MULTIFACETED INTERACTIVE ANALYSIS OF LANGUAGES AND PROCESSES

Dakotah Lambert • 10 May 2025

An analysis of a language or process is tied to the analyst's perspective on which kinds of structures and connections underlie that language or process. Other perspectives yield other analyses. Some are logically equivalent, while others might only result in the same pattern over the supposed alphabet. We detail a small hierarchy of classes based on logical systems, and connect these logical concepts to mathematical tools. These tools and techniques are the foundation upon which the Language Toolkit is built. We discuss the use of this software and related technologies for multifaceted interactive analysis of languages and processes.

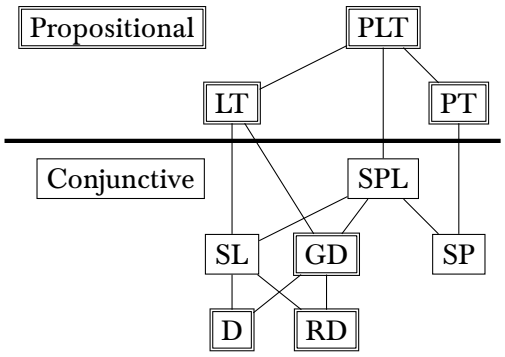

Figure 1: A sample of linguistically-relevant classes

1 Dependencies and Installation

Install GHCup (`https://www.haskell.org/ghcup/`), the main installer for the Haskell programming language, then install its components:

```
$ ghcup install ghc --set recommended
$ ghcup install cabal latest
$ cabal update && cabal install language-toolkit
```

For visualizations to work, be sure to also install GraphViz (`https://graphviz.org`) and a `display` program such as that provided by ImageMagick. Run `plebby` and type `:help` at the prompt for some introductory documentation.

2 Factor-Based Description Models for Phonotactics

The Piecewise-Local Expression Builder (PLEB) is a simple programming language defined by the Language Toolkit for describing and manipulating languages. Its interpreter, `plebby`, allows for interactive analysis. Generally I use Unicode syntax, but this handout prioritizes the equivalent ASCII syntax to ease following along.

2.1 Locality: Substrings Under Adjacency

Local dependencies are modeled by SUBSTRINGS: sequences that appear in order, adjacently. In PLEB, a substring is written as a sequence of symbols separated by **spaces**, surrounded by angle brackets: <s_1 s_2 ... s_n>. This expression represents the set of words containing the substring $s_1 s_2 \dots s_n$, which can be interpreted as the constraint "words must contain $s_1 s_2 \dots s_n$ as a substring".

The substring can be ANCHORED such that its first symbol aligns with the starting edge of the word, such that its final symbol aligns with the ending edge of the word, or both, using the `%|`, `|%`, and `%||%` modifiers, respectively. The modifiers are given as prefixes: given a symbol class "`stress`" that represents a stressed syllable, the expression `%|<stress>` represents words that begin with stress, `|%<stress>` represents words that end with stress, and `%||%<stress>` represents stressed monosyllables.

2.2 Long-Distance: Subsequences Under Precedence

One way to account for long-distance dependencies is to use SUBSEQUENCES: sequences that appear in order, but not necessarily adjacently. In PLEB, a subsequence is written as a sequence of symbols separated by **commas**, surrounded by angle brackets: <s_1,s_2,...,s_n>. The comma can be read as "an arbitrarily long gap". A GENERALIZED SUBSEQUENCE allows for arbitrary substrings in place of the symbols here; these can be specified with a mixture of spaces (where adjacency is required) and commas (where gaps are allowed).

2.3 Long-Distance: Tiers

Another way to handle long-distance dependencies is to apply a constraint after projection to a specific set of symbols (tier-based constraints). To this end, PLEB provides two prefix operators. First, [t_1,t_2,...,t_n]c specifies that constraint c applies to the projection to the symbols $\{t_1, t_2, \ldots, t_n\}$ — specifying the SALIENT symbols. For example, `[stress]%||%<stress>` can be read as "on the *stress* tier, there is one, and only one, (stressed) syllable". Alternatively, |t_1,t_2,...t_n|c specifies that constraint c applies to what remains after deleting $\{t_1, t_2, \ldots, t_n\}$ — specifying the NONSALIENT symbols.

2.4 Combinations

More complex constraints can be built from these factors in various ways. For example, the constraint that words contain one and only one stressed syllable could be described using tiers as "on the stress tier, there is one, and only one (stressed) syllable", written `[stress]%||%<stress>`, *or* it could be described using subsequences as "words contain a stressed syllable, but do not contain a stress... stress subsequence", written `/\{<stress>,!<stress,stress>}`.

In the latter case, we need to make use of BOOLEAN OPERATIONS. Some of the available operations include conjunction (intersection, AND: `/\{...}`), disjunction (union, OR: `\/{...}`), and negation (complement, NOT: `!...`).

3 Phonotactic Examples From Stress

The structural properties of a pattern are an emergent aspect of the system, not a product of the form of the constraints used to describe it. Using these base factor types and combinators, a pattern can be described in whatever way makes the most sense to the analyst, and then its properties can be investigated separately. Some classes that have been used in the study of linguistic phenomena include:

- Piecewise-locally testable: also called "dot-depth one", the acceptability of a word is decided by its set of generalized subsequences.
  - Locally testable: acceptability is decided by the set of length-k substrings
    - Strictly local: a word is accepted if all of its length-k substrings are acceptable
      - Definite: only right-anchored substrings are forbidden
      - Reverse definite: only left-anchored substrings are forbidden
    - Generalized definite: the prefix–suffix pair determines acceptability
  - Piecewise testable: acceptability is decided by the set of length-k subsequences
    - Strictly piecewise: a word is accepted if all of its length-k subsequences are acceptable
  - Strictly piecewise-local: a word is accepted if all of its subsequences up to length k of substrings up to length j are acceptable (contains strictly local and strictly piecewise)
- Tier-based and multitier extensions of these

3.1 Stress Final

Consider the stress-final pattern of Iban, which can be described as follows: words contain one and only one syllable with primary stress, and it is the final syllable. There are *many* equivalent ways to write this, each evoking a different classification:

```
> =stress {/L,/H} =unstress {/l,/h} =syl {stress,unstress} # set up symbols
> =stress-final-1 /\{!|%<unstress>,!<stress syl>} # strictly local
> =stress-final-2 /\{<stress>,!<stress,syl>} # piecewise testable
> =stress-final-3 /\{|%<stress>,[stress]%||%<stress>} # multitier definite
```

Each of these defines the same language, but each approaches the description from a different perspective. The first says "words do not end with unstressed syllables, and do not have any syllable (immediately) after a stressed one".[1] The second says "words contain a stressed syllable, and there is no syllable after it (at any distance)". The third says "words end with stressed syllables, and, on the stress tier, words contain only one (stressed) syllable". We can ask `plebby` to verify their equivalence, and to test the pattern's membership in a few classes:

```
> :equal stress-final-1 stress-final-2
True
> :equal stress-final-1 stress-final-3
True
> :isSL stress-final-1
True: k=2
> :isPT stress-final-1
True
> :isMTDef stress-final-1
True
> :isSP stress-final-1
False
```

3.2 Stress Rightmost Heavy Else Rightmost

A similar pattern that introduces more long-distance dependencies is the stress pattern of Golin, in which the final *heavy* syllable is stressed, if there is such a syllable, else the final (light) syllable is stressed. This can be written in a way that makes clear that it is multitier definite, but one can also verify that it is also piecewise testable (like stress-final) but *not* strictly local. The pattern: there is exactly one stressed syllable, and *either* the last heavy syllable is stressed *or* there are no heavy syllables and the last (light) syllable is stressed.

```
> =L /L =H /H =l /l =h /h =syl {l,h,L,H}
> =stress {L,H} =unstress {l,h} =light {l,L} =heavy {h,H}
> =golin /\{[stress]%||%<stress>,\/{[heavy]|%<H>,/\{[heavy]%||%<>,|%<L>}}}
> :isMTDef golin
True
> :isPT golin
True
> :isSL golin
False
```

As an exercise, consider how the expression might look if written from a piecewise testable perspective.

[1] The ability to specify "any symbol" is planned for a future release.

4 How Does This Work?

Algebra is the study of structure. The nice logical properties of the classes in Figure 1 translate to nice algebraic properties. All of the classes there are associated with a set of equations, where a language is in the class if and only if its associated algebraic structure satisfies those equations for every possible instantiation of its variables.

The algebraic structure is obtained as follows. Declare two strings, a and b, to be DISTINCT if there is a context $u\_\_v$, such that one of uav or ubv is accepted and the other is rejected. If there is no such distinguishing context, the strings are equivalent. Equivalence-classes of strings correspond to state-to-state functions on the canonical finite-state automaton; so begin with the functions representing the input alphabet and compose them to completion.

A brief description of the equational characterizations is as follows. The equation $x = y$ *means* that in all contexts $u\_\_v$ we have uxv and uyv are either both accepted or both rejected. And x^ω can be thought of as representing "a sufficiently long word". Therefore it is reasonable that $sx^\omega = x^\omega$ characterizes the DEFINITE patterns: given a sufficiently long suffix, preceding material is irrelevant.

Some of the classes use alternative algorithms to improve speed. But the ability to test arbitrary equations is exposed for those who are interested in exploring other classes beyond those with prepared decision algorithms. In the next release of the software, the corresponding equations will be provided in the documentation for every `:is`*`Class`* command.

5 Processes and Rational Functions

The same techniques apply to classifying string-to-string functions represented by finite-state transducers. Extract the transition semigroup or monoid, then evaluate the equations for the desired classes. This kind of analysis yields an input-oriented perspective: "Based on these properties of the input seen so far, the function will do this to the next input symbol."

While PLEB offers a way to define and manipulate languages, it has no such features for transductions. However, a separate program, `classify`, is provided to read and analyze transducers such as those created by OpenFST. Given a transducer in canonical form stored in `transducer.att`, the following will output a summary of whether the process is (structurally) definite, tier-based reverse definite, or piecewise testable.

```
$ classify -N Def TRDef PT <transducer.att
```

Further Reading

[1] Jeffrey Heinz. Culminativity times harmony equals unbounded stress. In Harry van der Hulst, editor, *Word Stress: Theoretical and Typological Issues*, chapter 8. Cambridge University Press, Cambridge, UK, 2014.

[2] Dakotah Lambert. Relativized adjacency. *Journal of Logic, Language and Information*, 32(4):707–731, October 2023.

[3] Dakotah Lambert. System description: A theorem-prover for subregular systems: The Language Toolkit and its interpreter, plebby. In *Functional and Logic Programming: 17th Annual Symposium, FLOPS 2024*, pages 311–328, May 2024.

[4] Dakotah Lambert. Multitier phonotactics with logic and algebra. *Phonology*, in press.

[5] Dakotah Lambert and Jeffrey Heinz. An algebraic characterization of total input strictly local functions. In *Proceedings of the Society for Computation in Linguistics*, volume 6, pages 25–34, Amherst, Massachusetts, June 2023.

[6] Dakotah Lambert and Jeffrey Heinz. Algebraic reanalysis of phonological processes described as output-oriented. In *Proceedings of the Society for Computation in Linguistics*, volume 7, pages 129–138, Irvine, California, June 2024.

[7] James Rogers, Jeffrey Heinz, Margaret Fero, Jeremy Hurst, Dakotah Lambert, and Sean Wibel. Cognitive and sub-regular complexity. In *Formal Grammar 2012 and 2013*, pages 90–108. 2013.