Computational Restrictions on Iterative Prosodic Processes

Hossep Dolatian[1], Nate Koser[2], Jonathan Rawski[1], Kristina Strother-Garcia[3]
[1]*Stony Brook University,* [2]*Rutgers University,* [3]*Bloomsburg University of Pennsylvania*

1 Introduction

This paper explores a computational characterization of iterative phonological processes including stress, epenthesis, and syllabification. Understanding the computational requirements of these iterative mappings leads to restrictive, testable, and learnable theories of phonology (Heinz, 2018). Previous work in this vein has been done on tone (Koser et al., 2019), and we expand the results to other kinds of processes. We show that these iterative prosodic processes are fundamentally local, thus fitting the typology of other computational work (Chandlee & Heinz, 2018). However, they require reference to local information in the *output*, rather than just the input. We formulate this output-centeredness via logical transductions (Courcelle, 1997), where an output element receives its output label or output property based on some requirement defined over the input.

Additionally, the iterative nature of the processes requires a notion of recursion, formalized in a logical transduction via *least fixed point logic* (LFP; Libkin, 2013) which we require to be *quantifier free* (QF; Chandlee & Lindell, in prep). The restriction to QF (i.e. no reference to logical quantifiers $\exists$ or $\forall$) ensures that the notion of locality in the output is preserved, while still characterizing the relevant iterative processes (Chandlee & Jardine, 2019). Rather than introduce the full LFP formalism here, we employ the *implicit definitions* of Rogers (1996), which allow for output predicates to reference themselves as part of their definition. This provides a more intuitive characterization of the recursion necessary to define the target phonological processes. Finally, a substantive restriction of our transduction to the use of only the predecessor *or* successor function – not both – further constrains the computational power of these transductions to better fit the observed typology of iterative phonological patterns. Informally, this means that we these iterative prosodic processes apply in a single direction: right-to-left, or left-to-right, but not bidirectional.

2 Logical structure of non-iterative prosody

In this section, we explain computational locality and our formal notation. We summarize formal results from Strother-Garcia (2019) on the computational locality of non-iterative syllabification. We use these simple examples to illustrate our formal notation.

Strother-Garcia (2019) demonstrates that syllabification processes are fundamentally local in a strict mathematical sense in a number of languages. For example, consider a hypothetical language which has maximally CVC syllables (Blevins, 1995). Given a string /patukpi/, the string is syllabified as [pa.tuk.pi] based on the following simple description:

* We thank the Facebook Messenger team for enabling the entirety of this project from start to finish during the 2020 pandemic.

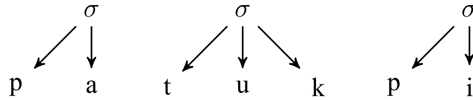
© 2021 Hossep Dolatian, Nate Koser, Jonathan Rawski, Kristina Strother-Garcia
Proceedings of *AMP 2020*

(1) *Simple syllabification in a CVC language*

1. vowels project a syllable,
2. vowels form the nucleus of a syllable,
3. pre-vocalic consonants are onsets, and
4. post-vocalic consonants are codas as long as they're not onsets.

We illustrate the syllable structure for $[\text{pa.tuk.pa}]$ below.

(2) *Tree-based syllable structure of* $[\text{pa.tuk.pi}]$



Computationally, there are at least two ways to generate the above structure. One is to faithfully generate tree structures over the segments (2). The other is to generate a set of 'structural roles' for each segment as a type of property of each segment (3), i.e., to give intervocalic /t/ the property of being [+onset]. The first approach is easier to model with tree-based context-free grammars (Coleman & Pierrehumbert, 1997; Coleman, 1998, 2000), while the second approach is easier for string-based finite-state transducers (Kiraz & Möbius, 1998; Yap, 2006; Hulden, 2006). Analogous monostratal approaches to syllabification have also been used earlier in Declarative Phonology (Walther, 1993, 1995; Bird, 1995; Coleman, 1996, 1998).

(3) *string-based syllable structure of* $[\text{pa.tuk.pi}]$

p_{ons} a_{nuc} t_{ons} u_{nuc} k_{cod} p_{ons} i_{cod}

In terms of generative capacity, both approaches are however equivalent because syllabification uses syllables of a bounded size (Strother-Garcia, 2019). In this paper, we use logical transducers which can work with either of the two approaches. Logical transductions are a generalized form of computation that operates over graphical structures and by using formal logic (Courcelle, 1994, 1997; Engelfriet & Hoogeboom, 2001). They have been used to model non-iterative syllabification (Strother-Garcia, 2018) and the generation of higher prosodic structure (Dolatian, 2020).

In this paper, we focus on formalizing the string-based structure for illustration. For a logical transduction, the representation of a word consists of 3 main components: a domain D of input elements, a set L of labels for these elements, and a set R of binary relations over these elements. The domain is represented as a set of indexes [1...n] where n is the size of the word. For our purposes, the labels L consist of phonological features or segments. The binary relations R are only the immediate successor relation $\mathsf{succ}(x, y)$ which connects any two consecutive segments. This binary relation $\mathsf{succ}(x, y)$ can be broken up to two functions $\mathsf{succ}(x)$ and $\mathsf{pred}(x)$ which return the immediate successor and immediate predecessor of an element x. For example, below is the representation of the unsyllabified /patukpi/.

(4) *Input representation* /patukpi/ *for logic-based syllabification*

$$\text{p}_1 \xrightarrow[\triangleleft]{} \text{a}_2 \xrightarrow[\triangleleft]{} \text{t}_3 \xrightarrow[\triangleleft]{} \text{u}_4 \xrightarrow[\triangleleft]{} \text{k}_5 \xrightarrow[\triangleleft]{} \text{p}_6 \xrightarrow[\triangleleft]{} \text{i}_7$$

Each domain element has a subscript that shows its index from the range 1-7. These elements are connected via the binary relation of successor, shown as $\triangleleft$-labeled edges. Each element x satisfies a set of labels, such as $+\mathsf{labial}(x)$ or $+\mathsf{vowel}(x)$. For illustration, our graphs show the main segment label for each element.

(5) *Output representation* $[\text{pa.tuk.pi}]$ *for logic-based syllabification*

$$\text{p}_{1'ons} \xrightarrow[\triangleleft]{} \text{a}_{2'nuc} \xrightarrow[\triangleleft]{} \text{t}_{3'ons} \xrightarrow[\triangleleft]{} \text{u}_{4'nuc} \xrightarrow[\triangleleft]{} \text{k}_{5'cod} \xrightarrow[\triangleleft]{} \text{p}_{6'ons} \xrightarrow[\triangleleft]{} \text{i}_{7'nuc}$$

The above output representation adds the apostrophe ' after an index in order to mark it as an output correspondent. The newly generated syllable role is visualized as a simple subscript. To actually generate the above form, we need a logical transduction that uses the following output functions. These function establish the correspondence relations between the input and output representations, and they determine the properties of the output segments.

(6) *Output functions for simple CVC-syllabification*

a. $\texttt{nuc}'(x) \stackrel{\text{def}}{=} \mathsf{V}(x)$

b. $\texttt{ons}'(x) \stackrel{\text{def}}{=} \mathsf{cons}(x) \wedge \mathsf{V}(\mathsf{pred}(x))$

c. $\texttt{cod}'(x) \stackrel{\text{def}}{=} \mathsf{cons}(x) \wedge \mathsf{V}(\mathsf{succ}(x)) \wedge \neg \texttt{ons'}(x)$

The above output functions capture the essential facts of simple syllabification. In the output, the input segment /u/ gains the property of being a nucleus $\texttt{nuc'}(x)$ because it is an underlying vowel. The segment /t/ surfaces as an onset $\mathsf{ons}'(x)$ because it is a consonant which precedes a vowel. And the segment /k/ surfaces as a coda $\texttt{cod}'(x)$ because it is a post-vocalic consonant which isn't parsed as an onset. Note how the coda function references the onset function.[1]

Note that the above transduction doesn't add any additional segments to the output. To do that, we need to incorporate a copy set which keeps track of how many additional output segments are generated by a transduction, such as for epenthesis. We postpone copy sets till later.[2]

Simple CVC syllabification demonstrates a strict formal property of computational locality. There are no long-distance dependencies involved in projecting a segment as a nucleus, vowel, or coda. To identify onsets and codas, it suffices to compare each consonant to its predecessor and to its successor. If we needed to reference a distant trigger such the word-final segment, then we would need quantifiers like $\exists$. In fact, local logical transductions have the property of being quantifier-free. They depend only on local information in the input string which can be accessed with functions like $\mathsf{succ}(x)$ or $\mathsf{pred}(x)$. QF functions correspond to *input strictly local* functions (ISL; Chandlee, 2014) when computed over FSTs. This contrasts with constraint-interaction accounts like OT (Prince & Smolensky, 2004) where the entire word must be considered (*global* evaluation), obfuscating the fact that the process is local.

This completes this section. The main takeaway is that simple non-iterative syllabification is a local process. It can be formalized with simple logical transductions that maintain this locality.

3 Iteration and directionality in segmental phonology

The generation of non-iterative syllabification is relatively straightforward. The main factor is that these processes don't reference long-distance information over the input or output. However, there are cases of directional syllabification which are long-distant over the input. However, they are local over the output because they are iterative.

Iterative prosodic processes are closely tied with iterative segmental processes. Both types of processes show directionality effects. Directionality is a significant aspect of the computation of phonology (Johnson, 1972). Before we formalize directional prosody, we first go over the directional computation of iterative segmental phonology.

Vowel harmony is a canonical example of iteration and directionality effects in phonology. In general, vowel harmony can be progressive, regressive, or bidirectional. We use toy examples below where the harmonic feature is ATR. For the bidirectional pattern, we assume that harmony spreads from the roots -tu-,-tʊ-.

(7) *Toy cases of progressive, regressive, and bidirectional ATR harmony*

a. Progressive: /pitʊkɔ/ → [pituko], /pɪtuko/ → [pɪtʊkɔ]

b. Regressive: /pɪtʊko/ → [pituko], /pitʊkɔ/ → [pɪtʊkɔ]

c. Bidrectional: /pɪ-tu-kɔ/ → [pituko], /pitʊko/ → [pɪtʊkɔ]

Each directionality parameter has its own computational analog (Gainor et al., 2012; Heinz & Lai, 2013; Heinz, 2018). Progressive harmony requires a left-subsequential function (Mohri, 1997) which memorizes the fact that the first vowel it saw was [+ATR]. It then carries this information across the word from left to right.

[1] Although the coda function references the onset function, this does not mean that logical functions are serially satisfied. They are all satisfied in parallel in a monotonic form, much like Declarative Phonology (Scobbie et al., 1996).

[2] For simple syllabification, no epenthesis is involved so the copy set is set to $\{1\}$, meaning that only 1 set of output elements (7 nodes) are generated. If we wanted to generate syllable trees, we would need an additional copy in our copy set (={1,2}) in order to generate the 3 additional syllable nodes.

This function is computed over a left-to-right deterministic finite-state transducer (FST). Similarly, regressive harmony is a right-subsequential function that processes the string from right to left; and it is computed over a right-to-left FST. Bidirectional harmony is strictly more expressive because it utilizes multiple directions. As a function, it is weakly deterministic and requires a non-deterministic FST.

When examined over the input, the information transferred by iterative processes is long-distance. In the case of progressive vowel harmony, information is transferred from the first vowel to the last. It is likewise long-distance over the input because any number of consonants can intervene between any two vowels.[3]

However, it is possible to have an iterative rule which is long-distance over the input, but local over the input. This nuance is clearer with segment-to-segment iterative rules (Howard, 1972). Consider a toy example of nasal spread. Nasality spread from a nasal segment onto a contiguous sequence of vowels and glides.

(8) *Toy examples of iterative nasal spread over vowels and glides*
/panata/ → [panũta]
/panawuta/ → [panãw̃ũta]
/panawajuta/ → [panãw̃ãj̃ũta]

Over the input, nasality spread is a long-distance process. Nasality spreads from the nasal /n/ to the vowel /u/ regardless of how many segments intervene between them. However, the property of being nasalized is locally predictable over the output. A non-consonantal segment (vowel or glide) is nasalized if it follows another nasalized segment. In theoretical phonology, this is the basis for iterative rules. In mathematical phonology, iterative segmental processes are formalized as Output-Strictly-Local functions (Chandlee et al., 2015). Informally, the output correspondent of a symbol /u/ is [ũ] if it follows another nasalized segment in the output [...Nũ]. Left-to-right iterative processes require left-OSL functions, i.e., an OSL function which reads and processes the input left-to-right.

We can capture the above output-local rule with the following logical transduction. Informally, a segment is nasalized if it is 1) underlyingly a nasal, or 2) a non-consonantal segment such that 3) its predecessor is a nasalized segment over the output.

(9) *Recursive output function to generate nasal spread*
$$\texttt{+nasal}'(x) \stackrel{\text{def}}{=} +\mathsf{nasal}(x) \vee [-\mathsf{cons}(x) \wedge \texttt{+nasal}'(\mathsf{pred}(x))]$$

Crucially, to nasalize some segment x, the above function references the *output* properties `+nasal'` of another output segment which precedes the segment x: $\mathsf{pred}(x)$. Thus, the nasalization function *recursively* references itself in order to spread nasality across a span of contiguous non-consonantal segments. The above function allows all underlyingly nasal segments $+\mathsf{nasal}(x)$ to faithfully surface as nasal $\texttt{+nasal}'(x)$. For any non-consonantal segment x, we check if its predecessor is nasalized. That means we check if the predecessor is a nasal (in the input) or is a non-consonant that precedes a nasalized segment (in the output). The recursive call continues until we hit a non-nasal or a consonant. We illustrate the calls below.

(10) *Recursive computation of nasal spread in* /panawuta/→[panãw̃ũta]

Input:	*p*	*a*	*n*	*a*	*w*	*u*	*t*	*a*
Index:	1	2	3	4	5	6	7	8
Predecessor $\mathsf{pred}(x)$:		1	2	3	4	5	6	7
Underlying $+\mathsf{nasal}(x)$:			⊤					
Underlying $-\mathsf{cons}(x)$:		⊤		⊤	⊤	⊤		⊤
Surface `+nasal'`(x):								
Iteration 1:			⊤					
Iteration 2:			⊤	⊤				
Iteration 3:			⊤	⊤	⊤			
Iteration 4:			⊤	⊤	⊤	⊤		
Output	*p*	*a*	*n*	*ã*	*w̃*	*ũ*	*t*	*a*

[3] Because vowel harmony ignores consonants, it is possible to analyze vowel harmony as a case of relativized adjacency over the input or output, as is commonly argued in Search-and-Copy approaches to phonology (Nevins, 2010; Samuels, 2011). Computationally, this amounts to calling vowel harmony a tier-based local system, whether over the input or output (Aksënova & Deshmukh, 2018; Andersson et al., 2020; Burness & McMullin, 2020).

The first rows show all the relevant input information. $\top$ marks a property as true for some indexed segment. To generate the output `+nasal'`(x) feature, the function operates recursively. It finds all underlying nasals in iteration 1, and it propagates this feature rightwards to any non-consonantal segments that succeed the surface nasalized segments.

The above recursive function (9) is called an *implicit definition* (Rogers, 1998) because we don't explicitly encode when the recursive call would end.[4] Instead, we can deduce that the recursive call will end once there are no more underlying non-consonantal segments which follow a surface nasalized segment. The nasalization function is local over the output because we examine only the linearly *adjacent* predecessor.

4 Directional prosody and logic

The previous section introduced recursive logic as a way to encode iterative and directional processes in segmental phonology. In this section, we use that system to also formalize iterative prosodic processes.

4.1 *Iterative tress* Recursive logic allows for intuitive definitions of iterative stress assignment. For example, Murinbata (Street & Mollinjin, 1981) applies stress to every other syllable beginning with the initial syllable:

(11) *Examples of iterative left-to-right trochaic secondary stress*
$\acute{\sigma}, \acute{\sigma}\sigma, \acute{\sigma}\sigma\grave{\sigma}, \acute{\sigma}\sigma\grave{\sigma}\sigma, \acute{\sigma}\sigma\grave{\sigma}\sigma\grave{\sigma}, \acute{\sigma}\sigma\grave{\sigma}\sigma\grave{\sigma}\sigma, \ldots$

For illustration, we formalize iterative stress over an input string of syllables. An equivalent transduction can be defined over simple segments. For the above process, primary stress is placed on the first syllable. A syllable is first if it has no predecessor.[5] Secondary stress is placed via a recursive output function. Secondary stress is placed 2 syllables away from the initial primary stressed syllable, and 2 syllables away from every secondary-stressed syllable. The superscript 2 is shorthand for $\mathsf{pred}(\mathsf{pred}(x))$.

(12) *Output function to generate iterative secondary stress*

a. $\acute{\sigma}'(x) \stackrel{\text{def}}{=} \sigma(x) \wedge \mathsf{pred}(x) = \textsc{null}$

b. $\grave{\sigma}'(x) \stackrel{\text{def}}{=} \acute{\sigma}'(\mathsf{pred}^2(x)) \vee \grave{\sigma}'(\mathsf{pred}^2(x))$

We illustrate for the following 10-syllable word. The recursive call goes through at most 4 iterations for this word. The first syllable receives primary stress. Secondary stresses are recursively placed on those syllables that have a stressed syllable that's two slots to the left in the output string.

(13) *Recursive computation of iterative secondary stress:* /σσσσσσσσσσ/→[σ́σσ̀σσ̀σσ̀σσ̀]

Input:	σ	σ	σ	σ	σ	σ	σ	σ	σ	σ
Index:	1	2	3	4	5	6	7	8	9	10
Predecessor $\mathsf{pred}(x)$:		1	2	3	4	5	6	7	8	9
Underlying $\sigma(x)$:	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
Surface $\acute{\sigma}'(x)$:	$\top$									
Surface $\grave{\sigma}'(x)$:										
Iteration 1:			$\top$							
Iteration 2:			$\top$		$\top$					
Iteration 3:			$\top$		$\top$		$\top$			
Iteration 4:			$\top$		$\top$		$\top$		$\top$	
Output	$\acute{\sigma}$	σ	$\grave{\sigma}$	σ	$\grave{\sigma}$	σ	$\grave{\sigma}$	σ	$\grave{\sigma}$	

[4] To turn this into an explicit statement, we would need to incorporate the logical device of least-fixed-point operations (Libkin, 2013). These have been used in previous work on directional effects in phonology (Koser et al., 2019; Chandlee & Jardine, 2019).

[5] If we used quantifiers, then $\acute{\sigma}'(x)$ would use the statement $\neg\exists y[\mathsf{succ}(y, x)]$ instead of $\mathsf{pred}(x) = \textsc{null}$. But without quantifiers, we need to either introduce a 'sink state' symbol NULL, or say that initial segments precede themselves $\mathsf{pred}(x) = x$.

4.2 *Syllabification epenthesis* This section formalizes the most complex phenomenon in this paper: iterative epenthesis. Iterative and directional syllabification is often motivated from cases of directional epenthesis (Itô, 1989) whereby syllables are parsed either left-to-right or right-to-left, and vowel epenthesis mimics this direction.

A classic case for iteration comes from how different Arabic dialects choose different epenthesis sites for consonant clusters. Consider the examples below from Cairene and Iraqi Arabic. In both dialects, the maximal syllable is CVC inside words. Schwa epenthesis is used to repair unsyllabifiable consonant clusters, but the location of the epenthetic vowel varies by dialect. The input string /katab+t+l+u/ has a cluster of 3 consonants: VCCCV. This cluster undergoes epenthesis in both dialects but the vowel is added in different locations: [ka.tab.ti.lu] in Cairene and [ka.ta.bit.lu] in Iraqi. In contrast, the input string /katab+t+l+ha/ has a cluster of 4 consonants: VCCCCV. Both dialects apply epenthesis and in the same location: [ka.tab.til.ha].

(14) *Different epenthesis locations in Arabic*

Input :	/katab+t+l+u/	/katab+t+l+ha/
Cairene:	[ka.tab.ti.lu]	[ka.tab.til.ha]
Iraqi:	[ka.ta.bit.lu]	[ka.tab.til.ha]
	'I wrote to him'	'I wrote to her'

There are various analyses for the Arabic data, some use directional syllabification (Itô, 1986, 1989; Farwaneh, 1995) and some do not (Broselow, 1992, 2017; Kiparsky, 2003). We use the Arabic examples as a simple illustrative case study for iterative syllabification. Itô (1989) reduces the dialectal difference to a difference in the direction of syllabification. Syllables are formed left-to-right in Cairene, while right-to-left in Iraqi. This is illustrated in (15). For /katab+t+l+u/, the input is underlyingly unsyllabified: <katabtlu> where the brackets <> mark unsyllabified segments. In Cairene, the input is syllabified by scanning left-to-right, fitting as many segments into a CVC syllable, and resyllabifying pre-vocalic codas into onsets. The first iteration forms .kat.<abtlu>. The second iteration forms a second syllable which resyllabifies the first one's coda: .ka.tab.<tlu>. In the third iteration, the cluster *tl* is read. There is no intervening vowel so a vowel is epenthesized: .ka.tab.til.<u>. Finally, the final vowel is parsed: [.ka.tab.ti.lu.]. In Iraqi, the same input string is syllabified right-to-left, fitting as many segments into a CVC syllable, but onsets cannot be resyllabified into codas. This correctly generates [.ka.ta.bit.lu]. Furthermore for medial 4-consonant clusters in /katab+t+l+ha/, both left-to-right and right-to-left parses give the same output [.ka.tab.til.ha.].

(15) *Directional syllabification and epenthesis in Arabic*

Input	Left-to-right parse in Cairene /katab+t+l+u/ <katabtlu>	/katab+t+l+ha/ <katabtlha>	Right-to-left parse in Iraqi /katab+t+l+u/ <katabtlu>	/katab+t+l+ha/ <katabtlha>
First iteration	.kat.<abtlu>	.kat.<abtlha>	<katabt>.lu.	<katabtl>.ha.
Second iteration	.ka.tab.<tlu>	.ka.tab.<tlha>	<kata>.bit.lu.	<katab>.til.ha.
Third iteration	.ka.tab.til.<u>	.ka.tab.til.<ha>	<ka>.ta.bit.lu.	<ka>.tab.til.ha.
Fourth iteration	.ka.tab.ti.lu.	.ka.tab.til.ha.	ka.ta.bit.lu.	ka.tab.til.ha.

We formalize the directional analysis with recursive logic. We formalize the two directional parses by using separate recursive functions. For Iraqi R-to-L parsing, the recursive function computes the prosodic positions from right to left, while Cairene L-to-R uses the reverse direction. The end-result is that the formalization shows output-based locality. We focus on the directional syllabification of word-medial clusters, since consonants at word-edges tend to display idiosyncratic syllabification (Côté, 2000; Broselow, 2017). Those idiosyncrasies do not affect the computation significantly. They merely require adding simple special cases (which are also computationally local).

We first formalize R-to-L parsing because it is simpler to illustrate. As we go through our explanation, we expand our logical functions. For easier explanation, we separate the above processes into two separate sets of function. The first set of functions will parse the input string and assign the right syllable roles. The second set of functions will epenthesize the vowels. Informally, the first stage creates syllables and defective syllables (without a nucleus head), while the second stage changes these defective syllables into headed syllables.[6]

[6] In the poster, we defined iterative epenthesis in terms of two output functions that recursively generate the left and right

(16) *Decomposing directional syllabification and epenthesis into separate stages*

	Cairene L-to-R	Iraqi R-to-L	Both L-to-R and R-to-L
Input :	/katab-t-l-u/	/katab-t-l-u/	/katab-t-l-ha/
Stage 1: Syllabification:	ka.tab.t.lu	ka.ta.bt.lu	ka.tab.tl.ha
Stage 2: Epenthesis:	[ka.tab.<u>ti</u>.lu]	[ka.ta.<u>bit</u>.lu]	[ka.tab.<u>til</u>.ha]

The above 2-stage decomposition is only for easier illustration. The composition (flattening) of these stages into one function or step does not affect generative capacity. We emphasize that the computational treatment characterizes *any* implementation of directional syllabification, whether with rules (Itô, 1986; Frampton, 2011), alignment constraints (Mester & Padgett, 1994), or directional constraints (Eisner, 2000).

4.3 *Right-to-left syllabification* In Iraqi R-to-L syllabification, syllables are recursively created by parsing the string right-to-left. When comparing the unsyllabified /katab-t-l-u/ and syllabified [.ka.ta.bt.lu], one generalization is that that underlying vowels always surface as nuclei. Second, all underlying consonants and vowels faithfully surface.

(17) *Output functions to transparently generate syllable roles in R-to-L parsing*

a. $\texttt{nuc'}(x) \stackrel{\text{def}}{=} \mathsf{V}(x)$

b. $\texttt{V'}(x) \stackrel{\text{def}}{=} \mathsf{V}(x)$

c. $\texttt{C'}(x) \stackrel{\text{def}}{=} \mathsf{cons}(x)$

When parsing consonant clusters in a right-to-left direction, the left-edge of syllable marks when a new syllable can be projected. Word-medially, this left-edge is always an onset. Thus, to recursively generate left-boundaries, we recursively generate onsets by examining previously outputted onsets. In this way, onsets act as the *anchor point* for directional syllabification. Onsets are created in the following *finite* list of contexts. The double slashes // mark intermediate representations before epenthesis.

1. *Contexts for generating word-medial onsets in R-to-L directional syllabification*

   1. /...<u>C</u>V.../: before a vowel
   2. //...<u>C</u>C(C...//: before a consonant that's before an onset

The above two contexts set up the following recursive function:

(18) *Recursive output function to generate onsets in R-to-L directional syllabification*

$$\texttt{ons'}(x) \stackrel{\text{def}}{=} \mathsf{cons}(x) \wedge [\mathsf{V}(\mathsf{succ}(x)) \vee [\mathsf{cons}(\mathsf{succ}(x)) \wedge \texttt{ons'}(\mathsf{succ}^2(x))]]$$

For codas, they surface in the following contexts:

(19) *Contexts for generating codas in R-to-L directional syllabification*

1. /...<u>C</u>#/: word-finally
2. //...<u>C</u>(C...//: before onsets

These contexts set up the following function. We assume that onsets are blocked from becoming codas.

(20) *Output function to generate codas in R-to-L directional syllabification*

$$\texttt{cod'}(x) \stackrel{\text{def}}{=} \mathsf{cons}(x) \wedge \neg\texttt{ons}'(x) \wedge [\mathsf{succ}(x) = \textsc{null} \vee \texttt{ons'}(\mathsf{succ}(x))]$$

We show the output of these functions below. The onset function recursively applies from right to left.

edges of syllables $L'(x)$ and $R'(x)$ before resyllabification. Resyllabification is only apparent in left-to-right parsing. That analysis works, but it is not output-local because the property of being left-edged or right-edged is not a visible feature in the output string.

(21) *Recursive computation of R-to-L directional syllabification*

Input	*k*	*a*	*t*	*a*	*b*	*t*	*l*	*u*	*k*	*a*	*t*	*a*	*b*	*t*	*l*	*h*	*a*
Index:	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9
$\mathsf{pred}(x)$:		1	2	3	4	5	6	7		1	2	3	4	5	6	7	8
$\mathsf{succ}(x)$:	2	3	4	5	6	7	8		2	3	4	5	6	7	8	9	
Input $\mathsf{V}(x)$		$\top$		$\top$				$\top$		$\top$		$\top$					$\top$
Input $\mathsf{cons}(x)$	$\top$		$\top$		$\top$	$\top$	$\top$		$\top$		$\top$		$\top$	$\top$	$\top$	$\top$	
Output $\texttt{nuc'}(x)$:		$\top$		$\top$				$\top$		$\top$		$\top$					$\top$
Output $\texttt{ons'}(x)$																	
Iteration 1:	$\top$		$\top$				$\top$		$\top$		$\top$					$\top$	
Iteration 2:	$\top$		$\top$		$\top$		$\top$		$\top$		$\top$			$\top$		$\top$	
Output $\texttt{cod'}(x)$:						$\top$							$\top$		$\top$		
Output:	(*k*	*a*.)	(*t*	*a*.)	(*b*	*t*.)	(*l*	*u*.)	(*k*	*a*.)	(*t*	*a*	*b*.)	(*t*	*l*.)	(*h*	*a*.)
	O	N	O	N	O	C	O	N	O	N	O	N	C	O	C	O	N

The above functions correctly generate //.ka.ta.bt.lu// where *bt* forms a defective syllable without a nucleus. A vowel is epenthesized in a separate transduction. This transduction uses a copy set of size 2: $C = \{1, 2\}$. The first copy outputs all segments and syllable roles faithfully (22a), while the second copy will epenthesize a high vowel *i* (22b) as the output correspondent of an onset that does not precede a nucleus.

(22) *Formalizing epenthesis inside defective syllables*

a. *Output function to faithfully output syllabified segments*

$\forall \mathsf{label} \in L : \texttt{label'}(x^1) \overset{\text{def}}{=} \mathsf{label}(x)$

b. *Output function to epenthesize a vowel in defective syllables*

$\texttt{i'}(x^2) \overset{\text{def}}{=} \mathsf{ons}(x) \land \neg\mathsf{nuc}(\mathsf{succ}(x))$

c. *Computation of epenthesis after R-to-L directional syllabification*

Input	*k*	*a*	*t*	*a*	*b*	*t*	*l*	*u*	*k*	*a*	*t*	*a*	*b*	*t*	*l*	*h*	*a*
Index:	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9
Output $\texttt{nuc'}(x)$:		$\top$		$\top$				$\top$		$\top$		$\top$					$\top$
Output $\texttt{ons'}(x)$	$\top$		$\top$		$\top$		$\top$		$\top$		$\top$			$\top$		$\top$	
Epenthesized $\texttt{i'}(x^2)$:					$\top$									$\top$			
Output:	*k*	*a*	*t*	*a*	*bi*	*t*	*l*	*u*	*k*	*a*	*t*	*a*	*b*	*ti*	*l*	*h*	*u*

Thus, right-to-left syllabification is an output-local process within medial clusters.

4.4 *Left-to-right syllabification* Left-to-right syllabification is slightly more complicated to describe than right-to-left syllabification. This is because left-to-right syllabification displays resyllabification effects, whereby some consonants are parsed as a coda in an earlier stage of the derivation but later become onsets. We illustrate below for Cairene /katab-t-l-u/.

(23) *Resyllabification effects in left-to-right directional syllabification*

Cairene	<katab-t-l-u>	.kat.<abtlu>	.ka.tab.<tlu>	.ka.tab.til.<u>	.ka.tab.ti.lu
(L-to-R)	<katab-t-l-ha>	.kat.<abtlha>	.ka.tab.<tlha>	.ka.tab.til.<ha>	.ka.tab.til.ha

For the left-to-right parse in Cairene [.ka.tab.til.u.], the underlined segments form the right-edge of a syllable at some step of the derivation: <ka$\underline{\text{t}}$a$\underline{\text{b}}$t$\underline{\text{lu}}$>. The underlined segments are mostly codas that are resyllabified as onsets; the final vowel is a simple nucleus.

For right-to-left parsing, we treated onsets as the anchor point to recursively calculate new syllables. For left-to-right parsing, the analogous anchor point is codas. But special care has to made to ensure output-locality, i.e., to avoid resyllabification. As with right-to-left syllabification, all underlying vowels surface as nuclei.

(24) *Output function to transparently generate nuclei in L-to-R directional syllabification*

$\texttt{nuc'}(x) \overset{\text{def}}{=} \mathsf{V}(x)$

To parse medial clusters, we have to define how surface codas are recursively generated. Codas are formed in the following finite list of contexts:

(25) *Contexts for generating codas in L-to-R directional syllabification*

1. /...C#/: word-finally
2. /...VCC.../: between a vowel and a consonant
3. //...C)CCC... //: after a coda, consonant, and before a consonant

These contexts take into consideration the effect of resyllabification. These contexts are encoded in the following recursive function for codas.

(26) *Recursive output function to generate codas in R-to-L directional syllabification*

$$\texttt{cod'}(x) \overset{\text{def}}{=} \mathsf{cons}(x) \wedge [\mathsf{succ}(x) = \text{NULL} \vee [\mathsf{V}(\mathsf{pred}(x)) \wedge \mathsf{cons}(\mathsf{succ}(x))] \vee [\mathsf{cons}(\mathsf{pred}(x)) \wedge \texttt{cod'}(\mathsf{pred}^2(x)) \wedge \mathsf{cons}(\mathsf{succ}(x))]]$$

The computation of onsets is somewhat complicated. They are generated in the following finite list of contexts.

(27) *Contexts for generating onsets in L-to-R directional syllabification*

1. /...CV.../: before a vowel
2. // ...C)C...//: after a coda

These contexts are used to form the following function for onsets. We assume that codas can't form onsets

(28) *Output function to generate onsets in R-to-L directional syllabification*

$$\texttt{ons'}(x) \overset{\text{def}}{=} \mathsf{cons}(x) \wedge \neg\texttt{cod}(x) \wedge [\mathsf{V}(\mathsf{succ}(x)) \vee [\texttt{cod'}(\mathsf{pred}(x))]]$$

We illustrate these functions below for L-to-R syllabification..

(29) *Recursive computation of L-to-R directional syllabification*

Input	*k*	*a*	*t*	*a*	*b*	*t*	*l*	*u*	*k*	*a*	*t*	*a*	*b*	*t*	*l*	*h*	*a*
Index:	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9
$\mathsf{pred}(x)$:		1	2	3	4	5	6	7		1	2	3	4	5	6	7	8
$\mathsf{succ}(x)$:	2	3	4	5	6	7	8		2	3	4	5	6	7	8	9	
Underlying $\mathsf{V}(x)$		⊤		⊤				⊤		⊤		⊤					⊤
Underlying $\mathsf{cons}(x)$	⊤		⊤		⊤	⊤	⊤		⊤		⊤		⊤	⊤	⊤	⊤	
Output $\texttt{nuc'}(x)$:		⊤		⊤				⊤		⊤		⊤					⊤
Output $\texttt{cod'}(x)$:																	
Iteration 1:					⊤								⊤				
Iteration 2:					⊤								⊤		⊤		
Output $\texttt{ons'}(x)$	⊤		⊤			⊤	⊤		⊤		⊤			⊤		⊤	
Output:	*.k*	*a.*	*.t*	*a*	*b.*	*.t.*	*.l*	*u.*	*.k*	*a.*	*.t*	*a*	*b.*	*.t*	*.l.*	*.h*	*a.*
	O	N	O	N	C	O	O	N	O	N	O	N	C	O	C	O	N

The same epenthesis transduction from before will correctly epenthesize a vowel *i* after the onset in a defective syllable (=an onset that doesn't precede a nucleus).

(30) *Computation of epenthesis after L-to-R directional syllabification*

Input	*k*	*a*	*t*	*a*	*b*	*t*	*l*	*u*	*k*	*a*	*t*	*a*	*b*	*t*	*l*	*h*	*a*
Index:	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9
Output $\texttt{nuc'}(x)$:		⊤		⊤				⊤		⊤		⊤					⊤
Output $\texttt{ons'}(x)$	⊤			⊤	⊤		⊤		⊤		⊤			⊤		⊤	
Epenthesized $\texttt{i'}(x^2)$:					⊤									⊤			
Output:	*k*	*a*	*t*	*a*	*b*	*ti*	*l*	*u*	*k*	*a*	*t*	*a*	*b*	*ti*	*l*	*h*	*u*

5 Conclusion

This paper defined a typological restriction on iterative phonological functions based on their computational power. By relying on a notion of recursion in the output that is local only via QFLFP logical transductions, a wide range of iterative processes can be accounted for. This characterization is independent of any particular linguistic theory of these processes, as it describes properties of the input-output mapping itself. Additionally, we coded and implemented our restriction into the Boolean Monadic Reursive Schemes (BMRS) of Bhaskar et al. (2020), a related computational restriction characterizing the more expressive subsequential transformations (Code is available on one of the author's GitHub page: `https://github.com/jhdeov/BMRS`)

Other iterative phonological processes remain to be analyzed in this fashion. For example, construction of feet in a metrical analysis of stress can be described using a QFLFP transduction. This is precisely because it depends on local, recursive output-string information, with placement of further boundaries relying on the location of other, previous boundaries in the output string. Locally recursive logic provides a necessary, but sufficiently limited extension to previous methods in computational phonology. We speculate that other prosodic processes at different scales of representation also possess such restrictions. Moreover, the fact that iterative processes are describable with locally recursive logic reinforces the view that locality lies at the heart of phonological processes.

References

Aksënova, Alëna & Sanket Deshmukh (2018). Formal restrictions on multiple tiers. *Proceedings of the Society for Computation in Linguistics*, vol. 1, 64–73.

Andersson, Samuel, Hossep Dolatian & Yiding Hao (2020). Computing vowel harmony: The generative capacity of Search & Copy. *Proceedings of the Annual Meetings on Phonology*, vol. 8.

Bhaskar, Siddharth, Jane Chandlee, Adam Jardine & Christopher Oakden (2020). Boolean monadic recursive schemes as a logical characterization of the subsequential functions. Leporati, Alberto, Carlos Martín-Vide, Dana Shapira & Claudio Zandron (eds.), *Proceedings of the 14th International Conference on Language and Automata Theory and Applications (LATA 2020)*.

Bird, Steven (1995). *Computational phonology: A constraint-based approach*. Studies in Natural Language Processing, Cambridge University Press, Cambridge.

Blevins, Juliette (1995). The syllable in phonological theory. Goldsmith, John (ed.), *The Handbook of Phonological Theory*, Blackwell Publishers, Cambridge, MA, 206–244, 1 edn.

Broselow, Ellen (1992). Parametric variation in Arabic dialect phonology. Broselow, Ellen, Mushira Eid & John McCarthy (eds.), *Perspectives on Arabic linguistics IV*, John Benjamins, Amsterdam and Philadelphia, 7–46.

Broselow, Ellen (2017). Syllable structure in the dialects of Arabic. Benmamoun, Elabbas & Reem Bassiouney (eds.), *The Routledge Handbook of Arabic Linguistics*, Taylor & Francis, New York, 32–47.

Burness, Phillip & Kevin McMullin (2020). Multi-tiered strictly local functions. *Proceedings of the 17th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, 245–255.

Chandlee, Jane (2014). *Strictly Local Phonological Processes*. Ph.D. thesis, University of Delaware, Newark.

Chandlee, Jane & Jeffrey Heinz (2018). Strict locality and phonological maps. *Linguistic Inquiry* 49:1, 23–60.

Chandlee, Jane & Adam Jardine (2019). Quantifier-free least fixed point functions for phonology. *Proceedings of the 16th Meeting on the Mathematics of Language (MoL 16)*, Association for Computational Linguistics, Toronto, Canada.

Chandlee, Jane & Steven Lindell (in prep). A logical characterization of input strictly local functions. Dolatian, Hossep, Jeffrey Heinz & Kristina Strother-Garcia (eds.), *Doing Computational Phonology*, Oxford University Press, Oxford.

Chandlee, Jane, Rémi Eyraud & Jeffrey Heinz (2015). Output strictly local functions. *14th Meeting on the Mathematics of Language*, 112–125.

Coleman, John (1996). Declarative syllabification in Tashlhit Berber. Durand, Jacques & Bernard Laks (eds.), *Current trends in phonology: Models and methods*, European Studies Research Institute, University of Salford, Salford, vol. 1, 175–216.

Coleman, John (1998). *Phonological representations: Their names, forms and powers*. Cambridge University Press, Cambridge.

Coleman, John (2000). Candidate selection. *The Linguistic Review* 17:2-4, 167–180.

Coleman, John & Janet Pierrehumbert (1997). Stochastic phonological grammars and acceptability. *Third meeting of the ACL special interest group in computational phonology: Proceedings of the workshop*, Association for computational linguistics, East Stroudsburg, PA, 49–56.

Côté, Marie-Hélène (2000). *Consonant cluster phonotactics: A perceptual approach*. Ph.D. thesis, Rutgers University.

Courcelle, Bruno (1994). Monadic second-order definable graph transductions: A survey. *Theoretical Computer Science* 126:1, 53–75.

Courcelle, Bruno (1997). The expression of graph properties and graph transformations in monadic second-order logic. Rozenberg, Grzegorz (ed.), *Handbook of Graph Grammars and Computing by Graph Transformations*, World Scientific, vol. 1, 313–400.

Dolatian, Hossep (2020). *Computational locality of cyclic phonology in Armenian*. Ph.D. thesis, Stony Brook University.

Eisner, Jason (2000). Directional constraint evaluation in optimality theory. *COLING 2000 Volume 1: The 18th International Conference on Computational Linguistics*.

Engelfriet, Joost & Hendrik Jan Hoogeboom (2001). MSO definable string transductions and two-way finite-state transducers. *Transactions of the Association for Computational Linguistics* 2:2, 216–254, URL `http://doi.acm.org/10.1145/371316.371512`.

Farwaneh, Samira (1995). *Directionality effects in Arabic dialect syllable structure*. Ph.D. thesis, University of Utah.

Frampton, John (2011). Gde syllabification-a generalization of dell and elmedlaoui's syllabification algorithm. *The Linguistic Review* 28:3, 241–279.

Gainor, Brian, Regine Lai & Jeffrey Heinz (2012). Computational characterizations of vowel harmony patterns and pathologies. Choi, Jaehoon, E. Alan Hogue, Jeffrey Punske, Deniz Tat, Jessamyn Schertz & Alex Trueman (eds.), *The Proceedings of the 29th West Coast Conference on Formal Linguistics*, Cascillida Press, Somerville, MA, 63–71.

Heinz, Jeffrey (2018). The computational nature of phonological generalizations. Hyman, Larry & Frans Plank (eds.), *Phonological Typology*, Phonetics and Phonology, Mouton de Gruyter, Berlin, chap. 5, 126–195.

Heinz, Jeffrey & Regine Lai (2013). Vowel harmony and subsequentiality. Kornai, Andras & Marco Kuhlmann (eds.), *Proceedings of the 13th Meeting on the Mathematics of Language (MoL 13)*, Association for Computational Linguistics, Sofia, Bulgaria, 52–63, URL `http://www.aclweb.org/anthology/W13-3006`.

Howard, Irwin (1972). *A directional theory of rule application in phonology.* Ph.D. thesis, Massachusetts Institute of Technology.

Hulden, Mans (2006). Finite-state syllabification. Yli-Jyrä, Anssi, Lauri Karttunen & Juhani Karhumäki (eds.), *Finite-State Methods and Natural Language Processing. FSMNLP 2005. Lecture Notes in Computer Science*, Springer, Berlin/Heidelberg, vol. 4002.

Itô, Junko (1986). *Syllable theory in prosodic phonology*. Ph.D. thesis, University of Massachusetts, Amherst.

Itô, Junko (1989). A prosodic theory of epenthesis. *Natural Language & Linguistic Theory* 7:2, 217–259.

Johnson, C Douglas (1972). *Formal aspects of phonological description*. Mouton, The Hague.

Kiparsky, Paul (2003). Syllables and moras in Arabic. Féry, Caroline & Ruben van de Vijver (eds.), *The syllable in optimality theory*, Cambridge University Press, Cambridge, 147–182.

Kiraz, George Anton & Bernd Möbius (1998). Multilingual syllabification using weighted finite-state transducers. *The third ESCA/COCOSDA workshop (ETRW) on speech synthesis*.

Koser, Nathan, Christopher Oakden & Adam Jardine (2019). Tone association and output locality in non-linear structures. *Supplemental proceedings of AMP 2019*.

Libkin, Leonid (2013). *Elements of finite model theory*. Springer Science & Business Media.

Mester, Armin & Jaye Padgett (1994). Directional syllabification in generalized alignment. Merchant, Jason, Jaye Padgett & Rachel Walker (eds.), *Phonology at Santa Cruz 3*, Linguistics Research Center, Santa Cruz: CA, 79–85.

Mohri, Mehryar (1997). Finite-state transducers in language and speech processing. *Computational Linguistics* 23:2, 269–311.

Nevins, Andrew (2010). *Locality in vowel harmony*. MIT Press, Cambridge.

Prince, Alan & Paul Smolensky (2004). *Optimality Theory: Constraint Interaction in Generative Grammar*. Blackwell Publishing, Oxford.

Rogers, James (1996). Strict lt_2 : Regular :: Local : Recognizable. *International Conference on Logical Aspects of Computational Linguistics*, Springer, 366–385.

Rogers, James (1998). *A Descriptive Approach to Language-Theoretic Complexity*. CSLI Publications, Stanford, CA.
Samuels, Bridget D (2011). *Phonological architecture: A biolinguistic perspective*. Oxford Studies in Biolinguistics, Oxford University Press.
Scobbie, James M, John S Coleman & Steven Bird (1996). Key aspects of declarative phonology. Durand, Jacques & Bernard Laks (eds.), *Current Trends in Phonology: Models and Methods*, European Studies Research Institute, Salford, Manchester, vol. 2.
Street, Chester S. & Gregory P. Mollinjin (1981). The phonology of Murinbata. *Australian phonologies: Collected papers* 183–244.
Strother-Garcia, Kristina (2018). Imdlawn Tashlhiyt Berber syllabification is quantifier-free. *Proceedings of the Society for Computation in Linguistics*, vol. 1, 145–153.
Strother-Garcia, Kristina (2019). *Using model theory in phonology: a novel characterization of syllable structure and syllabification*. Ph.D. thesis, University of Delaware.
Walther, Markus (1993). Declarative syllabification with applications to German. Ellison, T. Mark & James Scobbie (eds.), *Computational Phonology*, Centre for Cognitive Science, University of Edinburgh, 55–79.
Walther, Markus (1995). A strictly lexicalized approach to phonology. Kilbury, J. & Richard Wiese (eds.), *Proceedings of DGfS/CL'95*, Deutsche Gesellschaft für Sprachwissenschaft, Sektion Computerlinguistik, Düsseldorf, p. 108–113.
Yap, Ngee Thai (2006). *Modeling syllable theory with finite-state transducers*. Ph.D. thesis, University of Delaware.