CHAPTER 5

Rewrite Rules

The notion of a grammar as a relation between sets of strings is quite ancient. In the 4th century BCE, Pāṇini—Pynini's eponym—proposed a sophisticated computational system (see Penn and Kiparsky 2012) in which Sanskrit words and phrases are constructed by repeated application of **rewrite rules**. These rules consist of a target (for example, in Aṣṭdhyāyā 6.1.77, [i, iː, uː, r̩, r̩ː]), a replacement (the corresponding semivowels), and a context in which it is applied (before another vowel). Priscian, writing in the 6th century CE, proposes a system of rewrite rules generating inflectional variants of Latin verbs (Matthews 1972). In the 19th century, historical linguists—particularly those now known as the Neogrammarians—employed rewrite rules to describe language change. For instance, Grimm's Law correctly predicts that the initial **p* in the Proto-Indo-European word meaning 'father' (cf. Latin *pater*) surfaces as [f] in Germanic (e.g., Old High German *fater*). In the 20th century, linguists adapted rewrite rules—some of which closely resemble historical sound changes—to sketch the morphophonemics of languages such as Menomini (Bloomfield 1939), Russian (Jakobson 1948), and Modern Hebrew (Chomsky 1951), and began to formalize rewrite systems. One particularly influential formal approach is the phonological rule notation popularized by Chomsky and Halle (1968). Subsequent work, reviewed below, establishes strong connections between the notation used by Chomsky and Halle and the rational relations,[1] provides algorithms to compile rewrite rules into finite-state transducers, and employs them for speech and language technology applications. This chapter formalizes rewrite rules and shows how they are compiled into finite transducers, combined into rule cascades, and applied to input strings. Their use is then illustrated with applications in speech and language technology, including **grapheme-to-phoneme conversion**, **morphological generation**, and **text normalization**; the latter two applications are discussed in greater detail in later chapters.

5.1 THE FORMALISM

This section defines **context-dependent rewrite rules**, a distillation of the formalism popularized by Chomsky and Halle's 1968 book *The Sound Pattern of English*, henceforth SPE. While the phrase "context-dependent" seems to suggest the family of formal languages known as the **context-sensitive languages**, this similarity is misleading. In fact, context-dependent rewrite rules as formalized below are equivalent to rational relations (Johnson 1972) and correspond

[1]In contrast, syntactic phenomena and the **transformational rules** used to describe them belong to higher classes of formal languages (see Shieber 1985 and citations therein).

to finite-state transducers. Furthermore, since FSTs are closed under composition, an ordered cascade of context-dependent rewrite rules can be composed into a single FST.

Before we proceed further, however, we need to clear up one possible misconception. Those familiar with SPE and phonological theories of slightly later vintage will also recall the notion of the morphophonological *cycle*, which played an important role in that theory of phonology. To wit, morphologically complex words were constructed affix-by-affix, and with each affixation process, a battery of phonological rules was applied to the (partially) constructed word. Under this model, a rule could in principle *apply to its own output*, a situation which in fact is not covered by Johnson's argument for equivalence. However this only becomes a problem if the number of cycles is unbounded. With a bounded number of cycles, one can always model the situation simply by composing the FST associated with a rule as many times as it needs to apply in the derivation. If, for the sake of simplicity, one has a single rule R, and assuming we represent the ith affixation process to the base B as $A_i(B)$, and assuming further that we have no more than k "cycles", then the resulting word can be computed as $B \leftarrow R \circ A_i(B)$, for $1 \leq i \leq k$.

If k is in principle *unbounded*, then there is a potential problem. This situation would obtain with a lexical FST that itself is *cyclic*—i.e., one where the graph that represents the FST has cycles, not to be confused with the linguistic notion of cyclicity just discussed. As far as we know, Hankamer (1989) was the first to emphasize the importance of such cases in finite-state morphology. Hankamer worked with data from Turkish, but in fact it is possible to construct theoretically unbounded cases even in English. Consider the productive derivational affixation process that makes verbs from adjectives by affixing *-ize*; the productive derivational process that turns verbs ending in *-ize* into nouns by affixing *-ation*; and the productive derivational process that makes adjectives from nouns by adding *-al*. These can be combined in a theoretically unbounded way:

- márginal
- márginalize
- marginalizátion
- marginalizátional
- marginalizátionalize
- marginalizationalizátion
- marginalizationalizátional
- marginalizationalizátionalize
- marginalizationalizationalizátion
- …

Also marked, with an accent mark over the relevant vowel, is the location of the primary stress, which moves to the right as the complex word is built up. If one accepts the SPE theory that stress assignment is cyclic, then we would have a potentially unbounded morphological process where the stress assignment rule would apply to its own output at each affixation cycle. So while the morphology itself can in principle be handled by a finite state machine with cycles, the stress assignment rule could not be handled by a finite mechanism since the rule must be applied at each affixation point.[2] But as Hankamer (397) points out for Turkish, if one builds up long enough cycles of this kind "Turkish speakers and hearers tend to get confused", and the same is true for English constructions of the kind we discussed above. So unbounded cyclicity in the SPE sense is not really a practical problem.

But let us return to the main point. To begin let us consider a simplified version of the context-dependent rewrite rule formalism. Let Σ be the **alphabet**, the set of symbols over which the rule operates; note that whereas rational relations and finite transducers may have separate input and output alphabets, this is not permissible for rewrite rules. For phonological rules, Σ might consist of all phonemes (and, possibly, their allophones) in the given language; for a grapheme-to-phoneme rule, it would contain both graphemes and phonemes; and, for text generation or processing applications, might consist of all 256 bytes. If $s, t, l, r \in \Sigma^*$, then the following is a possible rewrite rule.

(1) $s \rightarrow t \;/\; l \;\_\_\; r$

In this formalism, $s \rightarrow t$ is the **structural change** and l and r is the **environment**. The final rule can be read as "s goes to t between l and r". By convention, one may omit an empty l and/or r, so that the following are also possible rules.

(2)
$s \rightarrow t \;/\; \_\_$
$s \rightarrow t \;/\; l \;\_\_$
$s \rightarrow t \;/\; \_\_\; r$

Informally speaking, rule (1) specifies a rational relation with domain and range Σ^* such that all instances of lsr are replaced with ltr but all other substrings in Σ^* are passed through. For example, let $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ and consider the following rule.[3]

(3) $\mathsf{b} \rightarrow \mathsf{a} \;/\; \mathsf{b} \;\_\_\; \mathsf{b}$

Some example input-output pairs for this rule are shown below.

[2]Needless to say, this is only a problem if one insists that stress assignment must be a cyclic process. In the case at hand, the problem can easily be addressed by, for example, assuming that each stressable affix, in this case *-ation*, comes with its stress marked. Then one would merely need a single rule applying on the fully formed word to destress, or demote the stress of preceding syllables.

[3]This example is adapted from Section 11.1 in Bale and Reiss 2018.

(4)	bbba	→	baba
	abbbabbbc	→	abababc
	cbbca	→	cbbca

In the last example, for instance, the conditions for application are not met so input and output are identical.

5.1.1 DIRECTIONALITY

However, the formalism developed so far does not specify how (3) is to apply to strings such as abbbba. This ambiguity arises from the fact that for this string, one application of the rule could block the application of the rule at another site. One might suppose that the rule is applied to all contexts that meet the structural description, regardless of whether the contexts overlap. In **simultaneous application**, both the third and fourth characters of the input string are flanked by b's, and both are transduced to a. Alternatively, one might instead suppose that the rule is applied by scanning the input string from the start to the end of the string, three segments at a time. In **left-to-right** or **right-linear application**, however, when the three-segment window is centered on the third segment—the second b—the structural description and environment are satisfied because at that point it is flanked by a b on both sides. Therefore, the rule is applied and that segment is mapped to a. However, the rule would not apply when this three-segment window was centered on the fourth segment because that b would no longer have a b to its immediate left. Finally, **right-to-left** or **left-linear application** applies the rule while scanning the input string from the end to the beginning. An example of applying (3) to an ambiguous string is shown below.

(5)	simultaneous application:	abbbba	→	abaaba
	left-to-right application:	abbbba	→	ababba
	right-to-left application:	abbbba	→	abbaba

The three application directions correspond directly to where one will find the left-hand and right-hand environment for the rule, in the input or in the output. For a simultaneous rule, the left-hand and right-hand environments for an application must occur in the input. With a left-to-right rule, the left-hand environment must be in the output string and the right-hand environment must be in the input string; thus, a left-to-right rule application potentially affects an application of the same rule to a subsequent position in that it might create, or remove, a left-hand environment in the output for the next application of the rule. And a right-to-left rule is the reverse of a left-to-right rule: the left-hand environment must occur in the input and the right-hand environment in the output.[4]

[4]Thanks to Jeffrey Heinz for suggesting we present the differences in this way.

In SPE, all rules are assumed to apply simultaneously (op. cit., 343f.). However, Johnson (1972: ch. 5) adduces a number of phonological examples where directional application—either left-to-right or right-to-left, depending—is required. One example of this sort comes from Latin. In classical inscriptions, no distinction was made between [i, iː] and the front glide [j], or between [u, uː] and the back glide [w]; they are spelled *i* and *v*, respectively. Generally speaking, when an *i* or *v* occurs intervocalically—i.e., flanked by two other vowels—it is realized as a glide. Some examples are given below; note that *c* is [k], that intervocalic [j] is subsequently lengthened to [j.j], and that vowel length is not indicated orthographically.

(6)	caveo	→	[ka.we.oː]	'I am careful'
	ovis	→	[o.wis]	'sheep'
	peior	→	[pej.jor]	'worse'

What happens to a sequence of two or more intervocalic *i*'s and *v*'s? According to Steriade (1984), just the left-most eligible segment is read as a glide, as shown below.

(7)	avia	→	[a.wi.a]	'grandmother'
	lascivia	→	[las.kiːwi.a]	'wantonness'
	pavio	→	[pa.wi.oː]	'I beat'

To obtain the correct result, the rules mapping intervocalic *i* and *v* to glides must be applied left-to-right. Were they applied simultaneously, for example, *avia* and *pavio* would erroneously surface as *[aw.ja] and *[paw.joː], respectively.

Note, however, that directionality of application has no discernable effect for perhaps the majority of rules, and can often be ignored.

5.1.2 BOUNDARY SYMBOLS

Let ^, $ $\notin \Sigma$ be **boundary symbols** disjoint from Σ. Now let ^, the beginning-of-string symbol, to optionally appear as the left-most symbol in l, and permit $, the end-of-string symbol, to optionally appear as the right-most symbol in r; the boundary symbols are not permitted to appear elsewhere in l or r, or anywhere within the structural description and change. Consider the following rule, a slight variant of (3).

(8) b → a / ^ b __ b

To apply this rule, one must scan for string-initial instances of bbb and replace it with bab; given this specification, directionality is irrelevant. Example input-output pairs are given below.

(9)	bbba	→	baba
	abbbc	→	abbbc

Note that the rule does not apply in the latter case because the `bbb` sequence appears non-initially. Or, consider another variant of (3).

(10) $\texttt{b} \rightarrow \texttt{a} \,/\, \texttt{b} \_\_ \$$

To apply this rule, one must merely scan for string-final instances of `bb` and replace them with `ba`.

5.1.3 GENERALIZATION

Now consider a generalization of rewrite rules, in which a rewrite rule is specified by a five-tuple consisting of

1. an alphabet Σ,
2. a structural change $\tau \subseteq \Sigma^* \times \Sigma^*$,
3. a **left environment** $L \subseteq \{\hat{}\}^? \Sigma^*$,
4. a **right environment** $R \subseteq \Sigma^* \{\$\}^?$, and
5. a **directionality** (one of: "simultaneous", "left-to-right", or "right-to-left").

In other words, the structural change previously expressed by the strings s and t is now a rational relation, and the left and right environments previously expressed by strings l and r are now regular languages. The generalization about Latin intervocalic glides immediately above can be stated as a single rewrite rule (ignoring the lengthening of [j] to [j.j]) such that $\tau = \{(\texttt{u}, \texttt{w}), (\texttt{i}, \texttt{j})\}$, L and R are the set of vowels $\{\texttt{a}, \texttt{e}, \texttt{o}, \texttt{u}, \ldots\}$, and direction of application is left-to-right.

5.1.4 ABBREVIATORY DEVICES

In the SPE tradition, the structural change and environment are usually specified as bundles of **phonological features**, an intensional specification of segments, rather than the extensional specifications of languages used above. For instance, [+`Vowel`] denotes the set of vowels, [+`Vowel`, +`High`] the set of high vowels, [+`Vocalic`, −`Vowel`] the set of glides, and so on. Using this notation, the intervocalic formation rule might be written roughly as follows:

(11) $\begin{bmatrix} +\texttt{Vocalic} \\ +\texttt{High} \end{bmatrix} \rightarrow \{-\texttt{Vowel}\} \,/\, [+\texttt{Vowel}] \_\_ [+\texttt{Vowel}]$ (left-to-right)

Assuming a finite inventory of segments and a mapping from segments to their features, it is possible to compute the regular language extension for any featural specification. However, it is not always the case that a given regular language corresponds to a featural specification. For instance, consider the following feature system for the vowels of Turkish shown in Table 5.1. A

Table 5.1: Featural specification for the vowels of Turkish.

	[+Front]		[-Front]	
	[-Round]	[+Round]	[-Round]	[+Round]
[+High]	i	ü	ı	u
[-High]	e	ö	a	o

simple featural specification like [+Round] defines the set {ü, u, ö, o}; a complex featural specification bundle such as [+Front, +Round] gives the intersection [+Front] ∩ [+Round], the set of segments which are both front and round, namely {ü, ö}. All sets defined by the intersection of one or more featural specifications are known as **natural classes**. However, not all sets of the above vowels can be described in the above fashion: for instance, there is no featural specification corresponding to the set {e, o} under this feature system. Such classes are called **unnatural classes**.[5] However, in our opinion, the extensional notation's ability to express unnatural classes is no defect. When dealing with phonological processes, one can construct natural classes via intersection of extensionally defined sets of segments. Furthermore, linguists have encountered many cases in which a superficially similar series of phonological changes lack a unified expression under current mappings between features and phones. Finally, there is no obvious equivalent to featural specifications for an alphabet consisting of, for instance, orthographic characters rather than phonemic-phonetic symbols.

Linguists may be familiar with several other types of abbreviatory devices used in SPE and related work. One such convention are the **Boolean variables**, traditionally written with **Greek-letter variables**, ranging over {+, −} (Zwicky 1965). Johnson (1972: ch. 3) proves that under certain reasonable assumptions, this is a purely abbreviatory convention and does not increase the expressivity of the SPE formalism. Another such device is the **curly brace notation** used to express disjunctions of feature bundles and/or segments, and is equivalent to union. Finally, SPE uses **parentheses-schemata** to express certain long-distance processes such as stress assignment and related phenomena such as the reduction of weak syllables. The SPE schemata notation was widely critiqued at the time (e.g., Anderson 1974: ch. 9, Kenstowicz and Kisseberth 1977:189f., Piggott 1975) and subsequent technical developments, including directional rule application and the conception of prosodic-metrical structure as belonging to a separate representational tier (e.g., Hayes 1980, Idsardi 1992), have largely made this convention obsolete.

[5] Unnatural classes are by no means uncommon in phonological descriptions. For example, the Breton consonant mutation process known as **lenition** changes voiceless stops into the corresponding voiced stop (e.g., /t/ → /d/) and voiced stops into fricatives. However the set of fricatives produced does not form a natural class, since while /b/ changes to /v/, and /d/ changes to /z/, both voiced fricatives, /g/ changes to /x/, a voiceless fricative. Therefore, a simple rule that states that a [+Voice] stop becomes [+Continuant] will not do: one must add a special case for /g/.

5.1.5 CONSTRAINT-BASED FORMALISMS

We have little to say about **Optimality Theory** (OT; Prince and Smolensky 2004), **Harmonic Grammar** (Legendre et al. 2006), and related frameworks which formulate transductions using violable constraints and global optimization. It is generally believed that one can restrict the generation procedure ("Gen") and constraint set ("Con") so that all OT grammars correspond to an equivalent rational relation, and to an equivalent finite transducer (e.g., Eisner 1997, Heinz et al. 2009, Karttunen 1998), and it seems likely that similar equivalencies may hold between harmonic grammars, weighted rational relations, and weighted finite transducers. However, the proposed restrictions have largely been ignored by OT practitioners. Furthermore, we find that finite-state grammars based on rewrite rules—or a mixture of rules and constraints—tend to be more terse, and more easily understood and maintained, than those expressed via constraints alone.[6] Finally, it seems quite likely that standard forms of Optimality Theory are overly restrictive in their inability to express **opaque mappings** (in the sense of Chandlee et al. 2018) such as **counterfeeding** and **counterbleeding** (Kiparsky 1973).

5.2 RULE COMPILATION

Johnson (1972) proves that context-dependent rewrite rules are rational relations, and this result can be generalized to **weighted rewrite rules** by allowing τ, the structural change, to be a weighted rational relation (Mohri and Sproat 1996). Rules which apply only at the end or beginning of a string are generally trivial to express as a finite transducer. For example, consider the following rewrite rules, which prepend a prefix p or append a suffix s, respectively.

(12)
$$\begin{array}{l} \emptyset \to \{p\} \;/\; \{\hat{}\} \_\_ \Sigma^* \\ \emptyset \to \{s\} \;/\; \Sigma^* \_\_ \{\$\} \end{array}$$

Such rules are equivalent to the rational relations $(\emptyset \times \{p\})\,\Sigma^*$ and $\Sigma^*\,(\emptyset \times \{s\})$, respectively. Greater difficulties arise from the possibility of multiple application with or without overlapping contexts for application, and the development of a general-purpose algorithm to compile (weighted) finite-state transducers from algebraic descriptions of rewrite rules proved quite challenging. Koskenniemi (1983), who developed one of the earliest finite-state grammars, a comprehensive description of Finnish morphophonology, used rewrite rules that were manually compiled into finite transducers, state-by-state and arc-by-arc. An early sketch of a rewrite rule compilation algorithm appeared in an unpublished 1981 lecture by Ronald Kaplan and Martin Kay, researchers at the Xerox Palo Alto Research Center (PARC), but was not published until much later (e.g., Kaplan and Kay 1994, Karttunen 1995).

[6] One example of this can be found in a study of stress in Shanghai compounds by Duanmu (1997); a rule-based analysis of the phenomenon fits on a single page whereas a roughly equivalent OT analysis of the same facts spans two dozen.

5.2.1 THE ALGORITHM

This section describes a generalization of the Kaplan and Kay-Karttunen rule compilation algorithm by Mohri and Sproat (1996). Their technique consists of the composition of five transducers, each a simple rational relation. For simplicity of explication we focus on left-to-right application, though minor variants of the algorithm produce right-to-left or simultaneous rules. First, if X is a language, let $\overline{X}$ denote its **complement**, the language consisting of all strings which are not elements of X; i.e., $\overline{X} = \{x \mid x \notin X\}$. Then, let $<_1, <_2, > \notin \Sigma$ be **marker symbols** disjoint from the alphabet Σ. Then, given a rule over Σ^* defined by structural change τ, left and right contexts L and R, the constituent transducers are defined as follows. (We remind the reader that π_i denotes the **projection** onto the input of a relation.)

1. ρ inserts the $>$ marker before all substrings matching R (i.e., it marks the beginnings of the righthand environment):
   $\emptyset \rightarrow > \ / \ \Sigma^* \_R$.

2. ϕ inserts markers $<_1$ and $<_2$ before all substrings matching $\pi_i(\tau) >$:
   $\emptyset \rightarrow \{<_1, <_2\}/(\Sigma \cup \{>\})^* \_\pi_i(\tau)$[7]

3. γ applies the structural change τ anywhere $\pi_i(\tau)$ is preceded by $<_1$ and followed by $>$. It simultaneously deletes the $>$ marker everywhere. This transducer is schematicized in Figure 5.1.

4. λ_1 admits only those strings in which L is followed by the $<_1$ marker and deletes all $<_1$ markers satisfying this condition:
   $<_1 \rightarrow \emptyset / \ \Sigma^* L\_$.

5. λ_2 admits only those strings in which all $<_2$ markers are not preceded by L and deletes all $<_2$ markers satisfying this condition:
   $<_2 \rightarrow \emptyset / \ \Sigma^* \overline{L}\_$

In short, the ρ, ϕ, and γ transducers apply the τ transduction only between R and $<_1$; λ_1 guarantees that $<_1$ occurs only when preceded by L, and λ_2 guarantees that non-application occurs only when L does not precede. Then, the final context-dependent rewrite rule transducer is given by $\zeta = \rho \circ \phi \circ \gamma \circ \lambda_1 \circ \lambda_2$. An example is given in Figure 5.2.

To obtain right-to-left application, for example, one reverses the process: a λ transducer would insert a $<$ marker after all substrings matching L; ϕ would insert $>_1, >_2$ after all strings matching $< \pi_i(\tau)$; γ would apply the structural change to any τ preceded by $<$ and followed by $>_1$; ρ_1 would admit only those strings in which the $>_1$ marker is followed by R; and ρ_2 would admit only those strings in which $>_2$ is followed by $\overline{R}$, with ρ_1 and ρ_2 also deleting their respective markers. Then, $\zeta = \lambda \circ \phi \circ \gamma \circ \rho_1 \circ \rho_2$.

[7]This introduces two paths, one with $<_1$ and one with $<_2$, which ultimately correspond to the cases where L does and does not, respectively, occur to the left of the structural change (see steps 4–5).

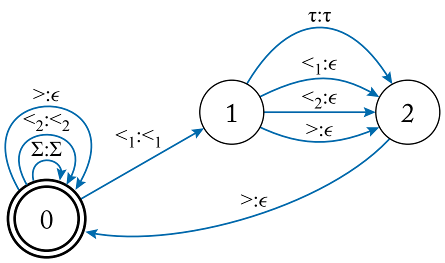

Figure 5.1: Schematic of the γ transducer, after Mohri and Sproat 1996, Figure 2. Note the arcs labeled with the regular language Σ and the rational relation τ; the corresponding automata must replace these arc to create γ.

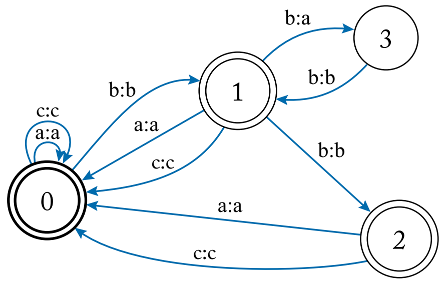

Figure 5.2: A transducer implementing the rewrite rule $\{\mathsf{b}\} \times \{\mathsf{a}\} \;/\; \{\mathsf{b}\} \underline{\quad} \{\mathsf{b}\}$ assuming left-to-right application and $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$.

One further detail is the handling of boundary symbols. When a boundary symbol is present in the context languages L or R, one can introduce them with a transducer $\beta = (\emptyset \times \{\text{\^{}}\})\ \Sigma^*\ (\emptyset \times \{\$\})$, and remove them using the inverse transducer β^{-1}. Note that the boundary symbols are not members of Σ, but ζ must treat them much as any other symbol. Therefore, one must construct ζ using $\Sigma' = \Sigma \cup \{\text{\^{}}, \$\}$ in place of Σ. Then, the full construction is given by $\zeta' = \beta \circ \zeta \circ \beta^{-1}$.

Finally, Mohri and Sproat (1996) discuss the extension of the algorithm to the weighted case. In general τ can be a weighted transducer, meaning that a span of arcs introducing the output side of $\pi_o(\tau)$ will be weighted. If, for example, a final stop devoicing rule devoices a final stop 90% of the time, and leaves it voiced 10% of the time, one can model this with a τ that maps, say, /d/ to /t/ with probability semiring weight 0.9, and leaves it as /d/ with probability semiring weight 0.1. Then, in the appropriate string-final environment, an underlying /d/ will surface as /t/ with weight 0.9 and as /d/ with weight 0.1. For a cascade of weighted rules, implemented using composition, the weights of the individual rules are combined using semiring $\otimes$.

5.2.2 EFFICIENCY CONSIDERATIONS

The operations constructing the component transducers are relatively efficient, with one exception: rules introducing markers require the underlying acceptors to be deterministic. Consider a transducer which inserts a marker # after the regular language L; this corresponds loosely to several of the constituent transducers used by the Mohri and Sproat construction. In order to do this we need to construct a finite transducer that allows any sequence of characters in Σ^*, possibly including previous instances of L, and inserts # after L. Such a transducer is obtained by inserting an arc with an ϵ input label and a # output label after the final state(s) of L and marking the new destination state final. This requires that the input acceptor be deterministic so that for any string $l \in L$, there is but one path through the automaton; otherwise the string could bypass insertion of #. As mentioned above, determinization can be quite expensive, particularly if Σ is large or if L is complex. So, for example, if one would like to write a rule that inserts a given word after a set of words, and this latter set of words numbers in the tens of thousands, then the compilation of said rule will be somewhat slow and the resulting FST quite large. Fortunately there is usually a way around such problems. In the hypothetical example above, one could arrange for all words in one's list of interest to be pre-tagged already with some marker, and condition the rule's environment on that marker alone. This will result in more efficient compilation and application, and produce a more compact rule transducer.

When constructing large or complex rules, it is often wise to apply optimization (section 4.1) to the resulting automata.

5.2.3 RULE COMPILATION IN PYNINI

The Pynini `cdrewrite` function is used to compile context-dependent rewrite rules into WFSTs. The first four arguments, all mandatory, include a transducer defining τ, unweighted acceptors defining the left and right environments L and R—an empty string is used for null environments—and a cyclic, unweighted acceptor representing Σ^*, the closure over the rule's alphabet. The reserved symbol `[BOS]` ("beginning of string") can be used to symbolize the left boundary symbol ^ when constructing L; similarly, `[EOS]` ("end of string") can be used to symbolize the right boundary symbol \$ when constructing R. The optional keyword argument `mode` allows one to specify directionality of application; left-to-right application is the default. Some example snippets are given below.

- Compile rule (3) for simultaneous application:

```
rule = pynini.cdrewrite(
    pynini.cross("b", "a"),
    "b",
    "b",
    pynini.union("a", "b", "c").closure(),
        direction="sim"
    )
```

- Compile rule (8) for left-to-right application:

```
rule = pynini.cdrewrite(
    pynini.cross("b", "a"),
    "[BOS]b",
    "b",
    pynini.union("a", "b", "c").closure()
)
```

5.3 RULE APPLICATION

At first glance, applying a rule to a string may seem as simple as composing the input string with the rewrite rule transducer and then extracting the output string. However, there are several complexities which may arise when the rule transducers, or the composition of the input string and rule, are cyclic, non-deterministic, and/or weighted. Advanced procedures for rewrite rule application are described below.

5.3.1 LATTICE CONSTRUCTION

The first step in rule application is the creation of a **lattice** of output strings. First, one composes the input acceptor with the rule transducer, possibly taking advantage of implicit conversion from strings to acceptors. If the string is not in the domain of the rule, this will result in an empty automaton, and one may wish to check for this case before continuing.

```
lattice = string @ rule
assert lattice.start() != pynini.NO_STATE_ID
```

Then, to convert this automaton from a transducer to an acceptor over output strings, one simply computes the output projection, optionally applying ϵ-removal afterward. Both operations can be applied in-place.

```
lattice.project("output").rmepsilon()
```

5.3.2 STRING EXTRACTION

There are several different ways to extract output strings from the lattice. If the rule provides only one output string—or if the rule transducer is deterministic—then there will be at most one output path. Similarly, if one wishes to extract only the shortest path from a lattice which may contain multiple paths, one can first call the `shortestpath` function. One can then simply call the `string` method to retrieve the output string, as shown below.

```
ostring = pynini.shortestpath(lattice).string()
```

If multiple strings are desired, one can simply extract all output strings using the path iterator so long as the lattice is acyclic. However, a non-deterministic lattice may be ambiguous (section 4.1), meaning that there are multiple paths generating an output string. To avoid this, one may wish to determinize the lattice before extracting strings. However, as discussed in section 4.1, not all weighted non-deterministic automata (NFAs) are determinizable, and even for those which are determinizable, the algorithm may produce a deterministic automaton (DFA) which is exponentially larger than the equivalent NFA.[8] If one needs only the n shortest paths, then Pynini's `shortestpath` function with `unique=True`, will perform just as much determinization as is required to find the n shortest paths. This will generate an acyclic, determistic, ϵ-free acceptor, from which one can retrieve the output strings. This is illustrated in the snippet below.

```
lattice = pynini.shortestpath(lattice, nshortest=n, unique=True)
ostrings = list(lattice.paths().ostrings())
```

Alternatively, one may be interested in retrieving all rewrites. In this case, to forestall the possibility of exponential blowup, one can approximate the full deterministic automaton using **pruned determinization** (Rybach et al. 2017). This algorithm computes a DFA approximating the NFA while preferring shorter paths as measured by path weight. The user must specify a state threshold—an upper bound for the size of the determinstic FSA—a weight threshold—a lower bound for the weights of paths in the approximation—or both. In practice, pruned determinization with a state threshold that is some small multiple of the number of states in the non-deterministic FSA (e.g., 4) plus a small constant factor (e.g., 256) yields an exactly-equivalent deterministic FSA for the vast majority of real-world cases, while eliminating the risk of exponential blowup. This is illustrated below.

```
state_threshold = 4 * lattice.num_states() + 256
lattice = pynini.determinize(lattice, nstate=state_threshold)
ostrings = list(lattice.paths().ostrings())
```

Another form of pruned determinization can be used to detect whether there are ties for the single shortest path and to avoid implementation-defined tie resolution. One can apply pruned determinization with a weight threshold of $\bar{1}$, which gives an acyclic, deterministic, ϵ-free acceptor containing all optimal paths, i.e., paths whose weight is the same as that of the single shortest path. This form of pruning is shown in the following snippet.

```
one = pynini.Weight.one(lattice.weight_type())
lattice = pynini.determinize(lattice, weight=one)
```

[8] For example, Hopcroft et al. (2008:§2.3.6) describe a class of NFAs with $n + 1$ states for which the equivalent DFA must have at least 2^n states. However, such cases are rare in practice.

If the resulting DFA contains multiple paths, a tie for the single shortest path is present. One can then extract all optimal paths from the DFA using a path iterator, as shown in the following snippet.

```
ostrings = list(lattice.paths().ostrings())
```

Note, however, that the shortest path and pruned determinization algorithms are well defined only over path semirings (subsection 1.5.1).

5.3.3 REWRITING LIBRARIES

The `rewrite` module, part of Pynini's extended library (Appendix C), automates the rewriting procedures described above. It defines several functions which take an input string (or automaton) and a rule transducer, construct the output lattice, and return either a single output string or a list thereof.

1. `rewrites` returns all output strings in an arbitrary order.
2. `top_rewrites` returns the n shortest-path output strings in an arbitrary order.
3. `top_rewrite` returns the shortest-path output string using implementation-defined tie resolution.
4. `one_top_rewrite` returns the single shortest-path output string, raising an exception if there is a tie for the single shortest path.
5. `optimal_rewrites` returns all output strings which have the same weight as the single shortest path in an arbitrary order.

The `rewrite` module also defines the `matches` function, which tests if a rule transducer ζ matches an input-output string pair. A rule ζ matches input x to output y if the intersection $\pi_o(x \circ \zeta) \cap y$ is non-null.

5.4 RULE INTERACTION

There are various mechanisms one can use to combine rewrite rules. These are described below.

5.4.1 TWO-LEVEL RULES

Influential early work by Koskenniemi (1983) proposes a **two-level** model of rule interaction. This approach is, in the authors' opinion, quite obsolete, but is still useful to motivate more sophisticated forms of rule interaction. In the two-level model, each rewrite rule is an assertion about relations between the substrings of the relation the rule governs, i.e., it expresses a constraint on the relation. For instance, Koskenniemi's **composite rules** state that for symbols $s, t \in \Sigma$ the input s must correspond to t in all environments defined by L and R. Two-level rules must be **surface-true** in the sense that they must express exceptionless generalizations about the relationship between input and output substrings. By definition, then, surface-true rules do not interact, and can be combined into a single transducer using an algorithm that is functionally equivalent to intersection (Roark and Sproat 2007:105f.).

As an example, consider the following generalizations regarding the pronunciation of *c* in Latin American dialects of Spanish.

1. *c* is read as [s] when followed by *i* or *e* (e.g., *cima* /sima/ 'summit').

2. *c* is read as [k] elsewhere (e.g., *escudo* /eskudo/ 'shield').

The following surface-true composite rules formalize these generalizations.

(13) $\{\mathtt{c}\} \times \{\mathtt{s}\} \,/\, \_\_ \, \{\mathtt{i}, \mathtt{e}\}$

(14) $\{\mathtt{c}\} \times \{\mathtt{k}\} \,/\, \_\_ \, (\Sigma - \{\mathtt{i}, \mathtt{e}\})$

Note that while the prose description refers to the reading of *c* as /k/ as the "elsewhere" case, the two-level model requires one to explicitly define the right-context regular language to avoid overlap with other rules. While this is not particularly difficult here, the surface-true restriction can quickly become onerous when working with large, complex collections of rewrite rules. That said, we note that work in the two-level tradition has continued, and there have been many refinements and improvements to the formalism, for example Yli-Jyrä and Koskenniemi 2006, Koskenniemi and Silfverberg 2010, Drobac et al. 2012 and Yli-Jyrä 2013.

5.4.2 CASCADING

The most common alternative to the two-level approach described above is **cascading** or **chaining**, in which rules are applied in a fixed, user-specified order, with the output of one rule fed as input to the next. While a cascade can include surface-true rules, which will apply independently of any other rule, an earlier rule may also—in the terminology of Kiparsky (1968)—**feed**—i.e., trigger, or create the environment for—or **bleed**—block, or eliminate the environment for—a later rule in the course of derivation. By loosening the requirement that rules be surface-true, one can simplify the rules governing Spanish *c*. Consider the rule below.

(15) $\{\mathtt{c}\} \times \{\mathtt{k}\} \,/\, \_\_$

Unlike earlier rules, this rule is not surface-true: a *c* followed by *i* or *e* does not correspond to surface [k]. However, if one applies (15) to the output of rule (13), this problem is eliminated. In other words, (15) is applied after earlier rules have eliminated any exceptions and therefore can apply "across the board". This is schematicized below.

(16)

cima	*escudo*	orthographic form
sima		(13)
	eskudo	(15)
sima	eskudo	phonemic form

There are two ways to implement cascading of a sequence of rewrite rules. In the first method, one simply loops over the rules in specified order. On the first iteration of the loop, one constructs a lattice following the method described in subsection 5.3.1. On subsequent iterations, the lattice from the previous iteration is used as input for the next rule in the cascade, and another lattice is constructed. An example function illustrating this logic is shown below.

```
def cascade(istring: pynini.FstLike,
            rules: Iterable[pynini.Fst]) -> pynini.Fst:
    lattice = istring
    for rule in rules:
        lattice @= rule
        assert lattice.start() != pynini.NO_STATE_ID
        lattice.project("output").rmepsilon()
    return lattice
```

Then, one can extract strings from the output lattice using the methods introduced above in subsection 5.3.2. A complete implementation of this procedure is provided by the Pynini extended library module `rule_cascade`; its `RuleCascade` class generalizes the `rewrite` functions to rule cascades.

However, it is also straightforward to combine a sequence of context-dependent rewrite rules into a single transducer. Since weighted finite transducers are closed under composition, one can create a single rule implementing the cascade by composing the rules in the order in which they are to be applied, optionally optimizing the resulting transducer. For instance, a cascade of the rule sequence `[r1, r2, r3]` is equivalent to simple rule application with the transducer `r1 @ r2 @ r3`. Composition and optimization of a rule cascade may be computationally expensive, particularly when the constituent rules are complex or when Σ is large. One generally prefers the loop-based cascading described earlier when an automaton produced by composing multiple rules would be prohibitively large.

5.4.3 EXCLUSION

Occasionally one may wish to apply a sequence of rules so that the successful application of an earlier rule blocks application of any and all later rules, a form of interaction known as **exclusion**. Exclusion is applicable when a series of rules are in competition, and are specified so that they may fail to apply to certain input strings. For instance, one may wish to compose an irregular rule so that it applies to a subset of possible inputs, pre-empting application a general rule that can apply to any string in Σ^*. As an example, consider the orthographic form of English noun plurals. Most plurals are formed by appending an *-s* to the singular form, but there are a number of exceptions. First, there are “zero” plurals like *deer*, stem-change plurals such as *feet* and *mice*, irregularly suffixed plurals as in *oxen*, plurals with *f*-*v* alternations as in *wolves*, and Greco-Latin plurals like *nuclei*. The following expression constructs a transducer mapping between singular and plural forms for nouns with irregular plurals.

```
irregular = pynini.string_map(
    [
        "deer",
        "fish",
        "sheep",
        ("foot", "feet"),
        ("mouse", "mice"),
        ("child", "children"),
        ("ox", "oxen"),
        ("wife", "wives"),
        ("wolf", "wolves"),
        ("analysis", "analyses"),
        ("nucleus", "nuclei"),
        ...
    ]
)
```

Only three other rules are required to handle the vast majority of noun plurals. But first, one must define the relevant inventories, including lowercase orthographic vowels (v), consonants (c), and the closure of their union (`sigma_star`).

```
v = pynini.union("a", "e", "i", "o", "u")
c = pynini.union(
    "b", "c", "d", "f", "g", "h", "j", "k", "l", "m", "n",
    "p", "q", "r", "s", "t", "v", "w", "x", "y", "z"
)
sigma_star = pynini.union(v, c).closure().optimize()
```

One rule converts a final *-y* to *-ies* when *-y* is immediately preceded by a consonant, as in *puppies*, but not when the *-y* is preceded by a vowel, as in *boys*.

```
ies = sigma_star + c + pynini.cross("y", "ies")
```

Another rule appends *-es* to stems ending in *-s*, *-sh*, *-ch*, *-x*, and *-z* as in *churches* or *faxes*. For this one can use the `insert` function from the `pynutil` module, part of Pynini’s extended library (see Appendix C). Given an acceptor representing the regular language A, `insert` constructs a transducer representing the rational relation $\emptyset \times A = \{(\epsilon, a\} \mid a \in A\}$.[9]

```
sibilant = pynini.union("s", "sh", "ch", "x", "z")
es = sigma_star + sibilant + pynutil.insert("es")
```

The third and final rule simply appends an *-s*. It imposes no conditions on application other than the requirement that the input be a substring of Σ^*.

```
s = sigma_star + pynutil.insert("s")
```

It should be clear that this last rule is not surface-true. But at the same time one cannot obtain the expected result with cascading either; a cascade of the four rules would produce the erroneous **wolvesess*.

There are two ways to apply these rules so that successful application of an earlier rule excludes subsequent rules. One method simply uses a loop to apply each rule in user-specified order. If at any point rule application succeeds—i.e., produces a non-empty lattice—this lattice is returned. Otherwise, the next rule in the list is applied to the input string. An example function implementing this procedure is provided below.

```
def exclude(istring: pynini.FstLike,
            rules: Iterable[pynini.Fst]) -> Optional[pynini.Fst]:
    for rule in rules:
        lattice = istring @ rule
        if lattice.start() == pynini.NO_STATE_ID:
            continue
        return lattice.project("output").rmepsilon()
```

However, it is also possible to combine all rules into a single transducer and still obtain the desired exclusion logic. The **priority union** (Kaplan 1987, Karttunen 1998) of two rational relations μ and ν is similar to their union except that the first relation takes precedence over the second. For instance, suppose that μ transduces string a to b, and ν transduces a to c. Then whereas $\mu \cup \nu$—their union—maps a to both b and c, their priority union gives precedent to μ, and thus only maps a onto b. Formally, priority union can be defined in terms of elementary operations over rational relations. Intuitively, the priority union requires one to filter from ν those relations which are governed by μ. Assuming that $\pi_i(\mu)$ is a subset of Σ^*, $\overline{\pi_i(\mu)}$, the complement of the domain of μ, is given by the regular language $\Sigma^* - \pi_i(\mu)$. Then, the priority union of μ and ν is written

$$\mu \cup_p \nu = \mu \cup \left(\overline{\pi_i(\mu)} \circ \nu\right)$$

[9]The `pyntil` module also defines a `delete` function which constructs a transducer implementing $A \times \emptyset = \{(a, \epsilon) \mid a \in A\}$.

where $\cup_p$ is the priority union operator. At the automaton level, priority union is well defined so long as $\pi_i(\mu)$ is determinizable, as it will be so long as it is unweighted. An example function implementing this logic is provided below.

```
def priority_union(mu: pynini.Fst, nu: pynini.Fst,
                   sigma_star: pynini.Fst) -> pynini.Fst:
    nu_not_mu = (sigma_star - pynini.project(mu, "input")) @ nu
    return mu | nu_not_mu
```

To construct the compound plural rule, one repeatedly applies this function, and then optionally optimizes the result, as in the following snippet.

```
rule = priority_union(
    irregular,
    priority_union(ies,
                   priority_union(es, s, sigma_star),
                   sigma_star),
    sigma_star,
).optimize()
```

The `plurals` module, part of Pynini's `examples` library (Appendix D), contains the above implementation of the pluralization rules. The interested reader might improve this module by adding to the list of irregulars or by exploiting other subregularities in the system.

5.5 EXAMPLES

Three applications of rewrite rules are now illustrated.

5.5.1 SPANISH GRAPHEME-TO-PHONEME CONVERSION

Speech technologies like automatic speech recognition and text-to-speech synthesis require mappings between words and their pronunciations. When large, digital pronunciation dictionaries are available, various machine learning techniques can be used to induce these mappings (e.g., Gorman et al. 2020). But for many orthographies, the relation is simple enough that one can simply enumerate the necessary rewrite rules and compile them into a transducer. Spanish is an example of what Rogers (2005) calls a **shallow phonemic orthography**, because there is a near one-to-one relation between characters (**graphemes**) and their pronunciations (**phonemes**). Thus, with a small number of rewrite rules, it is possible to correctly predict the pronunciation of every Spanish word—except for the occasional unassimilated loanword—from its spelling alone. In fact, the rules governing Spanish pronunciation are sufficiently simple that only a few rewrite rules are needed. We now sketch out a grapheme-to-phoneme conversion system that maps Spanish words onto a broad International Phonetic Alphabet (IPA) transcription of their pronunciation, using a cascade of context-dependent rewrite rules.

Let G be the set of graphemes and P the set of phonemes. Then, it is natural to conceive of grapheme-to-phoneme conversion as a function or relation between strings of graphemes in G^*, henceforth written in italics, and phonemes in P^*, henceforth enclosed in square brackets.[10] The snippet below defines these two sets for our sketch of Spanish; note that `g` and `p` are not properly disjoint because, for instance, *t* is pronounced [t].

```
g = pynini.union(
    "a", "á", "b", "c", "d", "e", "é", "f", "g", "h", "i", "í",
    "j", "k", "l", "m", "n", "ñ", "o", "ó", "p", "q", "r", "s",
    "t", "u", "ú", "ü", "v", "w", "x", "y", "z"
)
p = pynini.union(
    "a", "b", "d", "e", "f", "g", "i", "j", "k", "l", "ʝ", "m",
    "n", "ɲ", "o", "p", "r", "ɾ", "s", "ʃ", "t", "u", "w", "x", "z"
)
```

Recall, however, that context-dependent rewrite rules compilation requires that all rules be relations over some Σ^*. Therefore, Σ^* is defined to be $(G \cup P)^*$.

```
sigma_star = pynini.union(g, p).closure().optimize()
```

The cascade approach to rule interaction is assumed here. The first rewrite rule pronounces the digraphs *ch*, *ll* and *qu*, handles the readings of *j*, *ñ*, *v*, *x*, and *y*, and removes acute accents. While each of these might be thought of as a logically separate rule, they are all unconditioned, surface-true generalizations and therefore can be expressed as a single rewrite rule.

```
r1 = pynini.cdrewrite(
    pynini.string_map(
        [
            ("ch", "tʃ"),
            ("ll", "ʝ"),
            ("qu", "k"),
            ("j", "x"),
            ("ñ", "ɲ"),
            ("v", "b"),
            ("x", "s"),
            ("y", "j"),
            ("á", "a"),
            ("é", "e"),
            ("í", "i"),
```

[10]We use square brackets here because while the output of this grapheme-to-phoneme transducer is broad, we do not wish to make precise claims about the ontological status of these segments in the grammar of Spanish.

```
            ("ó", "o"),
            ("ú", "u"),
            ("ü", "w"),
        ]
    ),
    "",
    "",
    sigma_star,
).optimize()
```

The next rule is a simple one; it deletes *h*, which is normally silent. This rule must be applied after the previous block so that *ch* receives the proper reading. Were the rules applied in the opposite order—or simultaneously, as part of the same rewrite rule—one would not obtain the proper reading for the *ch* digraph.

```
r2 = pynini.cdrewrite(
    pynutil.delete("h"),
    "",
    "",
    sigma_star
).optimize()
```

The third block rewrite rules deals with the pronunciation of *r*. In intervocalic position, Spanish has a contrast between *r*, a flap, as in *pero* 'but', and the trill *rr*, as in *perro* 'dog'. This is addressed using two ordered rules. The first, `r3`, maps intervocalic *r* to the flap; the second, `r4`, maps *rr*—which only occurs intervocalically—onto the trill. One can verify that the two rules must be applied in the order specified here to obtain the correct result.

```
v = pynini.union("a", "e", "i", "o", "u")
r3 = pynini.cdrewrite(
    pynini.cross("r", "ɾ"),
    v,
    v,
    sigma_star
).optimize()
r4 = pynini.cdrewrite(
    pynini.cross("rr", "r"), "", "", sigma_star
).optimize()
```

The third and final block of rules deals with the realization of *c* and *g*. The relevant generalizations, repeated in part from earlier, are given below.

1. *c* is read as [s] when followed by *i* or *e*, as in *cima* [sima] 'summit'.

2. *c* is read as [k] in all other positions, as in *escudo* [eskudo] 'shield'.
3. *g* is read as [x] when followed by *i* or *e*, as in *gema* [xema] 'gem'.
4. *g* is read as [g] in all other positions, as in *gato* [gato] 'cat'.

The following rules handle these cases. The first rule, `r5`, handles the readings of *c* and *g* before the front vowels; the second, `r6`, maps *c* to /k/ in all other positions; any remaining *g* requires no additional rules.

```
r5 = pynini.cdrewrite(
    pynini.string_map([("c", "s"), ("g", "x")]),
    "",
    pynini.union("i", "e"),
    sigma_star
).optimize()
r6 = pynini.cdrewrite(
    pynini.cross("c", "k"), "", "", sigma_star
).optimize()
```

As defined above, each of the rules operates over a Σ^* containing both graphemes and phonemes. It may be desirable, however, to restrict the input to grapheme sequences and the output to phoneme sequences. Let us suppose that the cascade is represented by the transducer ζ. This filtering can then be affected by $G^* \circ \zeta \circ P^*$—which intersects the domain with G^* and the range with P^*—and optionally optimizing the result, as shown in the following snippet.

```
rules = r1 @ r2 @ r3 @ r4 @ r5 @ r6
g2p = pynini.closure(g) @ rules @ pynini.closure(p)
g2p.optimize()
```

These six rewrite rules, applied as a cascade or composed in order, sketch out a simple grapheme-to-phoneme converter. However, one should note that a few liberties with Spanish pronunciation have been taken:

- It was assumed that *x* is always read /s/ but it may also be read as [ks, x, h] depending on word and dialect.
- The realization of diphthongs such as *ie* [je] and *ue* [we] has not been addressed.
- No attempt has been made to handle stress assignment.

Furthermore, allophonic rules such as the lenition of *d* have been ignored. Such rules are needed to provide a narrow phonetic transcription, which may be useful in certain speech applications. An implementation of the above rules can be found in the `g2p` module, part of Pynini's `examples` library (Appendix D), and the interested reader might make improvements to address the limitations mentioned above.

5.5.2 FINNISH CASE SUFFIXES

Vowel harmony refers to a phonological process in which the properties of one vowel seemingly assimilate onto another. In some cases, the source and target vowel may be arbitrarily distant, and at first it might seem like this would make it difficult to encode using context-dependent rewrite rules. As shown below, this is not the case.

Finnish exhibits vowel harmony in several contexts, including in the distribution of **locative case suffixes**, which can be likened to English locational prepositions like *in* and *on*. Several of these suffixes, along with their traditional names and glosses, are listed below.

(17)

-tta/-ttä	abessive	'without'
-lta/-ltä	ablative	'off of'
-lla/-llä	adessive	'on'
-sta/-stä	elative	'out of'
-na/-nä	essive	'as a'
-ssa/-ssä	inessive	'in'

Each of the above suffixes above has two **allomorphs** (i.e., variants): one ending in *a* and one in *ä*. Their distribution is determined by the vowels of the adjective or noun to which they are attached. The back vowels—including *a*, *o*, and *u*—select the *a*-allomorph. The front rounded vowels—including *ä*, *ö*, and *y*—select the *ä*-allomorph. In the case that the stem contains both back and front round vowels, the harmonic vowel closest to the suffix governs harmony. Consonants and all other vowels are ignored for the purpose of harmony. Finally, the *ä*-allomorph is the default, used for those stems which contain neither back nor front rounded vowels. Several adessive case examples—some from Ringen and Heinämäki (1999)—are given below, with the harmony-governing stem vowel in bold.

(18)

ver**o**lla	'tax'
g**a**stilla	'sailor'
kes**y**llä	'tame'
k**ä**dellä	'hand'
veljellä	'brother'
vekillä	'pleat'

In his finite-state phonology of Finnish, Koskenniemi (1983:76f.) assumes that the harmonic suffixes contain an **archiphonemic** or **underspecified** vowel which is neither front or nor back. A series of surface-true rules map this archiphonemic vowel, written *A*, onto *a* or *ä*. It is somewhat easier yet to encode the harmony pattern using a cascade of two rewrite rules: the first maps *A* to *a* in the presence of a back-harmonic stem vowel, and the second maps remaining instances of *A* to *ä*. To implement this cascade of rules, one must first define the relevant segmental inventories, including the three classes of vowels.

```
back = pynini.union("a", "o", "u")
front = pynini.union("ä", "ö", "y")
neutral = pynini.union("e", "i")
v = pynini.union(back, front, neutral, "A")
c = pynini.union(
    "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
    "n", "p", "q", "r", "s", "t", "v", "w", "x", "z"
)
sigma_star = pynini.union(v, c).closure()
```

One can then use these to define the two harmony rules, which are composed and jointly optimized. In the first rule, the left context corresponds to an infinite language BN^* where B is the set of back-harmonic vowels and N the set of neutral segments; the second rule applies across the board.

```
harmony = (
    pynini.cdrewrite(
        pynini.cross("A", "a"),
        back + pynini.union(neutral, c).closure(),
        "",
        sigma_star
    ) @
    pynini.cdrewrite(
        pynini.cross("A", "ä"), "", "", sigma_star
    )
).optimize()
```

Given an input with the archiphonemic *A*, the cascade defined above applies the appropriate harmony pattern, as shown in the following snippet.

```
assert rewrite.one_top_rewrite("verollA", harmony) == "verolla"
assert rewrite.one_top_rewrite("kesyllA", harmony) == "kesyllä"
```

However, for some applications—such as morphological generation, the subject of the following chapter—it may be desirable to use this rule to help construct various inflectional forms of Finnish words. For instance, one could define a function which takes as input the citation form (the nominative singular) and generates the appropriate adessive form.

```
def adessive(cf: str) -> str:
    return rewrite.one_top_rewrite(cf + "llA", harmony)
```

An implementation of Finnish locative suffixes along these lines can be found in the `case` module, part of Pynini's `examples` library (Appendix D).

5.5.3 CURRENCY EXPRESSION TAGGING

Speech technologies like automatic speech recognition (ASR) and text-to-speech synthesis (TTS) often demand another type of mapping between written and spoken language. Speech recognizers and synthesizers usually operate over the "spoken domain", containing entities like *twelve pounds*, but not equivalent "written domain" expressions like *£12*, and speech systems which process or generate written-domain text (e.g., Pusateri et al. 2017, Shugrina 2010) must be able to map between the two domains. Sproat et al. (2001) refer to this process as **text normalization**, and Taylor (2009) refers to entities that require normalization—numbers, dates and times, currency, and measure expressions, abbreviations, etc.—as **semiotic classes**. Text normalization has traditionally been performed using finite-state transducers (e.g., Sproat 1996) although there have been many attempts to exploit machine learning for this task (see Zhang et al. 2019 for a recent review). Google's Kestrel text normalization engine, described in detail by Ebden and Sproat (2014), processes written-domain text in two stages. During the initial "tokenization and classification" stage, the various semiotic classes are identified in the text. Then, in the "verbalization" stage, semiotic class-specific finite-state transducer grammars are applied to each semiotic class span, producing spoken-domain text. The example here is a fragment of the tokenization and classification stage; the more challenging process of verbalization is illustrated in section 7.3.

Suppose one wishes to construct a finite-state transducer which is capable of "tagging" English currency expressions like *$4.59*, *€.80*, or *¥1400*. Such a transducer can naturally be expressed as an unconditioned context-dependent rewrite rule which matches any currency expression, inserting XML-style tags around the expression. Assume that such expressions consist of a currency symbol (e.g., *$*) or followed by either a sequence of Arabic digits (e.g., *140*) or a decimal to two places (e.g., *4.59*, *.80*). For simplicity, delimiters—like commas—used to group numbers are ignored. One begins by building acceptors that match the components of a currency expression so defined. First, the currency symbol acceptor matches the symbol for the dollar, euro, pound sterling, and yen, and can easily be expanded to handle other symbols.

```
cur_symbol = pynini.union("$", "€", "£", "¥")
```

The numeric portion of the expression is matched by `cur_numeric` below. This is defined using the constant `DIGIT` defined by the `byte` module, part of Pynini's extended library (Appendix C), which matches any one Arabic digit. This is used to define acceptors matching the "major" and "minor" portions of a currency expression.

```
major = byte.DIGIT ** (0, ...)
minor = byte.DIGIT ** (2, ...)
cur_numeric = major + ("." + minor) ** (0, 1)
```

We then concatenate the two expressions, producing an acceptor matching written currency expressions.

```
cur_exp = cur_symbol + cur_numeric
```

To turn this into a tagger automaton, we define τ, a transducer which matches the currency expression and inserts the XML-style tags. We then compile τ into a context-dependent rewrite rule using the closure over the vocabulary (here, bytes) as Σ^*, so that non-currency expression text is passed through unadulterated. This is illustrated in the following snippet.

```
tagger = pynini.cdrewrite(
    pynutil.insert("<cur>") + cur_exp + pynutil.insert("</cur>"),
    "",
    "",
    byte.BYTE ** (0, ...)
).optimize()
```

One can then use `tagger` to rewrite written-domain text with currency tags. However, this tagger automaton has many ambiguities. Consider for instance the input `I have £50`. One might expect this to be tagged `I have <cur>£50</cur>`, but the tagger defined above also permits `I have <cur>£5</cur>0`, in which the final `0` is left out of the expression. Let us assume that one would prefer a "greedy" tagging, in which the tags enclose the longest possible sequence.[11] One way to enforce this is to assign a negative weight to each Arabic digit matched, and then compute the shortest path output string, which will necessarily be the one with the longest possible sequence. Using `add_weight`, another helper function from the `pynutil` module, one can attach a weight to an FST. In the snippet below, this function is used to redefine the numeric components of the currency matcher.

```
major = pynutil.add_weight(byte.DIGIT, -1) ** (0, ...)
minor = pynutil.add_weight(byte.DIGIT, -1) ** (2, ...)
```

If these alternative weighted definitions are used to build `cur_exp` and `tagger` as above, then the shortest-path output string produced by applying the rule using `one_top_rewrite` will be greedy in the relevant sense.

The `tagger` module, part of Pynini's extended library (Appendix C), automates the construction and application of general-purpose tagger automata.

FURTHER READING

Bale and Reiss (2018) provide a detailed introduction to the SPE rule notation, conventions for rule application, phonological features, and natural classes.

Three of the early proponents of context-dependent rewrite rules—Ronald Kaplan, Lauri Karttunen, and Martin Kay—are recipients of the Association for Computational Linguistics'

[11] We note that it is not strictly necessary to use weights to achieve a greedy match. This can also be handled using, e.g., **directed replacement** (Karttunen 1996).

prestigious Lifetime Achievement Award, and all three discuss the import of this technology in the published transcriptions of their acceptance speeches (Kaplan 2019, Karttunen 2007, Kay 2005).

There have been a number of further refinements to rule compilation procedures since Kaplan and Kay 1994 and Mohri and Sproat 1996. See, for example Hetherington 2001, Skut et al. 2003, Yli-Jyrä and Koskenniemi 2006, Yli-Jyrä 2007, and Drobac et al. 2012.

Roark and Sproat (2007: ch. 4) review Koskenniemi's two-level model and its equivalence to rational relations.

Sproat (2000) argues that the process of grapheme-to-phoneme conversion can be described by a rational relation for all known writing systems; Ebden and Sproat (2014) make a similar claim for text normalization. Recent work by Jane Chandlee and colleagues, reviewed by Heinz (2018) and discussed in section 8.3 below, argues that nearly all known phonological and morphological processes are **subsequential functions** and can be implemented by deterministic finite-state transducers.

We have put aside the representation of tone and other prosodic-metrical phenomena; see Yli-Jyrä 2013 for some recent work on this topic.