order. As a special case, usually at the end of a long tree chain, the complement may be a phrase that is also a head.

A **head** may have any name that does not end with an apostrophe (followed by digits). It has no children; its content is a expression.

An **expression** is meant to derive part of the eventual surface form of the tree when it is spelled out. It contains one or more units, separated by space. A **morphological unit** names a morphological rule block and provides input in the form of morphological properties (`MP`s). For instance, the expression `D:def,3sg`

names the morphological rule `D`, perhaps intended to express a determiner, with the MPs definiteness and a 3sg subject. A **root unit** starts with √ and then names a root.

In the tree shown in Figure 1, the root is a phrase (as it must be): `VP`. It has a specifier `NP1` and an intermediate constituent `V'`. The specifier `NP1` is a head, containing the expression `√boy`, referring to a particular root. The intermediate constituent `V'` has a head `V` and a complement `NP2`, each of which is a head. The head `V` contains a expression with two units, the root unit `√see`, and a morphophonological rule `Verb`, perhaps expressing tense, with the morphological property `pres`. Finally, the phrase `NP2` is a head containing `√dog`.

2.2 Representation

A tree can be represented in either a bracket or a brace form. The **bracket form** is more conventional, although more verbose. Each node of the tree is represented in the form

```
[Name Content]
```

The tree shown in Figure 1 has this representation in bracket form:

```
[SP [NP1 √boy] [V' [V √see Verb:pres] [NP2 √dog]]]
```

Because it is clumsy to count matching brackets, the Syntacticon accepts missing brackets at the start or at the end of the bracket representation, adding brackets implicitly as needed. The representation, which should be in a single line, may be split into multiple lines, with intermediate lines terminated by \.

The **brace form** takes advantage of the convention, visible in Figure 1, that an intermediate constituent has the same name as its parent phrase, but with the `P` converted to an apostrophe, and that its head has the same name again, without the `P` or apostrophe. The brace form of the tree shown in Figure 1 is

```
{V {N1 √boy} √see_Verb:pres {N2 √dog}}
```

The Syntacticon automatically adds `P` to phrase names and apostrophes to intermediate constituents. It converts underscores to spaces in heads. It also inserts missing braces at the start and at the end of the brace representation. Again, long lines may be split with \ at the end of intermediate lines.

2.3 Multiple syntax patterns and wild cards

The syntax pattern input may contain multiple trees. In this case, the Syntacticon evaluates each tree separately.

A morphological unit in a syntax pattern may include a **wild card**. For instance, instead of `Verb:pres`,

the syntax pattern may say `Verb:all(Tense)`, referring to all the tenses. This specification refers to the `MPSs` region of the **Inputs** region, described in Section 4.3. In particular, this region might include the following:

```
Tense:  past pres future
```

In this case, the syntax pattern expands to three patterns, one with `Verb:past`, one with `Verb:pres`, and one with `Verb:future`. If a syntax pattern contains several wild cards, the Syntacticon generates all possible combinations as separate trees to be evaluated by the syntax rules.

3 Syntax rules

The purpose of syntax rules is first to manipulate the syntax tree, then to create a resulting surface form of an utterance. Each rule is on a single line, although long lines may be interrupted by a \ at the end of an intermediate line. An empty line separates sets of syntax rules; the Syntacticon applies each set anew to each syntax pattern. The character `%` introduces a comment that continues to the end of the line.

The syntax rules also allow **literate comments**, which start with `%!`, and which are meant to elucidate the purpose of the rules. The Syntacticon collects literate comments and presents them at the top of the output page.

3.1 Format

The format of all syntax rules is

```
RULENAME: COMMAND / CONDITION // FLAGS
```

A rule may omit the `RULENAME` and its colon. The purpose of the `CONDITION` is to restrict when the rule should apply. The purpose of the `FLAGS` is to modify how the rule applies. Both `CONDITION` and `FLAGS` are optional; if they are omitted, the syntax rule should also omit the `/` character(s) that introduce them.

Commands generally modify the syntax tree. Some commands are purely for debugging purposes. One command changes the tree into a surface string.

3.2 Conventions

In the following, we use these conventions to describe syntax rules.

- NAME: The name of a particular node.
- HEAD: The name of a head node.
- NODES: A set of nodes, possibly empty. If there are several nodes in the set, they are sorted breadth-first, left-to-right, according to their position in the tree.
  - a NAME
  - a Perl regular expression surrounded by slashes, such as /DP/. All nodes with names matching the regular expression (anywhere in the name) are included.
  - A specification in braces referring to MPs (in head nodes) and labels (on any nodes), such as {3sg}{past,fut}. All nodes with content or labels matching the specification are included. The rules for morphosyntactic specification are in Section 5.2.
  - A relation between nodes, as discussed below.
- UNIT: A morphological unit, such as Agr:apart,3sg, or a root, such as √come.
- WORD: Any word.
- PATTERN: a syntax tree in bracket form.
- RELATIONS: a comma-separated list of relations between nodes in the syntax tree. Each relation is a test that returns "true" or "false". Spaces are not allowed in relations. The relations are as follows.
  - isSister(NAME, NAME) Tests if the two named nodes have the same parent. In Figure 1, NP1 and V' are sisters.
  - dominates(UPPER_NAME, LOWER_NAME) Tests if the first named node is an ancestor of the second one. In Figure 1, V' dominates NP2.
  - ccommands(UPPER_NAME, LOWER_NAME) Tests if the first named node is either a sister of the second node or if its sister is an ancestor of the second node. In Figure 1, NP1 ccommands NP2.
  - isSpecifier(UPPER_NAME,LOWER_NAME) Tests if the second named node is a specifier of the first node, that is, its parent is the first node and its sister is an intermediate constituent. In Figure 1, NP1 is the specifier of VP.
  - matchesContent(NAME, REGEXP) Tests if the named node or any of its descendants has contents matching the regular expression. In Figure 1, V' matches /Verb:pres/.
  - isOccupied(HEAD) Tests if the given head node contains an expression. In Figure 1, NP1 contains the expression √N1.
  - headPhrase(HEAD_NAME, PHRASE_NAME) Tests if the given head node acts as the head of the given phrase. In Figure 1, V is the head of VP.

- `isHead(HEAD_NAME)` Tests if the given node is a leaf of the tree. In Figure 1, nodes `NP1`, `V`, and `NP2` are all heads.
- `agrees(NODE`$_1$`:RULEBLOCK`$_1$`,NODE`$_2$`:RULEBLOCK`$_2$`)` Tests if the parameters (that is the MPs) provided to the first rule block in the first node are the same (including order) as the ones in the second rule block in the second node. Both nodes must be head nodes. Optionally, the colons and `RULEBLOCK`s may be omitted, in which case the relation tests if all the rule blocks in the first node are identical, including parameters and order, to the ones in the second node.

3.3 Conditions

The condition optionally attached to a syntax rule is formed of a list of relations (no spaces allowed), each of which must hold for the syntax rule to apply. Conditions thereby allow a syntax rule to filter a set of nodes to select only the ones of interest.

Each relation may optionally be prefixed with `!` to indicate negation.

If a relation contains `_` instead of a `NAME`, it refers to the source of the syntax rule; if it has `=` instead of a `NAME`, it refers to the destination for syntax rules that have both source and destination, such as `CopyMP`.

3.4 Flags

Some flags are specific to particular commands; they are discussed as appropriate. Some flags apply to any command.

- `echo`: Show the command in the "Errors and messages" section of the output, for purposes of debugging.
- `once`: Only run the command once if `NODES` expands to more than one node. Here, "run" means "make an effective change to the tree;" nodes where no effective change happens are silently skipped.
- `atEnd`: In the `MoveContent` command, place the content at the start of the destination's content list instead of at the end.
- `global`: In the `Spellout` command, initialize the environment by collecting all the MPs in the tree instead of locally for each head.
- `NAME`: In the `Spellout` command, initialize the environment by collecting all the MPs in the subtree of the node with the given `NAME`.

3.5 Commands

The commands generally modify the current syntax tree.

- `DIR-Adjoin (NODES)(NAME)`

  Here, `DIR` is either `L` or `R`, indicating a left or right adjoin operation. The flags implicitly include `once`. The destination node is replaced by a new node with the same name but with a superscript suffix, such as [1]. The superscript is chosen to make the new name unique in the tree. The new node has the original destination node as one child and the source node as its other child, either to its left or right, as specified in the syntax command. The source node is replaced by a trace `[t]`. For example, applying `R-Adjoin(V)(VP)` to the tree in Figure 1 results in the tree in Figure 3.5.

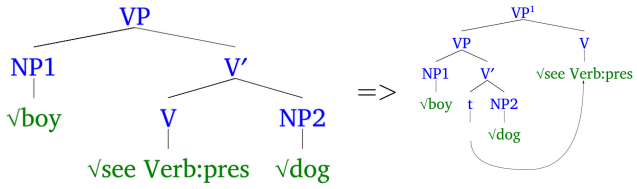

Figure 2: After `R-Adjoin(V)(VP)`

- `AdjoinLabel(WORD)(NODES)`

  Labels the destination nodes with the word. A **label** is like the content of a head node, but it can apply to any node, and it forms a part of the node name, separated by a colon. The purpose of labels is to condition subsequent syntax rules on the presence of the label. For instance, applying `AdjoinLabel(pres)(NP1)` to the tree in Figure 1 results in the tree in Figure 3.

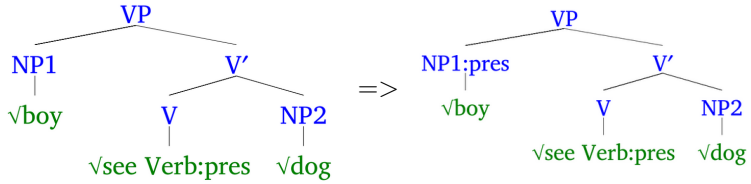

Figure 3: After `AdjoinLabel(pres)(NP1)`

- `MergeRoot(ROOT)(NODES)`

Adds the given root (with a prefix √) at the start of the content of the NODES, restricted to heads. For instance, applying `MergeRoot(dog)(V)` to the tree in Figure 1 results in the tree in Figure 4.

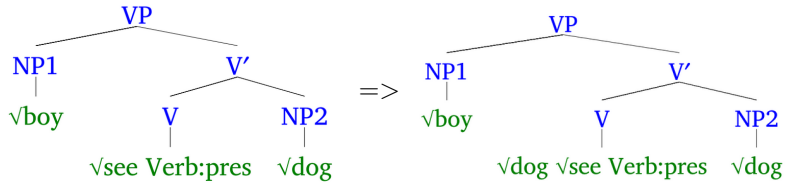

Figure 4: After `MergeRoot(dog)(V)`

- `MergeUnit(UNIT)(NODES)`

  Adds the morphological unit (such as `Agr:past`) to the given nodes, which should be heads that already have an expression. If the morphosyntactic rule set (such as `Agr`) is already represented in the destination node, then the MPs are added to that expression, with duplicates suppressed. Otherwise, the entire unit is added to the expression. For instance, applying `MergeUnit(Verb:pres,active)(V)` to the tree in Figure 1 results in the tree in Figure 5.

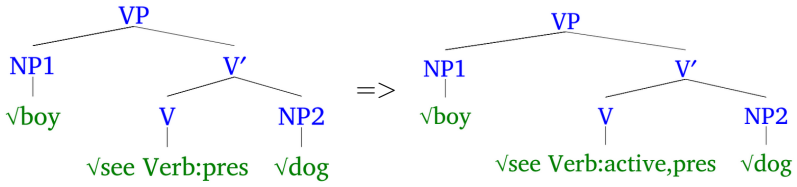

Figure 5: After `MergeUnit(Verb:pres,active)(V)`

- `CopyMP(RULEBLOCK`$_1$`,NODES`$_1$`)(RULEBLOCK`$_2$`,NODES`$_2$`)`

  Copies the MPs of the first morphosyntactic rule set collected from all of the first set of nodes into the second morphosyntactic rule set in all the second set of nodes. For instance, applying `CopyMP(Verb,V)(D,NP1)` to the tree in Figure 1 results in the tree in Figure 6.

- `DeleteContent(NODES,SPEC)`

  Modifies the expression of all the given nodes by removing some content. If SPEC is the name of a morphophonological rule set, it is removed along with all its MPs. If it is a single MP, it is removed from all all morphological units it belongs to. For instance, applying `DeleteContent(V,Verb)` to the tree in Figure 1 results in the tree in Figure 7.

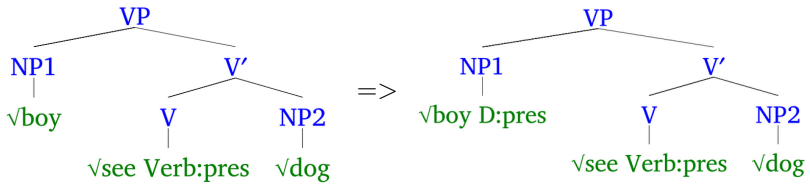

Figure 6: After `CopyMP(Verb,V)(D,NP1)`

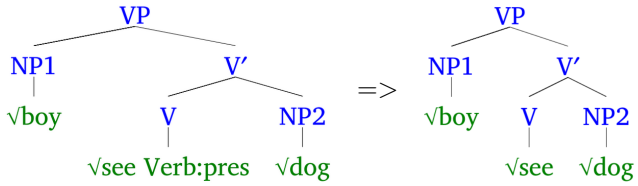

Figure 7: After `DeleteContent(V,Verb)`

- `DeleteNode(NODES)`

  Removes all the nodes, which must be phrases, along with their children. For instance, applying `DeleteNode(NP1)` to the tree in Figure 1 results in the tree in Figure 8.

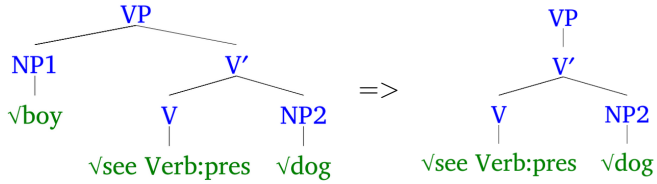

Figure 8: After `DeleteNode(NP1)`

- `MoveContent (NODES`$_1$`)(NODES`$_2$`)`

  The contents in the first set of nodes (the sources, restricted to heads) are moved to become or add to the contents in the second set of nodes (the destinations, restricted to heads). Proper use of this command requires that the `CONDITION` restrict the destinations to a single node, although `NODES`$_2$ may specify more than one node. The list that the Syntacticon generates for `NODES`$_2$ are sorted in reverse to the usual breadth-first, left-to-right order. Each source node acquires a special content, marked by `[t]`, indicating a trace remaining after its original content has moved. Generally, the

contents are placed at the start of the content list of the destination node; the flag saying `atEnd` overrides this convention. However, a content starting with a root (such as `√N1`) always goes to the front. For instance, applying `MoveContent(V)(NP1)` to the tree in Figure 1 results in the tree in Figure 9.

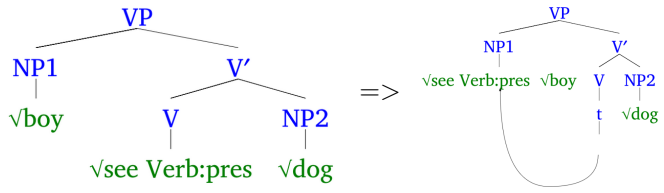

Figure 9: After `MoveContent(V)(NP1)`

- `MoveToSpecifier (NODES)(NAME)`

  Move the first valid node in the source to the specifier sibling of `NAME`, which must be an intermediate constituent that has no specifier. By default the node goes to the left of the destination, but if the destination has a right-hand sibling with the special name `[Spec]`, then the source is moved to the right-hand position, replacing `[Spec]`. The flags implicitly include `once`. For instance, applying `MoveToSpecifier(V)(VP)` to the tree in Figure 8 results in the tree in Figure 10.

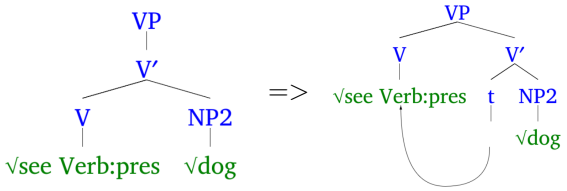

Figure 10: After `MoveToSpecifier(V)(VP)`

- `Spellout`

  This syntax rule expands all contents in heads, applying morphophonological rules to roots in those contents. It therefore refers to information about those rules; we describe those sections of the input in detail later. For now, we only present the necessary information to apply `Spellout` to the tree in Figure 1.

```
Roots
boy: boj
see: «seg» si «morph» pres | «seg» sɔ «morph» past
dog: «seg» dɔg
```

Applying `Spellout` to the tree in Figure 1 results in the tree in Figure 11.

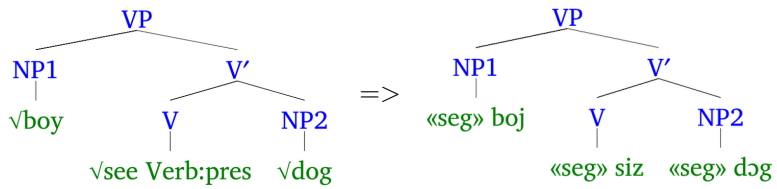

Figure 11: After `Spellout`

- `Apply (RULEBLOCK)(NODES)`

  This syntax rule is usually applied after `Spellout` to apply a particular block of rules, typically phonological rules, to the head-node descendants of the given nodes (typically phrases). As a special case, `NODES` can be the word `flattened`, in which case this rule is applied to the flattened (non-tree) current form.

- `Flatten`

  This syntax rule is usually the last one. It collects all the `«seg»` content of the heads in left-to-right order and results in a string, not a tree. Figure 12 shows the result of `Flatten` on the tree in Figure 11.

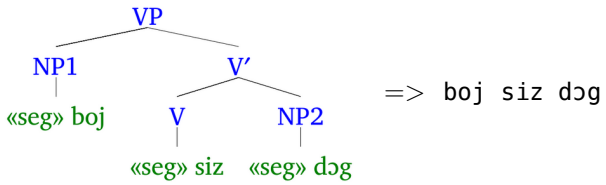

Figure 12: Result of Flatten

- `Crash (message)`

  This syntax rule is for debugging. It causes the Syntacticon to skip all remaining syntax rules (except for `Spellout` and `Flatten`). It usually is associated with some condition.

- `Expect TREE WHEN`

  This syntax rule is for debugging. It compares the given tree (which should be in bracketed form) against the current tree. If it succeeds, the given tree is shown in green (in the `Results` region of the output). If it fails, the tree is shown in red. This result does not affect the current tree in either case. The optional `WHEN` specifies the name of the result group as shown in the result region, typically indicating the wild-card replacements (if any) or a serial number such as `1.3`. If there is a `WHEN` specification, the `Expect` command only applies to that group.

4 Preamble

The preamble input section contains several regions, which are separated by a blank line.

4.1 Phonology

The Syntacticon associates segments with features. A **segment** is typically an IPA symbol, such as `ɛ`. **Features** are those discussed by Hayes [Hayes (2009)], including `syllabic`, `nasal`, and many more. Features of a segment have a `polarity`, which can be positive (`ɛ` is `[+syllabic]`), negative (`ɛ` is `[−nasal]`), unlicensed (`ɛ` is `[0strident]`, or unspecified (one can introduce a symbol with `[⊗dorsal]`.

The morphophonological rules in the Syntacticon generally produce segments that are IPA characters. However, many languages require IPA diacritics, such as `ú`, and some theories make use of custom phonemes. The Phonology section introduces such symbols. This section of the preamble starts with the word `Phonology` on a separate line and continues on the subsequent lines.

- `Inventory:` This keyword is followed by a space-separated list of IPA symbols (and diacritics if needed) that defines a restricted inventory of symbols, replacing the full inventory that the Syntacticon starts with. If the forms only use IPA symbols without diacritics, this specification is not necessary. But if any segments need diacritics, then this line is required and must include all segments, including standard IPA symbols, along with any diacritics, such as `Inventory: a á ã b t x`. Also include in the inventory any unusual punctuation symbols that you use, such as `✥`.

  If a rule would produce a segment outside the inventory, the rule is ineffective; any changes (even unrelated to the failed change) are abandoned, but the Syntacticon generates an error message.

  The shorthand `<all>` introduces all IPA symbols without diacritics. So one can write `Inventory: <all> á ã`.

Another shorthand adds diacritics that are associated with features to a set of IPA symbols.

It has this form: `<a,i,u,ɪ><tone_high,long;tone_low>` The available features include secondary articulations, such as `velar`. Spaces are not allowed in this shorthand. All the IPA symbols in the first delimited section (here `a,i,u,ɪ`) are entered into the inventory along with diacritics specified in the second section. That section has semicolon-delimited subsections (`tone_high,long` and `tone_low`). Each subsection has features (they can be either ordinary features or secondary articulations, such as `velar`).

Each subsection is independently applied to the given IPA symbols. If it lists n features, the result is all 2^n combinations of those features. In the example above, the result is `a á aː áː i í iː íː u ú uː úː ɪ ɪ́ ɪː ɪ́ː à ì ù ì̀`.

- `Define:` This keyword introduces **custom phonemes**. It is followed by a space-separated list of letters, typically capital Latin letters (but not `XVCSU`, which have special meaning) or Greek letters (but not `αβγδ`, which are used for variable polarities). These letters are intended to be used as morphophonological segments in underlying and intermediate, but not surface, forms. Each letter may be associated with a bracketed list of features with polarity, such as `A[+syllabic,-low]`. If a feature is omitted, it is treated as underspecified. Alternatively, a new letter may inherit all the features of an existing phoneme, with modifications, such as `A=a[+tense,0tone_low]`. The polarities may be any of `0+-⊗`.

  The Syntacticon automatically introduces any related features. For example, `[+low]` implies `[-high]`, and `[+round]` implies `[+labial]`.

  The features may include geometry clusters (see below), but only with polarity `⊗`.

  Don't include custom phonemes in the inventory specification.

- `Redefine:` This keyword is followed by a single IPA symbol representing a segment, followed by a bracketed list of features with polarity. This specification overrides the usual feature list for this segment, but the segment retains any features not mentioned in the list. The features may include geometry clusters (see below), but only with polarity `⊗`.

- `Disallow:` This keyword is followed by a space-separated set of patterns, much like the patterns in targets of morphophonological rules. Any rule that would result in a form that matches one of these disallowed patterns fails to apply.

- `Delete:` This keyword is followed by a bracketed list of features (without polarity) that the Phonomaton then removes from its set of features. Such features may not appear in any rule. Removing features can make multiple segments appear identical; for example, `Delete: voice` makes `t` and `d` indistinguishable.

- `Geometry:` A **geometry cluster** is a set of features, typically related in some way to each other. The Phonomaton predefines several useful geometry clusters:

name	feature set
cor	coronal anterior distributed strident
dor	dorsal high low front back tense
lab	labial labiodental round
place	cor dor lab
lar	constr_gl spread_gl voice

  You can introduce a new geometry cluster with the keyword `Geometry:` followed by a new name, `=`, then a space-separated list of features. For instance, `Geometry: StrTense = strident tense` introduces a new geometry cluster called `StrTense` that refers to the two features, `strident` and `tense`. New geometry clusters introduced by `Geometry:` may refer to these predefined geometry clusters as well. Geometry clusters can often be treated just the same as individual features. One such use is in in Section **??** when we discuss the pseudo-polarity `∀`.

4.2 Roots

This section of the preamble begins with `Roots` on its own line, followed by subsequent lines. Its purpose is to present the principal parts of vocabulary that the syntax rules might need. When a syntax rule refers to a root, such as `√see`, the Roots section of the preamble specifies its details. The details always include the relevant segments in the `«seg»` tier. They can also include category information in a `«cat»` tier, such as `noun` or `verbE` (for e-stem Latin verbs),

Simple roots, such as English nouns, can have short descriptions:

```
boy: boj
```

The tier `«seg»` is implicit.

Roots with multiple principal parts generally have morphological restrictions indicated by bracketed constraints in the `«morph»` tier:

```
see: «seg» si | «seg» sɔ «morph» past | «seg» seen «morph» pastPart
```

The Syntacticon uses the morphological information in choosing morphophonological rules, as discussed in Section **??**orphophonological.

4.3 MPSs

This section of the preamble begins with `MPSs` on its own line, followed by subsequent lines. Its purpose is to specify morphosyntactic property sets (MPSs) and their properties, such as

```
Tense:  past pres future
Pol: neg pos
Voice:  active pass
Aspect: perf prog
```

This information allows the Syntacticon to generate multiple syntax trees with wild cards and to select morphophonological rules within rule blocks.

A morphosyntactic class (MC), such as `Aspect`, can have one or more **binary** values, such as `+causative +repetitive`. If there are n binary values, then all 2^n combinations exist for this property, such as `+causative,+repetitive`, `+causative,-repetitive`, `-causative,+repetitive`, and `-causative,-repetitive`.

An MC name can have the suffix `?` to indicate that its presence is optional. This feature is generally used for single-value MCs, and is therefore equivalent to using a binary value indicated by `+`.

5 Morphophonological rules

5.1 Overview

The input region of morphophonological rules is organized into blank-line-separated **rule blocks**. Each rule block is prefaced by a single line indicating its **block name**, followed by an optional colon. The content of each block is a list of rewrite rules that convert **forms**. Most morphological rule blocks are **selective**: the Syntacticon selects at most one rule to apply. Other blocks are **inclusive**; they are marked with `(all)` after their name.

The Syntacticon applies rule blocks in response to the `Spellout` and `Apply` syntax rules. The form that is input to a block is given in the current syntax tree. In an `all` block, each rule provides input to the next one in the block.

A rule may have an optional `rule name`, which can be any string followed by a colon. Rule names are only useful for documentation; they need not be unique.

Following the name, a rule has a list of **components**. For instance, this rule

```
future-m: «cat» {conj3,conj4} «morph» {1sg} {future} «seg» 0 -> m / _>
```

has three components, each prefaced by a tier name in guillemet marks (« »). Some components are **conditions**, such as the «cat» and «morph» components. Others are **changes**, such as the «seg» component. A rule only applies if all its conditions are satisfied. Here, the conditions are satisfied if the «cat» tier, which contains information from the lexicon, has either conj3 or conj4 and the «morph» tier, which contains features from the syntax tree, has both 1sg and future. If the conditions are satisfied, the Syntacticon attempts all change components. In this example, the **target**, is the empty string in the «seg» tier, represented by 0. This target matches between all segments of the current form. The **replacement** is m. The **context** is _>, which restricts the target to be followed immediately by the end of the form. In other words, this rule puts the segment m at the end of the current form if the root category is either conj3 or conj4 and the morphological context is both 1sg and future.

If the current form does not match the target, the rule has no effect. Details of change components are in Section 5.3.

5.2 Conditions

The future-m rule above has two tier-based conditions. The first is in the «cat» tier with the single constraint {conj3,conj4}. This constraint is only met if that tier contains either conj3 or conj4. The second condition is in the «morph» tier; it has two constraints, which are only met if that tier contains both 1sg and the future.

Within a constraint, a value may be negated, such as {-conj4}. A negated value matches only if that given value is not in the current form.

In addition to a bracketed list of values, a constraint may be a bracketed pattern. For instance, the constraint {/1/} in tier «morph» matches 1sg and 1pl, but not 3sg.

If all the constraints in a rule are met by the current form, the rule is **available**. The **Pāṇini precedence** of the set of constraints in a rule is $\sum_j 1/size_j$, where the size of a constraint is the number of values it contains. If a rule has no constraints, its Pāṇini precedence is a minimal positive number. The future-m rule has Pāṇini precedence 2.5. As it applies a selective block to a form, the Syntacticon selects that available rule with the highest Pāṇini precedence. If there is a tie, the Syntacticon complains about the ambiguity.

Occasionally one needs to artificially raise the Pāṇini precedence of a constraint to avoid ambiguity. The easiest way to do that is to duplicate one or more of the constraints.

An entire rule block may be subject to conditions. The conditions are placed after the rule name, the optional colon, and the optional `(all)` designation. They must start with a tier name, such as `«morph»`. The entire block is skipped if the conditions fail.

5.3 Changes

Changes are applied to the current form, typically to the `«seg»` tier. A rule may have several changes, which the Syntacticon applies sequentially to the form. We say that a change is **effective** if its target and environment match the form and if its replacement modifies the form. If any of the changes in a rule fails to be effective, all the changes are blocked and the form remains as it was.

A change component has these constituents.

- **tier**, such `«seg»`. If a rule has only one component, then `«seg»` is assumed.
- **target**, such as `0`. Targets can be much more complex, such as `C¹V²[+long]`.
- **replacement** after an arrow (`→` or `->`), such as `[-tense]`
- optional **environment** introduced by '`/`', such as `_>`. The environment must have a single underscore '`_`'. The (optional) material before the underscore constitutes the **previous context**, and the (optional) material after it constitutes the **subsequent context**.
- **options** introduced by '`//`' that modify the way the Syntacticon applies the rules. By default, a rule applies every place in which it can within a form, even if those places overlap. Options override this behavior.
  - `once`: The rule applies only once, in the leftmost place in the form, even if it could apply in other places as well. For example,

    ```
    intensify: V → [-tense] // once
    ```

    changes `ibik` to `ɪbik`, modifying only the first `i`.
  - `iterate`: The rule applies repeatedly until it makes no further changes. For example,

    ```
    spreadTense: [+syllabic] → [+tense] / _C[+tense] // iterate
    ```

    changes `ɔfɪfi` to `ofifi`; without iteration, the result would be `ɔfifi`.
  - `exclusive`: The rule applies in all places, left to right, but suppressing later applications if they apply in a region matched by an earlier application. For example,

    ```
    Simple onset: ø → ʕ / _C?V // exclusive
    ```

changes abracadabra to `ab‹ra‹ca‹dab‹ra`; without exclusion, the result would be `‹ab‹r‹a‹c‹a‹d‹ab‹r‹a`.

The target and the environment constituents are composed of a sequence of **elements**, the concatenation of which is matched against the current form. A change matches a form if its target, surrounded if specified by the previous and subsequent contexts, matches the form. It can match in more than one place in the form. If a rule matches a form, the Syntacticon replaces all sequences in the form that match the target with the replacement (unless this behavior is overridden by an option such as `once` or `exclusive`).

5.3.1 Elements

An element may be

- an IPA letter, with optional diacritics, representing a segment, such as `ʁ` or `p͡f` or `á`, Such an element matches the identical segment in the form.
- a feature list: a bracketed, comma-separated list of features with polarities, such as `[+consonantal,-continuant]`. The polarities are usually `+` or `−`; we discuss other polarities in Section **??**. A feature list matches any segment in the form that agrees on the given features, ignoring any feature that it does not explicitly list.
- a shorthand: `V` for “vowel”, `C` for “consonant”, `X` for “any defined segment”, `S` for “white space”, and `U` for “subscript”. Subscripts arise in autosegmental phonology, discussed in Section 5.5.
- Null: `⌀` (Unicode 2005; one may use the ASCII `0` instead.) The null element matches between segments in the form.
- a custom phoneme, introduced by a `Define` clause in the Phonology section of the Preamble.
- a punctuation symbol. These symbols need not be listed in the inventory.

Symbol	Unicode	Conventional meaning
=	u003d	enclitic
⇒	u21d2	proclitic
-	u2011	affix symbol (not a minus sign)
'	u20c8	stress symbol (not an apostrophe)
⟨	u27e8	left infix boundary
⟩	u27e9	right infix boundary
≣	u2263	juncture
❬	u276c	left phrase boundary
❭	u276d	right phrase boundary
~	u0073	reduplicant boundary

- various components of Perl pattern-matching regular expressions (regexps), such as parentheses and repetition marks (*, +, and ?).
- abbreviations for more complex Perl regexps: ^ (the start of the form), $ (the end of the form), # (the start or end of a word in the form), < (the start of a word), > (the end of a word).
- **Ellipsis** extends Perl pattern matching. The expression ⟨...⟩ matches any form surrounded by the given brackets, even if there are enclosed bracketed expressions. The brackets may be any Unicode bracketing pair except for parentheses and { }, which have other meanings. Some examples: [ ] and ⌊ ⌋. Similarly, the expression ⟨...abc⟩ matches any bracketed expression that contains, in order, a, b, and c (which must be IPA segments), possibly with other intervening segments. For example, consider these rules

  hx: ⟨...hs⟩ → ∅
  ha: ⟨...ha⟩ → ∅
  hn: ⟨...hn⟩ → ∅

  with current form this⟨⟨that⟩then⟩. The hs rule makes no change. The ha rule results in the form this⟨then⟩ . The hn rule results in this.
- **Indexing** extends Perl pattern matching. A pattern in the target may include a superscript (limited to ¹²³⁴) following a segment-like part of the target to index it. A segment-like part is any of these:
  - A segment: an IPA symbol with optional diacritics
  - A bracketed list of features, such as [+voice,−syllabic].
  - A parenthesized region, such as (θɪs).
  - A shorthand: V, C, or X. (Shorthands are implicitly parenthesized.)

The index can then appear in the change to refer back to the part of the form that matches the indexed part. For example,

```
reduplicate: <(CV)¹C² → ²ə¹¹²
```

changes `fonim` to `nəfofonim`.

- **First occurrence** extends Perl pattern matching to match the first occurrence of a pattern in the target or subsequent context. There are two versions of the feature:
  - `ᶠ([+coronal]&[+trill])` matches the first `[+coronal]`; if there is none, the pattern does not match. If it finds an appropriate segment, it then succeeds only if that segment also matches `[+trill]`. The outer parentheses are required.
  - `ᶠ[+coronal]` matches the first `[+coronal]`; if there is none, the pattern does not match. The pattern may be surrounded by parentheses or brackets; Shorthands `V`, `C`, and `X` are implicitly parenthesized.

  For example,

  ```
  Rule1: x → g / _ᶠ([+coronal]&[+trill])a
  Rule2: x → g / _ᶠ[+coronal]a
  ```

  If the underlying representation includes `axaaalaa` and `axaaaraa`, then `Rule1` only applies to `axaaaraa`, because the first `+coronal` after `x` (`l` or `r` in the two underlying representations) must also be `+trill`, which is only true of `r`. However, `Rule2` applies to both underlying representations.

5.3.2 Replacements

The replacement may be

- `∅`; such a replacement simply deletes the matching target.
- a sequence of IPA letters with optional diacritics.
- a bracketed set of features, such as `[+voice,−tense]`. Such a replacement is only appropriate if the target matches a single segment. It modifies that segment by changing the polarities of the given features as shown, if possible. If no valid segment results, no change occurs. The Syntacticon accepts several special polarities that only apply in replacements; we discuss them in Section **??**.

5.4 Disjunctive rules

A morphophonological rule can be conditioned on whether the current form matches a given pattern. For example,

```
pluralize: IF [+strident]> THEN ø → əz / _> ELSE ø → z / _>
```

changes `kɪs` to `kisəz` and `kɪd` to `kɪdz`. The condition may be any pattern. The `THEN` and `ELSE` parts may be any rules, including environments and options. The `ELSE` part is special: it may be omitted (in which case the condition failing leads to the rule not triggering) and it may be, recursively, a disjunctive rule. The `THEN` part must not be disjunctive.

A similar condition is to place an `UNLESS-DO` sequence in the change after the tier name. For example,

```
pluralize: UNLESS [+strident]> DO ø → z / _>
```

changes `kid` to `kɪdz` but does not modify `kɪs`.

Disjunctive rules are intended for morphological rules, building morphological forms that depend on the phonology of stems. However, they are also valid for phonological rules.

5.5 Autosegmental phonology

The Syntacticon is able to deal with autosegmental phonology, first introduced by Bird and Ladd (1991). Besides the standard tiers, `«seg»`, `«morph»`, and `«cat»`, one can introduce other tiers, in particular, `«tone»`. The roots may include the `«tone»` tier. For example, the root in the following example contains both a `«seg»` tier and a `«tone»` tier.

```
utterance: «seg» gaga da bala ma «tone» HL N LH HL
```

Only a restricted set of symbols may appear in the `«tone»` tier: `THMLBthmlbN`. These refer to tones `top`, `high`, `medium`, `low`, and `bottom`, then their floating versions, then `unassigned`.

Autosegmental rules often have multiple change components. For example,

```
noDuplicateVowels: «seg» V¹ → ø / ¹_ «tone» H → N
```

simultaneously removes the second of a pair of identical vowels and reduces its tone to unassigned, but only if its tone was high.

In addition to ordinary change components, multiple-tier changes also respond to special autosegmental rules. Many of these rules take **flags**, signalled by `--`. We illustrate these rules by considering the following morphophonological rule block:

```
Sandhi (all)
deduplicate: «tone» H → N / _S*H
Associate:
Spread:
Realize: --null=tone_low
```

applied to this form, which contains an entire utterance:

```
«seg» gaga da bala ma «tone» HL N LH HL
```

The Syntacticon shows the results of this theory as shown in Figure 13. It aligns words to express their correspondence.

Original	[SP [S′ [S √utterance Sandhi:]]]			
Starting Spellout				
root⌿utterance	gaga HL	da N	bala LH	ma HL
Sandhi	gaga HL	da N	bala LN	ma HL
Associate	ga_1ga_2 H_1L_2	da_0 N_0	ba_3la_0 L_3N_0	ma_{45} H_4L_5
Spread	ga_1ga_2 H_1L_2	da_0 N_0	ba_3la_3 L_3N_0	ma_{45} H_4L_5
Realize	gágà dà bàlà mâ			
Finishing Spellout	[SP [S′ [S «seg» gágà dà bàlà mâ]]]			
Flatten	gágà dà bàlà mâ			

Figure 13: Autosegmental example

The `deduplicate` rule applies to the `«tone»` tier, replacing the `LH` (corresponding to `bala`) by `LN`. The output shows this result in the `Sandhi` row.

The other rules are examples of the following.

- `Associate: --tiers=tone-seg --dir=L-R --features=[+syllabic] --endpoint=# --max=3`
  This rule indicates which tiers to associate, the direction of association within a chunk (either `L-R` (the default) or `R-L`), to which segments in tier_2 to associate (by default, vowels), how to separate the chunks (by default, by word boundaries), and how many elements of tier_1 at most to associate with a segment in tier_2 (by default, 3). Traditionally, autosegmental theories display links between tiers by drawing lines. Instead we show association by a common subscript in two tiers. So the first `a` in the `«seg»` tier is associated with the first `H` in the `«tone»` tier. The `N` tone is always associated with subscript `0`. Often the number of tones is identical to the number of vowels. However, the last word

ma has two tones, HL. The result has two subscripts on the «seg» tier: ma$_{45}$.

- Spread: --tier=seg --dir=L-R --endpoint=#
  This rule spreads associations in the given tier in the given direction across 0-subscripted elements. In our example, it spreads L$_3$ to associate with both vowels in bala, changing ba$_3$la$_0$ to ba$_3$la$_3$ in the «seg» tier.

- Realize: --tiers=tone-seg --null=tone_low
  This rule uses tier$_1$ (typically «tone») to realize the associated values in tier$_2$ (typically «seg»). Unassociated segments acquire the value in the --null option, which has no default value and must be stated explicitly. In our example, the result is gágà dà bàlà mâ. The a in da is associated with the N tone, so it becomes tone_low. The final a is associated with two tones, H and L, so it acquires a falling tone.

The Syntacticon recognizes three autosegmental rules not covered by the example in Figure 13. We demonstrate them by considering the following morphophonological rule block:

```
Sandhi (all)
Associate:
glide formation: «seg» [+syllabic,+high] → [-syllabic] / _Ua
Reassociate: --max=2
Join:
Split:
Realize: --null=tone_low
```

applied to this form, which contains an entire utterance:

```
«seg» ti tiari «tone» L LHL
```

The glide formation rule uses the abbreviation U in the environment; this abbreviation matches any numeric subscript, such as those introduced by Associate. Figure 14 shows the output for this theory.

- Reassociate: --tiers=tone-seg --dir=L-R --features=[+syllabic] --endpoint=# --max=1
  This rule moves associations that no longer apply to later (if L-R) or earlier (if R-L) segments. An association no longer applies if a rule has changed its segment so it no longer satisfies the features parameter. In our example, after the glide formation rule, link 2 is now associated with j, which is not syllabic. Reassociate fixes that problem. It is necessary to override the default max=1, though, to allow two tones to link to the a in tjari, resulting in tja$_{23}$ri$_4$.

- Join: --tiers=tone-seg

Original	[SP [S′ [S √utterance Sandhi:]]]
Starting Spellout	
root⩥utterance	ti tiari H LHL
Associate	ti_1 $ti_2a_3ri_4$ H_1 $L_2H_3L_4$
Sandhi	ti_1 $tj_2a_3ri_4$ H_1 $L_2H_3L_4$
Reassociate	ti_1 $tja_{23}ri_4$ H_1 $L_2H_3L_4$
Join	«tone-seg» $ti_1H\ tja_2L_3Hri_4L$
Split	ti_1 $tja_{23}ri_4$ H_1 $L_2H_3L_4$
Realize	tí tjǎrì
Finishing Spellout	[SP [S′ [S «seg» tí tjǎrì]]]
Flatten	tí tjǎrì

Figure 14: Autosegmental example

This rule places information from $tier_1$ into $tier_2$, resulting in just one tier, here called «tone–seg» for the default tiers. The purpose is to combine the tiers into a single pseudo-tier so further rules can refer to the combined content. In our example, each subscript in the «seg» tier is followed by the associated tone in the «tone» tier.

- `Split:`
  This rule undoes the effect of the `Join` command, creating the original two tiers, so far as possible. It takes no options. In our case, no rules intervene between `Join` and `Split`, and the Syntacticon therefore re-creates the previous two-tier value.

6 User interface

The Syntacticon is an interactive web site with many features that assist the user in preparing syntactic theories. The easiest way to see the features is to go to the `Library` section and click any of the green buttons. The Syntacticon then accepts the theory you have selected and presents its results. The following discussion describes the result page from top to bottom.

6.1 General

Both the **Metadata** and the **Inputs** regions contain a show/hide button, which lets you toggle whether or not the region is shown.

There are Submit buttons in several places. These buttons submit the current theory. They do not refresh the material in the lower part of the page (Library, IPA chart, Segments and features). While the Syntacticon is working, the background of the page turns green; it reverts to its standard color when the results are ready. You can also invoke the submit function by typing <ctrl-enter> anywhere in the page. All submissions are logged for debugging purposes.

The **Inputs** region includes a Keyboard button that brings up a frame with buttons for characters that are not in the ASCII set, particularly IPA symbols and features that give rise to diacritics. Clicking any of these buttons places the corresponding character or diacritics at the current cursor position in the region. The Close button at the bottom dismisses this frame, as does a second click the Keyboard button.

6.2 Quick links

Right after the banner is list of sections. You can click any of the items to move to the named section, which is helpful if there are lots of results that you want to quickly move past.

6.3 Discussion

This section only appears if the theory includes literate comments. It collects all the literate comments from the theory (lines starting `%!` in the Roots, MPSs, syntaxRules and morphological rules parts of the **Inputs** region). A literate comment follows the rule it documents; literate comments at the head of a section are generic to the whole section.

6.4 Errors and messages

This section only appears if the theory is malformed in some way. The Syntacticon tries to continue even in the presence of errors, but it ignores rules that have errors.

6.5 Results

The pop out button generates a separate window showing the results, which lets you examine results while you look at other things such as the syntax rules in the main screen. The results window updates when you resubmit the theory.

The results are organized in groups. Each syntax pattern in the input has its own group, with a subgroup for each substitution of `all()` in that pattern.

Within a group are lines for the original syntax tree and for the result of each syntax rule that modifies the tree. Syntax rules that do not apply (for example, because their condition is not met) do not generate a result line; rules that apply more than once (for example, when the source pattern refers to several nodes) generate multiple result lines.

Figure 15 shows the line that results from invoking `AdjoinLabel(pres)(NP1)`, as shown in Figure 3.

Figure 15: Result of `AdjoinLabel(pres)(NP1)`

A result line has two parts. On the left is a heading based on the syntax rule as it applies. In figure 15, the heading is `AdjoinLabel-NP1`. If you hover the cursor over this heading, you see the literate comment associated with this rule, if any.

On the right is either a bracket-form representation of the resulting syntax tree or the `«seg»` tier of the result of applying a morphophonological rule. In figure 15, this representation is `[VP [NP1:pres √boy] [V′ [V √see Verb:past] [NP2 √dog]]]`. If you hover the cursor over this section, you see the name of the rule. If the rule has no name, the Syntacticon invents a name based on the rule itself.

If you click the bracket-form representation, the Syntacticon opens a window with a graph of the tree. If you resubmit the theory, that graph acquires a red line at the bottom, indicating that it might no longer apply to the updated results. You can dismiss the graph window by typing `q` in that window.

After the syntax rule `Flatten`, the right side shows the content of all the `«seg»` features, collected in left-to-right order from the tree. If you click this information, the Syntacticon sends the IPA to a text-to-speech program and plays the resulting sound. The result is not necessarily of high quality. In addition to a media-control interface, which allows you to replay the sound, the Syntacticon also presents a new button, spectrogram, which you can click to send the resulting sound file to a Praat routine to

generate a spectrogram of the sound. The result can be useful in demonstrating how spectrograms look.

6.6 Metadata

This input region contains optional information about the syntax theory. Its fields, with sample values, are these:

```
Language = English
Date = 2024-12-3
Family = Germanic, Indo-European
Code = stan1293
Sources = various books
Implementation = Jane R. Doe
Details = active and passive forms
Complexity = 5
```

The `Code` is, by convention, the Glottolog code [Hammarström et al. (2020)]. The `Complexity` is a guess at the difficulty of constructing and understanding the theory, on a scale from 1 to 5.

6.7 Inputs

The **Inputs** region contains several parts, each headed by a line starting with `[` naming the part.

The populate button establishes initial information in some of the parts regions of **Inputs** region based on inspecting the syntax patterns. It establishes (1) `Roots` (see 4.2) based on entries in the syntax pattern starting with √, (2) MPSs (see 4.3) based on occurrences of `all()`. (3) (see 5), empty morphophonological rule blocks for every occurrence of a word followed by a colon, such as `C:int`.

6.7.1 syntaxPatterns

This content of the input region is described in detail in Section 2.

6.7.2 syntaxRules

This input region is described in detail in Section 3. The suggestions button opens or closes a short summary of the syntax rules.

6.7.3 morphologicalRules

This input region is described in detail in Section **??**.

6.8 Function buttons

The row of buttons after the **Inputs** region provides several ways to enter a theory. The Submit button causes the Syntacticon to evaluate the current theory. It does so without reloading the whole page, so any changes you have made to an input region you can usually undo with `<ctrl-Z>`. The Refresh-submit button also submits the theory, but it also refreshes the whole page. This function is seldom needed, but it is helpful if the Syntacticon has gotten confused (which would be a software bug). You can use the browser's "back" arrow to return to the previous page.

The checkbox `show all tiers` is useful for debugging; it causes the results lines to show not only the `«seg»` tier but any other tier provided by the lexicon.

The Clear button empties all the input regions.

The Download button formats the theory and downloads it to the browser as a text file on your computer. You can edit this file and then uploaded it by selecting it with the Browse button and then using any of the Submit buttons (or the `<ctrl-enter>` equivalent).

6.9 Sample rules

We have found many morphophonological rules to be helpful. Some of these are visible if you expand **Sample rules**. The rules are categorized by general type, such as **Syllabification** and **Footing**. Each rule is presented as a button, such as `Labialization: C → [+round]`. Clicking a rule adds it to the end of the current block of morphophonological rules. You most likely want to modify the rule to fit your theory.

6.10 Library

We have built a syntax theory for English syntax, and we plan to build others. These theories are available if you expand **Library**; they are arranged in a table. The `Search` input lets you restrict the table to lines containing all the (space-separated) strings you type in. Each table entry names a language and names the phenomena the theory explores. When you click its green button, the Syntacticon evaluates the associated

theory.

For pedagogical and testing purposes, some theories in the **Library** might be password-protected, in which case the theory name includes a lock symbol. The passwords are kept on the server.

6.11 IPA chart

The IPA chart is a valuable tool to allow you to see the features underlying every IPA symbol, to compare the features of several symbols, to select symbols based on features, and to choose restricted inventories.

The chart is divided into Vowels, Consonants, Other symbols (like ʍ), Affricates (like ʧ), and Doubly-articulated stops (like k͡p). The layout of the vowels and consonants is meant to reflect standard organization, separating characteristics such as tongue positions (such as `front` for vowels and `dental` for consonants).

When you click any entry in the chart, it is selected and acquires a colored background, and a light blue pop-out on the right side of the page shows its features. Click the entry again to unselect it. You can click the `Features` region of that pop-out to copy the list of features with polarity into your clipboard; you can then paste the list into a `Define` rule in the Phonology part of the **Inputs** regions.

If you click multiple entries in the IPA chart, the pop-out shows the features that distinguish those entries. For instance, if you click `t` and `d`, you see that `t` is `−voice`, whereas `d` is `+voice`. The list of features they share is also shown (and can be copied by a click).

You can select all IPA symbols by clicking Select all, and you can unselect all IPA symbols by clicking Clear selection.

If you have selected several IPA entries, you can click Make inventory at the top of the chart. IPA entries in the inventory are shown with blue-outlined cells, and a new or replacement `Inventory` line appears in the **Inputs** region. Once you have built an explicit inventory, selecting entries shows how those entries are distinctive within the inventory; that is, if they form a natural class. For instance, if the inventory contains only `p`, `b`, `t`, and `d`, then selecting `t` and `d` causes the blue pop-up to show not only that they differ with respect to `voice`, but that they have the distinction within the inventory of being `−labial`. Similarly, `p` and `t` are have the inventory-based distinction `−voice`. However, `p` and `d` do not form a natural class.

Below the IPA chart is a Select by features button, which leads to a pop-up menu listing every phonological feature. Each can be asserted to have polarity `+`, `−`, `0`, or `any`. Initially, all are set to `any`. You can assert any combination of features and then click Select (at the end of the list, which scrolls up and

down). For instance, if you select `+distributed −strident +sonorant`, you see that ɲ, ɻ, and ʎ are selected in the chart, and the associated blue pop-out frame shows their features.

6.12 Segments and features

The chart of segments and features is an alphabetical list containing a row for each segment and a column for each phonological feature. You can select segments here just as in the IPA chart; in fact, a selection in either is also displayed in the other and brings up the blue pop-out. The header showing the features also emboldens those features for which the selected segments differ.

7 Acknowledgements

The IPA charts are based on ones at `www.internationalphoneticalphabet.org/ipa-sounds/ipa-chart-with-sounds/`. The text-to-pronunciation tool is based on code at `ipa-reader.xyz/` by Katie Linero. The graphing tool for syntax patterns (`mshang.ca/syntree`) is by Miles Shang

The authors coded the Syntacticon in Perl (Wall et al., 2000). The code is available under a Creative Commons CC-BY license.

References

Bird, S. and Ladd, D. R. (1991). Presenting autosegmental phonology1, *Journal of Linguistics* **27**(1): 193–210.

Hammarström, H., Forkel, R., Haspelmath, M. and Bank, S. (eds) (2020). *Glottolog 4.2.1*, Max Planck Institute for the Science of Human History, Jena.
URL: *https://glottolog.org/*

Hayes, B. (2009). *Introductory Phonology*, Wiley-Blackwell, Hoboken, NJ.

Wall, L., Christiansen, T. and Orwant, J. (2000). *Programming Perl*, " O'Reilly Media, Inc.".

Index

autosegmental phonological rule
    `Associate`, 24
    `Join`, 25
    `Realize`, 25
    `Reassociate`, 25
    `Split`, 26
    `Spread`, 25
autosegmental phonology, 23
available rule (morphophonological rule whose constraints are met), 18

block name (of a rule block), 17
brace form (representation of a syntax tree), 5
bracket form (representation of a syntax tree), 5

change (component of a morphophonological rule), 18
component (of a morphophonological rule), 17
condition (component of a morphophonological rule), 18
content (of a syntax-tree head node), 4
context (constituent of a change, introduced by `/`), 18
custom phoneme (phoneme like `ð`, introduced by `Define`), 15

effective change (of a morphological rule application), 19

feature (characteristic of a segment, such as `syllabic`), 14
flags (on autosegmental phonological rules), 23
flags (on syntax rules), 8
form (the input and output of a morphophonological rule), 17

geometry cluster (set of features, such as `cor`), 16

head (kind of syntax-tree node), 4

inclusive rule block, 17
intermediate constituent (kind of a syntax-tree node), 4
IPA chart (region of the web page), 31

label (on a syntax-tree node), 9
lexeme unit (part of a surface expression, marked with √), 5
literate comments, 6

Metadata (region of the web page), 29
morphological unit (part of a surface expression), 4
Morphophonological rules (region of the web page), 17
morphosyntactic class (MC, set of morphosyntactic properties), 17
morphosyntactic property, 17
MP (morphological property), 4

name (of a syntax-tree node), 4

Pāṇini precedence (indication of the level of constraint of a morphophonological rule), 18
phonology command
    `Define`, 15
    `Delete`, 15
    `Disallow`, 15
    `Geometry`, 16
    `Inventory`, 14
    `Redefine`, 15
phrase (kind of a syntax-tree node), 4

Preamble (region of the web page), 14

replacement (constituent of a change), 18
Results (region of the web page), 28
rule block (section of a morphophonological rules), 17

segment (smallest fragment in the segmental tier, typically an IPA symbol), 14
Segments and features (region of the web page), 32
selective rule block, 17
surface expression (content of a head node), 4
syntax command
- `AdjoinLabel`, 9
- `Apply`, 13
- `CopyMPs`, 10
- `Crash`, 13
- `DIR-Adjoin`, 9
- `DeleteContent`, 10
- `DeleteNode`, 11
- `Expect`, 14
- `Flatten`, 13
- `MergeRoot`, 10
- `MergeUnit`, 10
- `MoveContent`, 11
- `MoveToSpecifier`, 12
- `Spellout`, 12

Syntax patterns (region of the web page), 4
Syntax rules (region of the web page), 6

target (constituent of a change, introduced by →) , 18

wild card (in a syntax pattern), 5