Regular Models of Phonological Rule Systems

Ronald M. Kaplan*
Xerox Palo Alto Research Center

Martin Kay†
Xerox Palo Alto Research Center
and Stanford University

This paper presents a set of mathematical and computational tools for manipulating and reasoning about regular languages and regular relations and argues that they provide a solid basis for computational phonology. It shows in detail how this framework applies to ordered sets of context-sensitive rewriting rules and also to grammars in Koskenniemi's two-level formalism. This analysis provides a common representation of phonological constraints that supports efficient generation and recognition by a single simple interpreter.

1. Introduction

Ordered sets of context-sensitive rewriting rules have traditionally been used to describe the pronunciation changes that occur when sounds appear in different phonological and morphological contexts. Intuitively, these phenomena ought to be cognitively and computationally simpler than the variations and correspondences that appear in natural language syntax and semantics, yet the formal structure of such rules seems to require a complicated interpreter and an extraordinarily large number of processing steps. In this paper, we show that any such rule defines a *regular relation* on strings if its non-contextual part is not allowed to apply to its own output, and thus it can be modeled by a symmetric finite-state transducer. Furthermore, since regular relations are closed under serial composition, a finite set of rules applying to each other's output in an ordered sequence also defines a regular relation. A single finite-state transducer whose behavior simulates the whole set can therefore be constructed by composing the transducers corresponding to the individual rules. This transducer can be incorporated into efficient computational procedures that are far more economical in both recognition and production than any strategies using ordered rules directly. Since orthographic rules have similar formal properties to phonological rules, our results generalize to problems of word recognition in written text.

The mathematical techniques we develop to analyze rewriting rule systems are not limited just to that particular collection of formal devices. They can also be applied to other recently proposed phonological or morphological rule systems. For example, we can show that Koskenniemi's (1983) two-level parallel rule systems also denote regular relations. Section 2 below provides an intuitive grounding for the rest of our discussion by illustrating the correspondence between simple rewriting rules and transducers. Section 3 summarizes the mathematical tools that we use to analyze both rewriting and two-level systems. Section 4 describes the properties of the rewriting rule formalisms we are concerned with, and their mathematical characterization

* 3333 Coyote Hill Road, Palo Alto CA 94304. E-mail: kaplan@parc.xerox.com
† 3333 Coyote Hill Road, Palo Alto CA 94304. E-mail: kay@parc.xerox.com

is presented in Sections 5 and 6. A similar characterization of two-level rule systems is provided in Section 7.

By way of introduction, we consider some of the computational issues presented by simple morphophonemic rewriting rules such as these:

Rule 1
N → m / ___ [+labial]

Rule 2
N → n

According to these rules an underspecified, abstract nasal phoneme *N* appearing in the lexical forms *iNpractical* and *iNtractable* will be realized as the *m* in *impractical* and as the *n* in *intractable*. To ensure that these and only these results are obtained, the rules must be treated as obligatory and taken in the order given. As obligatory rules, they must be applied to every substring meeting their conditions. Otherwise, the abstract string *iNpractical* would be realized as *inpractical* and *iNpractical* as well as *impractical*, and the abstract *N* would not necessarily be removed from *iNtractable*. Ordering the rules means that the output of the first is taken as the input to the second. This prevents *iNpractical* from being converted to *inpractical* by Rule 2 without first considering Rule 1.

These obligatory rules always produce exactly one result from a given input. This is not the case when they are made to operate in the reverse direction. For example, if Rule 2 is inverted on the string *intractable*, there will be two results, *intractable* and *iNtractable*. This is because *intractable* is derivable by that rule from both of these strings. Of course, only the segments in *iNtractable* will eventually match against the lexicon, but in general both the *N* and *n* results of this inversion can figure in valid interpretations. Compare the words *undecipherable* and *indecipherable*. The *n* in the prefix *un-*, unlike the one in *in-*, does not derive from the abstract *N*, since it remains unchanged before labials (c.f. *unperturbable*). Thus the results of inverting this rule must include *undecipherable* for *undecipherable* but *iNdecipherable* for *indecipherable* so that each of them can match properly against the lexicon.

While inverting a rule may sometimes produce alternative outputs, there are also situations in which no output is produced. This happens when an obligatory rule is inverted on a string that it could not have generated. For example, *iNput* cannot be generated by Rule 1 because the *N* precedes a labial and therefore would obligatorily be converted to *m*. There is therefore no output when Rule 1 is inverted on *iNput*. However, when Rule 2 is inverted on *input*, it does produce *iNput* as one of its results. The effect of then inverting Rule 1 is to remove the ambiguity produced by inverting Rule 2, leaving only the unchanged *input* to be matched against the lexicon. More generally, if recognition is carried out by taking the rules of a grammar in reverse order and inverting each of them in turn, later rules in the new sequence act as filters on ambiguities produced by earlier ones.

The existence of a large class of ambiguities that are introduced at one point in the recognition process and eliminated at another has been a major source of difficulty in efficiently reversing the action of linguistically motivated phonological grammars. In a large grammar, the effect of these spurious ambiguities is multiplicative, since the information needed to cut off unproductive paths often does not become available until after they have been pursued for some considerable distance. Indeed, speech understanding systems that use phonological rules do not typically invert them on strings but rather apply them to the lexicon to generate a list of all possible word forms (e.g. Woods et al. 1976; Klatt 1980). Recognition is then accomplished by standard table-

lookup procedures, usually augmented with special devices to handle phonological changes that operate across word boundaries. Another approach to solving this computational problem would be to use the reversed cascade of rules during recognition, but to somehow make the filtering information of particular rules available earlier in the process. However, no general and effective techniques have been proposed for doing this.

The more radical approach that we explore in this paper is to eliminate the cascade altogether, representing the information in the grammar as a whole in a single more unified device, namely, a finite-state transducer. This device is constructed in two phases. The first is to create for each rule in the grammar a transducer that exactly models its behavior. The second is to compose these individual rule transducers into a single machine that models the grammar as a whole.

Johnson (1972) was the first to notice that the noncyclic components of standard phonological formalisms, and particularly the formalism of *The Sound Pattern of English* (Chomsky and Halle 1968), were equivalent in power to finite-state devices despite a superficial resemblance to general rewriting systems. Phonologists in the SPE tradition, as well as the structuralists that preceded them, had apparently honored an injunction against rules that rewrite their own output but still allowed the output of a rule to serve as context for a reapplication of that same rule. Johnson realized that this was the key to limiting the power of systems of phonological rules. He also realized that basic rewriting rules were subject to many alternative modes of application offering different expressive possibilities to the linguist. He showed that phonological grammars under most reasonable modes of application remain within the finite-state paradigm.

We observed independently the basic connections between rewriting-rule grammars and finite-state transducers in the late 1970s and reported them at the 1981 meeting of the Linguistic Society of America (Kaplan and Kay 1981). The mathematical analysis in terms of regular relations emerged somewhat later. Aspects of that analysis and its extension to two-level systems were presented at conferences by Kaplan (1984, 1985, 1988), in courses at the 1987 and 1991 Linguistics Institutes, and at colloquia at Stanford University, Brown University, the University of Rochester, and the University of Helsinki.

Our approach differs from Johnson's in two important ways. First, we abstract away from the many details of both notation and machine description that are crucial to Johnson's method of argumentation. Instead, we rely strongly on closure properties in the underlying algebra of regular relations to establish the major result that phonological rewriting systems denote such sets of string-pairs. We then use the correspondence between regular relations and finite-state transducers to develop a constructive relationship between rewriting rules and transducers. This is accomplished by means of a small set of simple operations, each of which implements a simple mathematical fact about regular languages, regular relations, or both. Second, our more abstract perspective provides a general framework within which to treat other phonological formalisms, existing or yet to be devised. For example, two-level morphology (Koskenniemi 1983), which evolved from our early considerations of rewriting rules, relies for its analysis and implementation on the same algebraic techniques. We are also encouraged by initial successes in adapting these techniques to the autosegmental formalism described by Kay (1987).

2. Rewriting Rules and Transducers

Supposing for the moment that Rule 2 ($N \rightarrow n$) is optional, Figure 1 shows the transition diagram of a finite-state transducer that models it. A finite-state transducer has

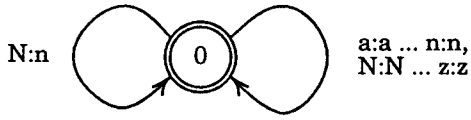


Figure 1
Rule 2 as optional.

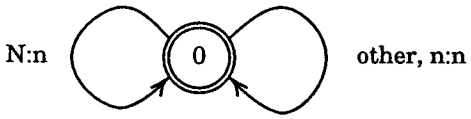


Figure 2
Rule 2 as obligatory.

two tapes. A transition can be taken if the two symbols separated by the colon in its label are found at the current position on the corresponding tapes, and the current position advances across those tape symbols. A pair of tapes is accepted if a sequence of transitions can be taken starting at the start-state (conventionally labeled 0) and at the beginning of the tapes and leading to a final state (indicated by double circles) at the end of both tapes. In the machine in Figure 1, there is a transition from state 0 to state 0 that translates every phoneme into itself, reflecting the fact that any phoneme can remain unchanged by the optional rule. These are shown schematically in the diagram. This machine will accept a pair of tapes just in case they stand in a certain relation: they must be identical except for possible replacements of N on the first tape with n on the second. In other words, the second tape must be one that could have resulted from applying the optional rule to the string on the first tape.

But the rule is in fact obligatory, and this means that there must be no occurrences of N on the second tape. This condition is imposed by the transducer in Figure 2. In this diagram, the transition label "other" abbreviates the set of labels $a{:}a, b{:}b, \ldots z{:}z$, the identity pairs formed from all symbols that belong to the alphabet but are not mentioned explicitly in this particular rule. This diagram shows no transition over the pair $N{:}N$ and the transducer therefore *blocks* if it sees N on both tapes. This is another abbreviatory convention that is typically used in implementations to reduce transducer storage requirements, and we use it here to simplify the state diagrams we draw. In formal treatments such as the one we present below, the transition function is total and provides for transitions from every state over every pair of symbols. Any transition we do not show in these diagrams in fact terminates at a single nonfinal state, the "failure" state, which we also do not show.

Figure 3 is the more complicated transducer that models the obligatory behavior of Rule 1 ($N \rightarrow m /$ ___ $+[labial]$). This machine blocks in state 1 if it sees the pair $N{:}m$ not followed by one of the labials p, b, m. It blocks in state 2 if it encounters the pair $N{:}N$ followed by a labial on both tapes, thus providing for the situation in which the rule is not applied even though its conditions are satisfied. If it does not block and both tapes are eventually exhausted, it accepts them just in case it is then in one of the *final* states, 0 or 2, shown as double circles. It rejects the tapes if it ends up in the nonfinal state 1, indicating that the second tape is not a valid translation of the first one.

We have described transducers as acceptors of pairs of tapes that stand in a certain relation. But they can also be interpreted asymmetrically, as functions either from

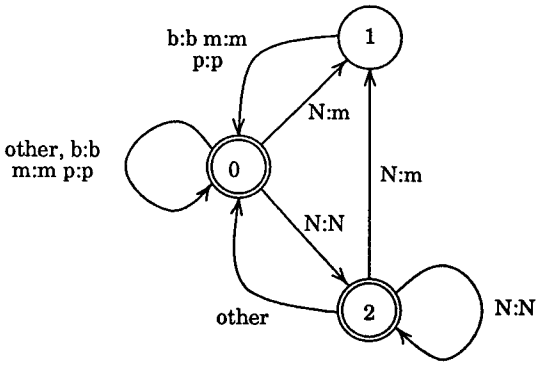


Figure 3
Rule 1 as obligatory.

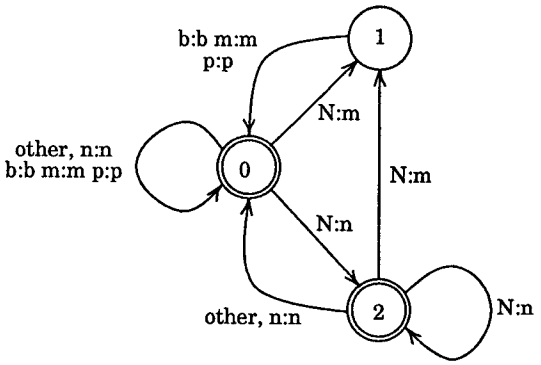


Figure 4
Composition of obligatory Rules 1 and 2.

more abstract to less abstract strings or the other way around. Either of the tapes can contain an input string, in which case the output will be written on the other. In each transition the machine matches the symbol specified for the input tape and writes the one for the output. When the first tape contains the input, the machine models the generative application of the rule; when the second tape contains the input, it models the inversion of the rule. Thus, compared with the rewriting rules from which they are derived, finite-state transducers have the obvious advantage of formal and computational simplicity. Whereas the exact procedure for inverting rules themselves is not obvious, it is clearly different from the procedure required for generating. The corresponding transducers, on the other hand, have the same straightforward interpretation in both directions.

While finite-state transducers are attractive for their formal simplicity, they have a much more important advantage for our purposes. A pair of transducers connected through a common tape models the composition of the relations that those transducers represent. The pair can be regarded as performing a transduction between the outer tapes, and it turns out that a single finite-state transducer can be constructed that performs exactly this transduction without incorporating any analog of the intermediate tape. In short, the relations accepted by finite-state transducers are closed under serial composition. Figure 4 shows the composition of the *m*-machine in Figure 3 and the *n*-machine in Figure 2. This transducer models the cascade in which the output of Rule 1 is the input to Rule 2.

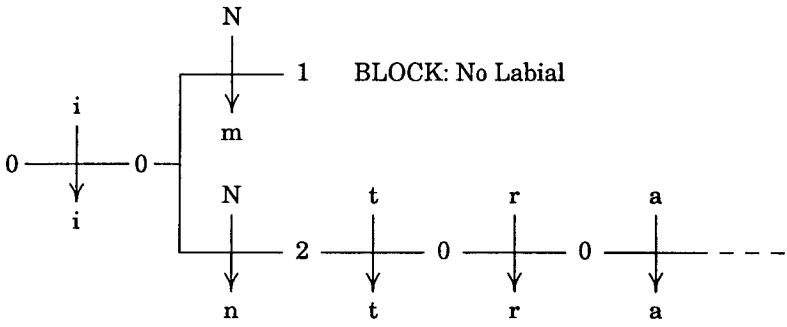


Figure 5
Generation of *intractable*.

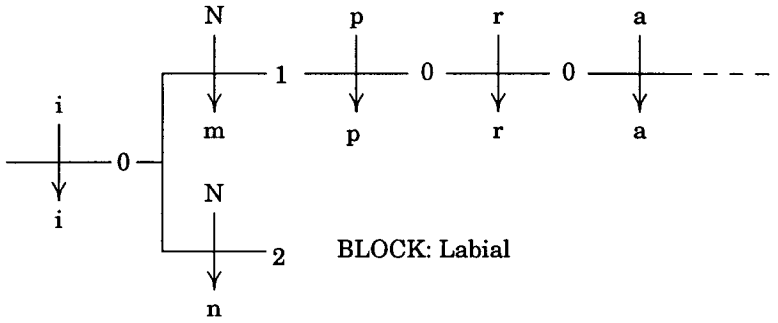


Figure 6
Generation of *impractical*.

This machine is constructed so that it encodes all the possible ways in which the *m*-machine and *n*-machine could interact through a common tape. The only interesting interactions involve *N*, and these are summarized in the following table:

input	*m*-machine	output		
		input	*n*-machine	output
N	*labial follows*	m		m
N	*nonlabial follows*	N		n

An *N* in the input to the *m*-machine is converted to *m* before a labial and this *m* remains unchanged by the *n*-machine. The only instances of *N* that reach the *n*-machine must therefore be followed by nonlabials and these must be converted to *n*. Accordingly, after converting *N* to *m*, the composed machine is in state 1, which it can leave only by a transition over labials. After converting *N* to *n*, it enters state 2, from which there is no labial transition. Otherwise, state 2 is equivalent to the initial state.

Figure 5 illustrates the behavior of this machine as a generator applied to the abstract string *iNtractable*. Starting in state 0, the first transition over the "other" arc produces *i* on the output tape and returns to state 0. Two different transitions are then possible for the *N* on the input tape. These carry the machine into states 1 and 2 and output the symbols *m* and *n* respectively. The next symbol on the input tape is *t*. Since this is not a labial, no transition is possible from state 1, and that branch of the process therefore blocks. On the other branch, the *t* matches the "other" transition back to state 0 and the machine stays in state 0 for the remainder of the string. Since state 0

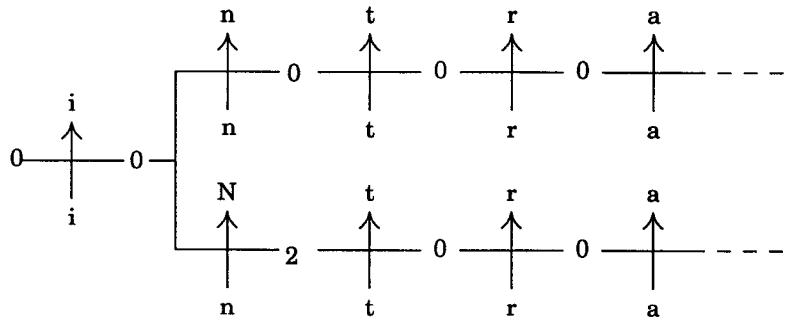


Figure 7
Recognition of *intractable*.

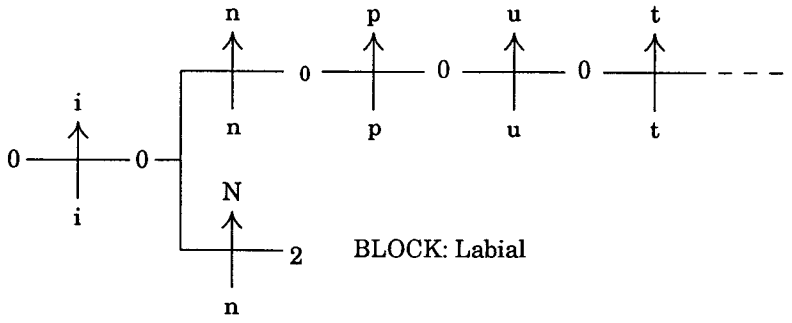


Figure 8
Recognition of *input*.

is a final state, this is a valid derivation of the string *intractable*. Figure 6 is a similar representation for the generation of *impractical*.

Figures 7 and 8 illustrate this machine operating as a recognizer. As we pointed out earlier, there are two results when the cascade of rules that this machine represents is inverted on the string *intractable*. As Figure 7 shows, the *n* can be mapped into *n* by the *n*:*n* transition at state 0 or into *N* by the transition to state 2. The latter transition is acceptable because the following *t* is not a labial and thus matches against the "other" transition to state 0. When the following symbol is a labial, as in Figure 8, the process blocks. Notice that the string *iNput* that would have been written on the intermediate tape before the machines were composed is blocked after the second symbol by constraints coming from the *m*-machine.

Repeated composition reduces the machines corresponding to the rules of a complete phonological grammar to a single transducer that works with only two tapes, one containing the abstract phonological string and the other containing its phonetic realization. General methods for constructing transducers such as these rely on fundamental mathematical notions that we develop in the next section.

3. Mathematical Concepts and Tools

Formal languages are sets of strings, mathematical objects constructed from a finite alphabet Σ by the associative operation of concatenation. Formal language theory has classified string sets, the subsets of Σ^*, in various ways and has developed corre-

spondences between languages, grammatical notations for describing their member strings, and automata for recognizing them. A similar conceptual framework can be established for *string relations*. These are the collections of ordered tuples of strings, the subsets of $\Sigma^* \times \ldots \times \Sigma^*$.

We begin by defining an n-way concatenation operation in terms of the familiar concatenation of simple strings. If $X = \langle x_1, x_2, \ldots x_n \rangle$ and $Y = \langle y_1, y_2, \ldots y_n \rangle$ are n-tuples of strings, then the concatenation of X and Y, written $X \cdot Y$ or simply XY, is defined by

$$X \cdot Y =_{df} \langle x_1 y_1, x_2 y_2, \ldots x_n y_n \rangle$$

That is, the n-way concatenation of two string-tuples is the tuple of strings formed by string concatenation of corresponding elements. The length of a string-tuple $|X|$ can be defined in terms of the lengths of its component strings:

$$|X| =_{df} \sum_i |x_i|$$

This has the expected property that $|X \cdot Y| = |X| + |Y|$, even if the elements of X or of Y are of different lengths. Just as the empty string ϵ is the identity for simple string concatenation, the n-tuple all of whose elements are ϵ is the identity for n-way concatenation, and the length of such a tuple is zero.

3.1 Regular Relations and Finite-State Transducers

With these definitions in hand, it is immediately possible to construct families of string relations that parallel the usual classes of formal languages. Recall, for example, the usual recursive definition of a regular language over an alphabet Σ (superscript i denotes concatenation repeated i times, according to the usual convention, and we let Σ^ϵ denote $\Sigma \cup \{\epsilon\}$):

1. The empty set and $\{a\}$ for all a in Σ^ϵ are regular languages.
2. If L_1, L_2, and L are regular languages, then so are
   $L_1 \cdot L_2 = \{xy \mid x \in L_1, y \in L_2\}$ (concatenation)
   $L_1 \cup L_2$ (union)
   $L^* = \bigcup_{i=0}^{\infty} L^i$ (Kleene closure)
3. There are no other regular languages.

We can use exactly the same scheme to define regular n-relations in terms of n-way concatenation:

1. The empty set and $\{a\}$ for all a in $\Sigma^\epsilon \times \ldots \times \Sigma^\epsilon$ are regular n-relations.
2. If R_1, R_2, and R are regular n-relations, then so are
   $R_1 \cdot R_2 = \{xy \mid x \in R_1, y \in R_2\}$ (n-way concatenation)
   $R_1 \cup R_2$ (union)
   $R^* = \bigcup_{i=0}^{\infty} R^i$ (n-way Kleene closure)
3. There are no other regular n-relations.

Other families of relations can also be defined by analogy to the formal language case. For example, a system of context-free rewriting rules can be used to define a context-free n-relation simply by introducing n-tuples as the terminal symbols of the grammar. The standard context-free derivation procedure will produce tree structures with n-tuple leaves, and the relational yield of such a grammar is taken to be the set of n-way concatenations of these leaves. Our analysis of phonological rule systems does not depend on expressive power beyond the capacity of the regular relations, however, and we therefore confine our attention to the mathematical and computational properties of these more limited systems. The relations we refer to as "regular," to emphasize the connection to formal language theory, are often known as "rational relations" in the algebraic literature, where they have been extensively studied (e.g. Eilenberg 1974).

The descriptive notations and accepting automata for regular languages can also be generalized to the n-dimensional case. An n-way regular expression is simply a regular expression whose terms are n-tuples of alphabetic symbols or ϵ. For ease of writing we separate the elements of an n-tuple by colons. Thus the expression $a{:}b\ \epsilon{:}c$ describes the two-relation containing the single pair $\langle a, bc\rangle$, and $a{:}b{:}c^*\ q{:}r{:}s$ describes the three-relation $\{\langle a^n q, b^n r, c^n s\rangle \mid n \geq 0\}$. The regular-expression notation provides for concatenation, union, and Kleene-closure of these terms. The accepting automata for regular n-relations are the n-way finite-state transducers. As illustrated by the two-dimensional examples given in Section 2, these are an obvious extension of the standard one-tape finite-state machines.

The defining properties of the regular languages, regular expressions, and finite-state machines are the basis for proving the well-known Kleene correspondence theorems showing the equivalence of these three string-set characterizations. These essential properties carry over in the n-way generalizations, and therefore the correspondence theorems also generalize. In particular, simple analogs of the standard inductive proofs show that

> Every n-way regular expression describes a regular n-relation;
> Every regular n-relation is described by an n-way regular expression;
> Every n-tape finite-state transducer accepts a regular n-relation; and
> Every regular n-relation is accepted by an n-tape finite-state transducer.

The strength of our analysis method comes from the equivalence of these different characterizations. While we reason about the regular relations in algebraic and set-theoretic terms, we conveniently describe the sets under discussion by means of regular expressions, and we prove essential properties by constructive operations on the corresponding finite-state transducers. In the end, of course, it is the transducers that satisfy our practical, computational goals.

A nondeterministic (one-tape) finite-state machine is a quintuple $(\Sigma, Q, q, F, \delta)$, where Σ is a finite alphabet, Q is a finite set of states, $q \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. The transition function δ is a total function that maps $Q \times \Sigma^\epsilon$ to 2^Q, the set of all subsets of Q, and every state s in Q is vacuously a member of $\delta(s, \epsilon)$. We extend the function δ to sets of states, so that for any $P \subseteq Q$ and $a \in \Sigma^\epsilon$, $\delta(P, a) = \cup_{p \in P}\, \delta(p, a)$. We also define the usual extension of δ to a transition function δ^* on Σ^* as follows: for all r in Q, $\delta^*(r, \epsilon) = \delta(r, \epsilon)$ and for all $u \in \Sigma^*$ and $a \in \Sigma^\epsilon$, $\delta^*(r, ua) = \delta(\delta^*(r, u), a)$. Thus, the machine accepts a string x just in case $\delta^*(q, x) \cap F$ is nonempty; that is, if there is a sequence of transitions over x beginning at the initial

state and ending at a set of states at least one of which is final. We know, of course, that every regular language is also accepted by a deterministic, ϵ-free finite-state machine, but assuming vacuous ϵ transitions at every state reduces the number of special cases that have to be considered in some of the arguments below.

A nondeterministic n-way finite-state transducer (fst) is defined by a quintuple similar to that of an fsm except for the transition function δ, a total function that maps $Q \times \Sigma^\epsilon \times \ldots \times \Sigma^\epsilon$ to 2^Q. Partly to simplify the mathematical presentation and partly because only the binary relations are needed in the analysis of rewriting rules and Koskenniemi's two-level systems, from here on we frame the discussion in terms of binary relations and two-tape transducers. However, the obvious extensions of these properties do hold for the general case, and they may be useful in developing a formal understanding of autosegmental phonological and morphological theories (for an illustration, see Kay 1987).

The transition function δ of a transducer also extends to a function δ^* that carries a state and a pair of strings onto a set of states. Transitions in fsts are labeled with pairs of symbols and we continue to write them with a colon separator. Thus, $u{:}v$ labels a transition over a u on the first tape and a v on the second. A finite-state transducer T defines the regular relation $R(T)$, the set of pairs $\langle x, y \rangle$ such that $\delta^*(q, x, y)$ contains a final state. The pair $\epsilon{:}\epsilon$ plays the same role as a label of transducer transitions that the singleton ϵ plays in one-tape machines, and the ϵ-removal algorithm for one-tape machines can be generalized to show that every regular relation is accepted by an $\epsilon{:}\epsilon$-free transducer. However, it will also be convenient for some arguments below to assume the existence of vacuous $\epsilon{:}\epsilon$ transitions.

We write xRy if the pair $\langle x, y \rangle$ belongs to the relation R. The image of a string x under a relation R, which we write x/R, is the set of strings y such that $\langle x, y \rangle$ is in R. Similarly, R/y is the set of strings that R carries onto y. We extend this notation to sets of strings in the obvious way: $X/R = \cup_{x \in X} x/R$. This relational notation gives us a succinct way of describing the use of a corresponding transducer as either a generator or a recognizer. For example, if R is the regular relation recognized by the transducer in Figure 4, then $R/$*intractable* is the set of strings that R maps to *intractable*, namely {*intractable*, *iNtractable*}, as illustrated in Figure 7. Similarly, *iNtractable*/R is the set of strings {*intractable*} that R maps from *iNtractable* (Figure 5).

We rely on the equivalence between regular languages and relations and their corresponding finite-state automata, and we frequently do not distinguish between them. When the correspondence between a language L and its equivalent machine must be made explicit, we let $M(L)$ denote a finite-state machine that accepts L. Similarly, we let $T(R)$ denote a transducer that accepts the relation R, as provided by the correspondence theorem. We also rely on several of the closure properties of regular languages (Hopcroft and Ullman 1979): for regular languages L_1 and L_2, L_1L_2 is the regular language containing all strings x_1x_2 such that $x_1 \in L_1$ and $x_2 \in L_2$. We use superscripts for repeated concatenation: L^n contains the concatenation of n members of L, and L^* contains strings with arbitrary repetitions of strings in L, including zero. The operator *Opt* is used for optionality, so that $Opt(L)$ is $L \cup \{\epsilon\}$. We write $\overline{L}$ for the complement of L, the regular language containing all strings not in L, namely, $\Sigma^* - L$. Finally, $Rev(L)$ denotes the regular language consisting of the reversal of all the strings in L.

3.2 Properties of Regular Relations

There are a number of basic connections between regular relations and regular languages. The strings that can occur in the domain and range of a regular relation R

($Dom(R) = R/\Sigma^*$ and $Range(R) = \Sigma^*/R$) are the regular languages accepted by the finite-state machines derived from $T(R)$ by changing all transition labels $a{:}b$ to a and b respectively, for all a and b in Σ^ϵ. Given a regular language L, the identity relation $Id(L)$ that carries every member of L into itself is regular; it is characterized by the fst obtained from an fsm $M(L)$ by changing all transition labels a to $a{:}a$. Clearly, for all languages L, $L = Dom(Id(L)) = Range(Id(L))$. The inverse R^{-1} of a regular relation R is regular, since it is accepted by a transducer formed from $T(R)$ by changing all labels $a{:}b$ to $b{:}a$. The reversal $Rev(R)$, consisting of pairs containing the reversal of strings in R's pairs, is also regular; its accepting transducer is derived from $T(R)$ by generalizing the standard one-tape fsm construction for regular language reversal.

Given a pair of regular languages L_1 and L_2 whose alphabets can, without loss of generality, be assumed equal, the relation $L_1 \times L_2$ containing their Cartesian product is regular. To prove this proposition, we let $M_1 = (\Sigma, Q_1, q_1, F_1, \delta_1)$ and $M_2 = (\Sigma, Q_2, q_2, F_2, \delta_2)$ be fsms accepting L_1 and L_2 respectively and define the fst

$$T = (\Sigma, Q_1 \times Q_2, \langle q_1, q_2\rangle, F_1 \times F_2, \delta)$$

where for any $s_1 \in Q_1$, $s_2 \in Q_2$ and $a, b \in \Sigma^\epsilon$

$$\delta(\langle s_1, s_2\rangle, a, b) = \delta_1(s_1, a) \times \delta_2(s_2, b)$$

We can show by induction on the number of transitions that for any strings x and y,

$$\delta^*(\langle q_1, q_2\rangle, x, y) = \delta_1^*(q_1, x) \times \delta_2^*(q_2, y)$$

This result holds trivially when x and y are both ϵ by the general definition of δ^*. If a and b are in Σ^ϵ and u and v are in Σ^*, then, using the definition of δ^* and the definition just given for δ of the Cartesian product machine, we have

$$\begin{aligned}
\delta^*(\langle q_1, q_2\rangle, ua, vb) &= \delta(\delta^*(\langle q_1, q_2\rangle, u, v), a, b) \\
&= \delta(\delta_1^*(q_1, u) \times \delta_2^*(q_2, v), a, b) && \text{by induction} \\
&= \delta_1(\delta_1^*(q_1, u), a) \times \delta_2(\delta_2^*(q_2, v), b) \\
&= \delta_1^*(q_1, ua) \times \delta_2^*(q_2, vb)
\end{aligned}$$

Thus, $\delta^*(\langle q_1, q_2\rangle, x, y)$ contains a final state if and only if both $\delta_1^*(q_1, x)$ and $\delta_2^*(q_2, y)$ contain final states, so T accepts exactly the strings in $L_1 \times L_2$. □

Note that $L \times L$ is not the same as $Id(L)$, because only the former can map one member of L onto a different one. If L contains the single-character strings a and b, then $Id(L)$ only contains the pairs $\langle a, a\rangle$ and $\langle b, b\rangle$ while $L \times L$ also contains $\langle a, b\rangle$ and $\langle b, a\rangle$.

A similar construction is used to prove that regular relations are closed under the composition operator discussed in Section 2. A pair of strings $\langle x, y\rangle$ belongs to the relation $R_1 \circ R_2$ if and only if for some intermediate string z, $\langle x, z\rangle \in R_1$ and $\langle z, y\rangle \in R_2$. If $T(R_1) = (\Sigma, Q_1, q_1, F_1, \delta_1)$ and $T(R_2) = (\Sigma, Q_2, q_2, F_2, \delta_2)$, the composition $R_1 \circ R_2$ is accepted by the composite fst

$$(\Sigma, Q_1 \times Q_2, \langle q_1, q_2\rangle, F_1 \times F_2, \delta)$$

where

$$\delta(\langle s_1, s_2\rangle, a, b) = \{\langle t_1, t_2\rangle \mid \text{for some } c \in \Sigma^\epsilon, t_1 \in \delta(s_1, a, c) \text{ and } t_2 \in \delta(s_2, c, b)\}$$

In essence, the δ for the composite machine is formed by canceling out the intermediate tape symbols from corresponding transitions in the component machines. By an induction on the number of transitions patterned after the one above, it follows that for any strings x and y,

$$\delta^*(\langle q_1, q_2\rangle, x, y) = \{\langle t_1, t_2\rangle \mid \text{for some } z \in \Sigma^*, t_1 \in \delta_1^*(q_1, x, z) \text{ and } t_2 \in \delta_2^*(q_2, z, y)\}$$

The composite transducer enters a final state just in case both component machines do for some intermediate z. This establishes that the composite transducer does represent the composition of the relations R_1 and R_2, and that the composition of two regular relations is therefore regular. Composition of regular relations, like composition of relations in general, is associative: $(R_1 \circ R_2) \circ R_3 = R_1 \circ (R_2 \circ R_3) = R_1 \circ R_2 \circ R_3$. For relations in general we also know that $Range(R_1 \circ R_2) = Range(R_1)/R_2$.

We can use this fact about the range of a composition to prove that the image of a regular language under a regular relation is a regular language. (It is well known that the images under a regular relation of languages in other classes, for example the context-free languages, also remain within those classes (e.g. Harrison 1978), but these other results do not concern us here.) That is, if L is a regular language and R is an arbitrary regular relation, then the languages L/R and R/L are both regular. If L is a regular language, we know there exists a regular relation $Id(L)$ that takes all and only members of L into themselves. Since $L = Range(Id(L))$ it follows that

$$\begin{aligned} L/R &= (Range(Id(L)))/R \\ &= Range(Id(L) \circ R) \end{aligned}$$

$Id(L) \circ R$ is regular and we have already observed that the range of any regular relation is a regular language. By symmetry of argument we know that R/L is also regular.

Just like the class of regular languages, the class of regular relations is by definition closed under the operations of union, concatenation, and repeated concatenation. Also, the Pumping Lemma for regular languages immediately generalizes to regular relations, given the definitions of string-tuple length and n-way concatenation and the correspondence to finite-state transducers. The regular relations differ from the regular languages, however, in that they are not closed under intersection and complementation. Suppose that R_1 is the relation $\{\langle a^n, b^n c^*\rangle \mid n \geq 0\}$ and R_2 is the relation $\{\langle a^n, b^* c^n\rangle \mid n \geq 0\}$. These relations are regular, since they are defined by the regular expressions $a{:}b^*\ \epsilon{:}c^*$ and $\epsilon{:}b^*\ a{:}c$ respectively. The intersection $R_1 \cap R_2$ is $\{\langle a^n, b^n c^n\rangle \mid n \geq 0\}$. The range of this relation is the context-free language $b^n c^n$, which we have seen is not possible if the intersection is regular. The class of regular relations is therefore not closed under intersection, and it immediately follows that it is also not closed under complementation: by De Morgan's law, closure under complementation and union would imply closure under intersection. Nonclosure under complementation further implies that some regular relations are accepted by only nondeterministic transducers. If for every regular relation there is a deterministic acceptor, then the standard technique (Hopcroft and Ullman 1979) of interchanging its final and nonfinal states could be used to produce an fst accepting the complement relation, which would therefore be regular.

3.3 Same-Length Regular Relations

Closure under intersection and relative difference, however, are crucial for our treatment of two-level rule systems in Section 7. But these properties are required only for the *same-length* regular relations, and it turns out that this subclass is closed in the necessary ways. The same-length relations contain only string-pairs $\langle x, y\rangle$ such that

the length of x is the same as the length of y. It may seem obvious that the relevant closure properties do hold for this subclass, but for the sake of completeness we sketch the technical details of the constructions by which they can be established.

We make use of some auxiliary definitions regarding the *path-language* of a transducer. A path-string for any finite-state transducer T is a (possibly empty) sequence of symbol-pairs $u_1{:}v_1\ u_2{:}v_2 \ldots u_n{:}v_n$ that label the transitions of an accepting path in T. The path-language of T, notated as $Paths(T)$, is simply the set of all path-strings for T. $Paths(T)$ is obviously regular, since it is accepted by the finite-state machine constructed simply by interpreting the transition labels of T as elements of an alphabet of unanalyzable pair-symbols. Also, if P is a finite-state machine that accepts a pair-symbol language, we define the path-relation $Rel(P)$ to be the relation accepted by the fst constructed from P by reinterpreting every one of its pair-symbol labels as the corresponding symbol pair of a transducer label. It is clear for all fsts T that $Rel(M(Paths(T))) = R(T)$, the relation accepted by T.

Now suppose that R_1 and R_2 are regular relations accepted by the transducers T_1 and T_2, respectively, and note that $Paths(T_1) \cap Paths(T_2)$ is in fact a regular language of pair-symbols accepted by some fsm P. Thus $Rel(P)$ exists as a regular relation. Moreover, it is easy to see that $Rel(P) \subseteq R_1 \cap R2$. This is because every string-pair belonging to the path-relation is accepted by a transducer with a path-string that belongs to the path-languages of both T_1 and T_2. Thus that pair also belongs to both R_1 and R_2.

The opposite containment does not hold of arbitrary regular relations. Suppose a pair $\langle x, y\rangle$ belongs to both R_1 and R_2 but that none of its accepting paths in T_1 has the same sequence of transition labels as an accepting path in T_2. Then there is no path in $Paths(T_1) \cap Paths(T_2)$ corresponding to this pair and it is therefore not contained in $Rel(P)$. This situation can arise when the individual transducers have transitions with ϵ-containing labels. One transducer may then accept a particular string pair through a sequence of transitions that does not literally match the transition sequence taken by the other on that same pair of strings. For example, the first fst might accept the pair $\langle ab, c\rangle$ by the transition sequence $a{:}\epsilon\ \ b{:}c$, while the other accepts that same pair with the sequence $a{:}c\ \ b{:}\epsilon$. This string-pair belongs to the intersection of the relations, but unless there is some other accepting path common to both machines, it will not belong to $Rel(P)$. Indeed, when we apply this construction to fsts accepting the relations we used to derive the context-free language above, we find that $Rel(P)$ is the empty relation (with no string-pairs at all) instead of the set-theoretic intersection $R_1 \cap R_2$.

However, if R_1 and R_2 are accepted by transducers none of whose accepting paths have ϵ-containing labels, then a string-pair belonging to both relations will be accepted by identically labeled paths in both transducers. The language $Paths(T_1) \cap Paths(T_2)$ will contain a path-string corresponding to that pair, that pair will belong to $Rel(P)$, and $Rel(P)$ will be exactly $R_1 \cap R_2$. Thus, we complete the proof that the same-length relations are closed under intersection by establishing the following proposition:

Lemma
R is a same-length regular relation if and only if it is accepted by an ϵ-free finite-state transducer.

Proof
The transitions of an ϵ-free transducer T set the symbols of the string-pairs it accepts in one-to-one correspondence, so trivially, $R(T)$ is same-length. The proof in the other direction is more tedious. Suppose R is a same-length regular relation accepted by some transducer T which has transitions of the form $u{:}\epsilon$ or $\epsilon{:}v$ (with u and v not ϵ;

we know all $\epsilon{:}\epsilon$ transitions can be eliminated by the obvious generalization of the one-tape ϵ-removal algorithm). We systematically remove all ϵ-containing transitions in a finite sequence of steps each of which preserves the accepted relation. A path from the start-state to a given nonfinal state will contain some number of $u{:}\epsilon$ transitions and some number of $\epsilon{:}v$ transitions, and those two numbers will not necessarily be identical. However, for all paths to that state the difference between those numbers will be the same, since the discrepancy must be reversed by each path that leads from that state to a final state. Let us define the *imbalance* characterizing a state to be the difference in the number of $u{:}\epsilon$ and $\epsilon{:}v$ transitions on paths leading to that state. Since an acyclic path cannot produce an imbalance that differs from zero by more than the number of states in the machine, the absolute value of the imbalance is bounded by the machine size. On each iteration our procedure has the effect of removing all states with the maximum imbalance. First, we note that transitions of the form $u{:}v$ always connect a pair of states with the same imbalance. Such transitions can be eliminated in favor of an equivalent sequence of transitions $\epsilon{:}v$ and $u{:}\epsilon$ through a new state whose imbalance is one less than the imbalance of the original two states. Now suppose that $k > 0$ is the maximum imbalance for the machine and that all $u{:}v$ transitions between states of imbalance k have been eliminated. If q is a k-imbalance state, it will be entered only by $u{:}\epsilon$ transitions from $k-1$ states and left only by $\epsilon{:}v$ transitions also to $k-1$ states. For all transitions $u{:}\epsilon$ from a state p to q and all transitions $\epsilon{:}v$ from q to r, we construct a new transition $u{:}v$ from p to r. Then we remove state q from the machine along with all transitions entering or leaving it. These manipulations do not change the accepted relation but do reduce by one the number of k-imbalance states. We repeat this procedure for all k states and then move on to the $k-1$ states, continuing until no states remain with a positive imbalance. A symmetric procedure is then used to eliminate all the states whose imbalance is negative. In the end, T will have been transformed to an ϵ-free transducer that still accepts R. □

The same-length regular relations are obviously closed under union, concatenation, composition, inverse, and reverse, in addition to intersection, since all of these operations preserve both regularity and string length. An additional path-language argument shows that they are also closed under relative difference. Let T_1 and T_2 be ϵ-free acceptors for R_1 and R_2 and construct an fsm P that accepts the regular pair-symbol language $Paths(T_1) - Paths(T_2)$. A string-pair belongs to the regular relation $Rel(P)$ if and only if it has an accepting path in T_1 but not in T_2. Thus $Rel(P)$ is $R_1 - R_2$. Being a subset of R_1, it is also same-length.

3.4 Summary of Mathematical Tools

Let us summarize the results to this point. If L_1, L_2, and L are regular languages and R_1, R_2, and R are regular relations, then we know that the following relations are regular:

$$R_1 \cup R_2 \qquad R_1 \cdot R_2 \qquad R^* \qquad R^{-1} \qquad R_1 \circ R_2 \qquad Id(L) \qquad L_1 \times L_2 \qquad Rev(R)$$

We know also that the following languages are regular (x is a string):

$$Dom(R) \qquad Range(R) \qquad L/R \qquad R/L \qquad x/R \qquad R/x$$

Furthermore, if R_1, R_2, and R are in the same-length subclass, then the following also belong to that restricted subclass:

$$R_1 \cup R_2 \qquad R_1 \cdot R_2 \qquad R^* \qquad R^{-1} \qquad R_1 \circ R_2 \qquad Rev(R) \qquad R_1 \cap R_2 \qquad R_1 - R_2$$

$Id(L)$ is also same-length for all L. Intersections and relative differences of arbitrary regular relations are not necessarily regular, however. We emphasize that all these set-theoretic, algebraic operations are also constructive and computational in nature: fsms or fsts that accept the languages and relations that these operations specify can be constructed directly from machines that accept their operands.

Our rule translation procedures makes use of regular relations and languages created with five special operators. The first operator produces a relation that freely introduces symbols from a designated set S. This relation, $Intro(S)$, is defined by the expression $[Id(\Sigma) \cup [\{\epsilon\} \times S]]^*$. If the characters a and b are in Σ and S is {\$}, for example, then $Intro(S)$ contains an infinite set of string pairs including $\langle a, a\rangle$, $\langle a, \$a\rangle$, $\langle a, a\$\$\rangle$, $\langle ab, \$\$a\$b\$\$\rangle$, and so on. Note that $Intro(S)^{-1}$ removes all elements of S from a string if S is disjoint from Σ.

The second is the *Ignore* operator. Given a regular language L and a set of symbols S, it produces a regular language notated as L_S and read as "L ignoring S." The strings of L_S differ from those of L in that occurrences of symbols in S may be freely interspersed. This language is defined by the expression $L_S = Range(Id(L) \circ Intro(S))$. It includes only strings that would be in L if some occurrences of symbols in S were ignored.

The third and fourth operators enable us to express if-then and if-and-only-if conditions on regular languages. These are the operators *If-P-then-S* ("if prefix then suffix") and *If-S-then-P* ("if suffix then prefix"). Suppose L_1 and L_2 are regular languages and consider the set of strings

$$If\text{-}P\text{-}then\text{-}S(L_1, L_2) = \{x \mid \text{for every partition } x_1x_2 \text{ of } x, \text{ if } x_1 \in L_1, \text{ then } x_2 \in L_2\}$$

A string is in this set if each of its prefixes in L_1 is followed by a suffix in L_2. This set is also a regular language: it *excludes* exactly those strings that have a prefix in L_1 followed by a suffix *not* in L_2 and can therefore be defined by

$$If\text{-}P\text{-}then\text{-}S(L_1, L_2) = \overline{L_1\overline{L_2}}$$

This operator, the regular-language analog of the logical equivalence between $P \rightarrow Q$ and $\neg(P \wedge \neg Q)$, involves only concatenation and complementation, operations under which regular languages (though not relations) are closed. We can also express the symmetric requirement that a prefix be in L_1 if its suffix is in L_2 by the expression

$$If\text{-}S\text{-}then\text{-}P(L_1, L_2) = \overline{\overline{L_1}L_2}$$

Finally, we can combine these two expressions to impose the requirement that a prefix be in L_1 if and only if its suffix is in L_2:

$$P\text{-}iff\text{-}S(L_1, L_2) = If\text{-}P\text{-}then\text{-}S(L_1, L_2) \cap If\text{-}S\text{-}then\text{-}P(L_1, L_2)$$

These five special operators, being constructive combinations of more primitive ones, can also serve as components of practical computation.

The double complementation in the definitions of these conditional operators, and also in several other expressions to be introduced later, constitutes an idiom for expressing universal quantification. While a regular expression $\alpha\beta\gamma$ expresses the proposition that an instance of β occurs between *some* instance of α and *some* instance of γ, the expression $\overline{\alpha\overline{\beta}\gamma}$ claims that an instance of β intervenes between *every* instance of α and a following instance of γ.

4. Rewriting Rule Formalisms

Phonological rewriting rules have four parts. Their general form is

$$\phi \rightarrow \psi / \lambda \_\_ \rho$$

This says that the string ϕ is to be replaced by (rewritten as) the string ψ whenever it is preceded by λ and followed by ρ. If either λ or ρ is empty, it is omitted and, if both are empty, the rule is reduced to

$$\phi \rightarrow \psi$$

The *contexts*, or *environments*, λ and ρ are usually allowed to be regular expressions over a basic alphabet of *segments*. This makes it easy to write, say, a vowel-harmony rule that replaces a vowel that is not specified for backness as a back or front vowel according as the vowel in the immediately preceding syllable is back or front. This is because the Kleene closure operator can be used to state that any number of consonants can separate the two vowels. The rule might be formulated as follows:

$$V_i \rightarrow B_i / B_j C * \_\_$$

where B_i is the back counterpart of the vowel V_i, and B_j is another (possibly different) back vowel. There is less agreement on the restrictions that should apply to ϕ and ψ, the portions that we refer to as the *center* of the rule. They are usually simple strings and some theorists would restrict them to single segments. However, these restrictions are without interesting mathematical consequences and we shall be open to all versions of the theory if we continue to take it that these can also denote arbitrary regular languages.

It will be important to provide for multiple applications of a given rule, and indeed, this will turn out to be the major source of difficulty in reexpressing rewriting rules in terms of regular relations and finite-state transducers. We have already remarked that our methods work only if the part of the string that is actually rewritten by a rule is excluded from further rewriting by that same rule. The following optional rule shows that this restriction is necessary to guarantee regularity:

$$\epsilon \rightarrow \mathrm{ab} / \mathrm{a} \_\_ \mathrm{b}$$

If this rule is allowed to rewrite material that it introduced on a previous application, it would map the regular language $\{ab\}$ into the context-free language $\{a^n b^n \mid 1 \leq n\}$, which we have already seen is beyond the power of regular relations.

However, we do not forbid material produced in one application of a rule from serving as *context* for a subsequent application of that rule, as would routinely be the case for a vowel-harmony rule, for example. It is this restriction on interactions between different applications of a given rule that motivates the notation

$$\phi \rightarrow \psi / \lambda \_\_ \rho$$

rather than

$$\lambda \phi \rho \rightarrow \lambda \psi \rho$$

The context refers to a part of the string that the current application of the rule does not change but which, since it may have been changed in a previous application, allows for an interaction between successive applications.

The important outstanding question concerning interactions between applications of one and the same rule at different positions in a string has to do with the relative order in which they take place. Consider the obligatory rule

$$a \rightarrow b / ab \_\_ ba$$

as applied to the string

abababababa

At least three different outcomes are possible, namely:

(1) abbbabbbaba

(2) ababbbabbba

(3) abbbbbbbbba

Result (1) is obtained if the first application is at the leftmost eligible position in the string; each successive application applies to the output of any preceding one, and further to the right in the string. We call this the *left-to-right* strategy. The corresponding *right-to-left* strategy gives rise to (2). Result (3) comes from identifying all possible rule applications in the original string and carrying them out *simultaneously*. All three strategies have been advocated by phonologists. We shall assume that each rule is marked individually to show which strategy is to be employed for it. We shall concentrate on these three strategies, but other less obvious ones can also be treated by simple rearrangements of our techniques.

Optional rules and most obligatory rules will produce at least one output string, perhaps just a copy of the input if the conditions for application are nowhere satisfied. But certain obligatory rules are anomalous in that they may produce no output at all. The following left-to-right rule is a case in point:

$$\epsilon \rightarrow b / b \_\_$$

If a string containing the symbol *b* is input to this rule, another *b* will be inserted immediately after it, and that one will serve to trigger the rule again. This process will never terminate, and no finite-length output is ever produced. Strange as they may seem, rules like this are useful as filters to eliminate undesired paths of derivation.

In contrast to obligatory rules, optional rules typically produce many outputs. For example, if the rule above ($a \rightarrow b/ab \_\_ ba$) is marked as optional and left-to-right and is also applied to the string *abababababa*, the following, in addition to (1), would be among its outputs:

(4) abbbabababa

(5) ababbbababa

The string (4) is similar to (1) except that only the leftmost application of the rule has been carried out. For (5) the application in the middle would not have been possible for the obligatory rule and is possible here only because the necessary context was not destroyed by an application further to the left.

Kenstowicz and Kisseberth (1979), who discuss a number of rule application strategies in great detail, cite a case in which one rule seems to be required in the grammars

of two languages. However, it must be applied left to right in one, but right to left in the other. In the Australian language Gidabal, the long vowel of certain suffixes becomes short if the vowel of the preceding syllable is long. We find, for example, *yagā+ya* 'should fix' where we would otherwise expect *yagā+yā*. (We use + to mark the point at which the suffix begins and a bar over a vowel to show that it is long.) The interesting question concerns what happens when several of these suffixes are added to the same stem. Some examples are:

Underlying	Surface
barbar+yā+dāng	*barbar+ā+dang* 'straight above'
djalum+bā+dāng+bē	*djalum+bā+dang+bē* 'is certainly right on the fish'
gunūm+bā+dāang+bē	*gunūm+ba+dāng+be* 'is certainly right on the stump'

The rule that Kenstowicz and Kisseberth propose is essentially the following:

$$\bar{V} \rightarrow V/\bar{V}\ C^*\ \_\_$$

This produces the desired result only if applied left to right and only if obligatory. The alternation of long and short vowels results from the fact that each application shortens a long vowel that would otherwise serve as part of the context for a subsequent application.

The same rule appears as the *rhythmic law* in Slovak—all suffix vowels are shortened following a long vowel, as in the following examples:

vol+ā+me 'we call'	*chīt+a+me* 'we read'
vol+āv+a+me 'we call often'	*chīt+av+a+me* 'we read often'

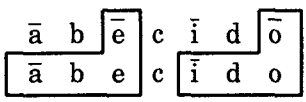
This time the rule must be applied either simultaneously or from right to left.

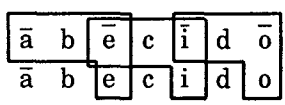
It might seem that a transducer mimicking the operation of a right-to-left rule would have to examine its tapes in the opposite order from one that implemented a left-to-right rule, and it is difficult to see how two transducers operating in different directions could then be composed. However, we shall see that directionality in rewriting rules is not mirrored by directionality in the transducers. Instead, directionality determines which of the two tapes the left and right contexts must appear on. In a left-to-right rule, the left context of the rule is to be verified against the portion of the string that results from previous applications of that rule, whereas the right context is to be verified against the portion of the string that has not yet been changed but may eventually be modified by applications further to the right. In a right-to-left rule, the situation is reversed.

Consider again the left-to-right rule schema

$$\bar{V} \rightarrow V/\bar{V}\ C^*\ \_\_$$

which applies to the string *ābēcīdō* to give *ābecīdo*. The portions of the tapes that support the two applications of the rule are boxed in the diagram on the left below. The diagram on the right shows how it comes about that there are three applications when the rule is taken as moving from right to left.





It is often convenient in phonological rules to introduce a special symbol to mark the beginning and end of the string. This allows edge-conditioned string transformations to be encoded in rewriting rules. For example, Kenstowicz and Kisseberth give the following rule to describe the devoicing of final obstruents in German and Russian:

$$[+\text{obstruent}] \rightarrow [-\text{voiced}] \; / \; \_\_ \; \#$$

We will consider the feature notation exemplified here shortly. For the moment, it can be taken as equivalent to a set of rules whose effect is to replace any segment that is classified as an obstruent by its unvoiced equivalent before the boundary symbol that marks the end of a word. It accounts for the phonological realization of the Russian form *xleb* 'bread' as *xlep*. The boundary symbol # is special in the rule formalism in that it can only appear in the context parts of a rule, never in the input or output patterns, and it never matches an element that appears explicitly in the string. Although boundary-context rules require distinctive mathematical treatment, we show below that they also denote only regular string relations.

As we have said, we take it that each rule in a grammar will be annotated to show which strategy is to be used in applying it. We also assume that rules are annotated to show whether they are to be taken as obligatory or optional. We have considered only obligatory rules up to now, but optional rules are also commonly used to account for cases of free variation. The mathematical treatment of optional rules will turn out to be a simpler case of what must be done for obligatory rules and, therefore, a natural step in the general development.

As well as providing for various strategies for reapplying a single rule, we also consider the possibility of what we call a *batch* rule. This is a set of rules that the application strategies treat as one entity, the individual rules being otherwise unordered relative to one another. This mode of rule application will turn out to be interesting even if it is not an explicit part of any particular phonological formalism because, as we shall see, it constitutes an essential step in the interpretation of rules that use features to refer to underspecified segments. A good example of this is the vowel-harmony rule referred to earlier, namely

$$V_i \rightarrow B_i \; / \; B_j C^*$$

In feature notation, this could be written

$$\begin{bmatrix} +\text{syllabic} \\ -\text{consonantal} \end{bmatrix} \rightarrow [\ +\text{back}\ ] \; / \; \begin{bmatrix} +\text{back} \\ +\text{syllabic} \\ -\text{consonantal} \end{bmatrix} [\ +\text{consonantal}\ ]^* \; \_\_$$

meaning that a segment that is specified as a vowel comes also to be specified as back when the most recent preceding vowel is back; all other features remain unchanged. The grammar will presumably contain another rule that will apply in circumstances when this one does not, namely

$$V_i \rightarrow F_i \; / \; F_j C^*$$

or

$$\begin{bmatrix} +\text{syllabic} \\ -\text{consonantal} \end{bmatrix} \rightarrow [\ -\text{back}\ ] \; / \; \begin{bmatrix} -\text{back} \\ +\text{syllabic} \\ -\text{consonantal} \end{bmatrix} [\ +\text{consonantal}\ ]^* \; \_\_$$

except, of course, that the context can be omitted from whichever of the two is placed second in an ordered list of rules. But this is precisely the question: What is the proper order of this pair of rules?

Consider an actual case, namely, vowel harmony in Turkish. Let *A* represent an abstract vowel with *e* and *a* as its front and back realizations, and *I* another abstract vowel with *i* and dotless ı as its front and back counterparts. The first of these occurs, for example, in the abstract plural suffix *lAr*, and the second occurs in the possessive suffix *Im*, meaning 'my.' Both suffixes can be used together, and the harmony is illustrated by the different realizations of the abstract vowels in the forms *apartmanlArIm* and *adreslArIm*. These appear as *apartmanlarım* 'my apartments' and *adreslerim* 'my addresses.' Using only the simple non-feature notation we started out with, we can describe this variation with the following four rules:

$$\begin{array}{lcl} \text{A} & \rightarrow & \text{e / e C}^* \_\_ \\ \text{A} & \rightarrow & \text{a} \\ \text{I} & \rightarrow & \text{i / e C}^* \_\_ \\ \text{I} & \rightarrow & \text{ı} \end{array}$$

The proper surface forms for these words are produced if these rules are ordered as we have given them—inserting front vowels first—and if each of them is applied from left to right. However, applying the rules in this way gives the wrong result when we create the dative possessive form of *adres* instead of the possessive plural. The dative suffix is spelled simply as the abstract vowel *A*, and the abstract *adresIm*A should be realized as *adresime* if harmony is respected. But the rules as given will map *adresImA* to *adresima* instead. This is because the earlier rules apply to the final *A* at a time before the context required for that vowel has been established. Reordering the rules to fix this problem will cause the previous correct analyses to fail. The proper results in all cases come only if we describe Turkish vowel harmony with rules that proceed left to right through the string as a group, applying at each position whichever one matches. This is the mode of application for a set of rules collected together as a batch.

The notion of a batch rule apparently has not arisen as a distinctive formal concept in phonological theories. The reason is doubtless that batch rules are unnecessarily prolix and, in particular, they fail to capture generalizations that can almost always be made about the individual rules that make up a batch. Phonologists prefer rules that are based on feature matrices. These rules allow segments to be referred to by specifying which members of a finite set of properties they do or do not have. A feature matrix can specify a segment completely, in which case it is equivalent to the unanalyzable segment names we have been using, or it can leave it underspecified. Feature matrices therefore constitute an abbreviatory convention with the advantage that what is easy to abbreviate will be motivated to just the extent that the features themselves are motivated. An underspecified segment corresponds to a set of fully specified segments, and a rule that contains underspecified segments corresponds to a set of rules that are to be applied in batch mode.

Feature matrices based on a well-motivated set of features allow the phonologist to capture significant generalizations and thus effectively to reduce the components of our batch rules to a single rule in most cases. A significant addition that has been made to the basic machinery of feature-based rules consists of variables written with lowercase Greek letters α, β, γ, etc. and ranging over the values + and −. We can use them, for example, to collapse our vowel-harmony rules into a single one as follows:

$$\begin{bmatrix} +\text{syllabic} \\ -\text{consonantal} \end{bmatrix} \rightarrow [\ \alpha\text{back}\ ] \,/\, \begin{bmatrix} \alpha\text{back} \\ +\text{syllabic} \\ -\text{consonantal} \end{bmatrix} [\ +\text{consonantal}\ ]^{*} \ \_\_$$

Both occurrences of the variable α must be instantiated to the same value, either + or −, at each application of the rule. What the rule now says is that a vowel takes its backness from the vowel in the preceding syllable; that is, the most recent preceding vowel that is separated from it by zero or more consonants.

While the explicit use of variables is an important addition to the notation, it was in fact foreshadowed by a property of the initial feature system, namely that features not explicitly mentioned in the center of a rule were assumed to be carried over from the input to the output. Without this convention, an explicit variable would have been required for each of these. Explicit feature variables do indeed increase the abbreviatory power of the notation, but, as we show below, they can be translated systematically into batch rules over unanalyzable segments.

We pay special attention to batch rules, feature matrices, and feature variables because they require some nonobvious extensions to the treatment we provide for ordinary rules with unanalyzable symbols. On the other hand, we have nothing to say about the many other notational devices that phonologists have proposed for collapsing rules. These abbreviatory conventions are either already subsumed by the general regular languages we allow as rule components or can be translated in obvious ways to simply ordered rules or batch rules.

5. Rewriting Rules as Regular Relations

We now come to the central problem of proving that an arbitrary rule in our formalism denotes a regular string relation and is thus accepted by an equivalent finite-state transducer. A rule has the general form

$$\phi \rightarrow \psi / \lambda \ \_\_ \ \rho$$

where ϕ, ψ, λ, and ρ are arbitrary regular expressions. The mode of application of the rule is governed by additional parametric specifications, including for example whether the rule applies from left to right or right to left, and whether it is obligatory or optional.

The replacement that such a rule performs is modeled by a relation *Replace* initially defined as follows:

$$Replace = [Id(\Sigma^{*})\ Opt(\phi \times \psi)]^{*}$$

The final asterisk allows for repetitions of the basic $\phi \times \psi$ mapping, and $Id(\Sigma^{*})$ allows identical corresponding substrings to come between successive applications of the rule. The $\phi \times \psi$ replacement is optional to allow for the possibilities that the rule itself may be optional or that there may be no eligible instances of ϕ in the input string. *Replace* is the set of pairs of strings that are identical except for possible replacements of substrings belonging to ϕ by substrings belonging to ψ. This set clearly contains all the pairs that satisfy the rule, though perhaps other pairs as well. The problem now is to impose restrictions on this mapping so that it occurs in the proper contexts and in accordance with the parameters specified for the rule. We do this in a series of approximations.

5.1 Context Requirements

As a first step, we might be tempted simply to add the context restrictions as necessary conditions of the $\phi \times \psi$ replacement:

$$Replace = [Id(\Sigma^*)\ Opt(Id(\lambda)\ \phi \times \psi\ Id(\rho))]^*$$

This relation includes strings where the $\phi \times \psi$ replacement occurs only when immediately preceded and followed by identical substrings satisfying λ and ρ, respectively. But this formulation does not allow for the fact, noted above, that the context strings of one application may overlap either the contexts or the center strings of another. For example, consider the following optional rule, which allows an abstract *B* to be rewritten as *b* intervocalically:

$$B \rightarrow b\ /\ V\ \_\_\ V$$

With the definition of *Replace* just given, the string pair on the left below would be accepted but the pair on the right would not:

V	B	V	B	V		V	B	V	B	V
V	b	V	B	V		V	b	V	b	V

But the second pair also represents a valid application of the rule, one in which the center vowel is serving as the right context of one application and the left context of the other.

The problem is that a given string symbol can simultaneously serve several different roles in the application of a rule, and all possible interactions must be accounted for. As a next approximation, we avoid this confusion by carefully distinguishing and keeping track of these various roles. We first consider how to apply a rule to strings that have been preprocessed so that every instance of the left context λ is followed by the auxiliary symbol $<$ and every instance of the right context ρ is preceded by the symbol $>$, where $<$ and $>$ are not in Σ. This means that the replacement operator can be defined solely in terms of these distinct context-marking brackets, without regard to what λ and ρ actually specify and what they might have in common with each other or with ϕ and ψ. In essence, we assume that the replacement relation for the above rule applies to the upper strings shown below, and that all three string pairs are acceptable because each of the corresponding *B*-*b* pairs is bracketed by $<$ and $>$.

>V<B>V<B>V<	>V<B>V<B>V<	>V<B>V<B>V<
>V<b>V<b>V<	>V<b>V<B>V<	>V<B>V<b>V<

To take a somewhat more realistic example, when the rule at the beginning of the paper

$$N \rightarrow m\ /\ \_\_\ [+labial]$$

is applied to the string *iNprobable*, the preprocessed input string would contain the sequence

<i<N<>p<r<o<>b<a<>b<l<e<

The left context of the rule is empty, so there is a left-context marker $<$ after every character from the original string. Every labial is an instance of the right context, and accordingly there is a $>$ immediately preceding *p*'s and *b*'s. The rule properly applies to rewrite the *N* because it is bracketed by $<$ and $>$. On the other hand, the $>$ is missing

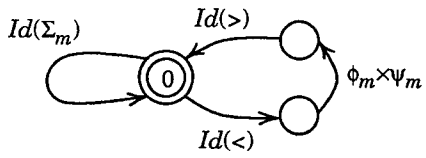


Figure 9
The *Replace* transducer.

and the rule does not apply to the *N* in the preprocessed version of *iNtractable*, namely

<i<N<t<r<a<c<t<a<>b<l<e<

The definition of the *Replace* operator must be modified in two ways in order to operate on such preprocessed strings. First it must allow the $\phi \times \psi$ mapping only between the appropriate context markers. Second, some occurrences of the left and right context strings do not result in rule applications, either because the rule is optional or because the other conditions of the rule are not satisfied. Thus, the relation must disregard the markers corresponding to those occurrences inside the identity substrings between rule applications. Relations with this behavior can be obtained through the use of the ignoring operator defined in Section 3, which is notated by subscripting. Let m (for marker) be $\{<, >\}$, the set of both markers. Then our next approximation to the replacement relation is defined as follows:

$$Replace = [Id(\Sigma^*_m)\ Opt(Id(<)\ \phi_m \times \psi_m\ Id(>))]^*$$

This allows arbitrary strings of matching symbols drawn from $\Sigma \cup \{<, >\}$ between rule applications and requires $<:<$ and $>:>$ to key off a ϕ-ψ replacement. The subscript m's also indicate that $<$ and $>$ can be ignored in the middle of the replacement, since the appearance of left- or right-context strings is irrelevant in the middle of a given rule application. Figure 9 shows the general form of the state-transition diagram for a transducer that accepts a replacement relation. As before, the start-state is labeled 0 and only transitions are shown from which the final-state is reachable.

We must now define relations that guarantee that context-markers do in fact appear on the strings that *Replace* applies to, and only when sanctioned by instances of λ and ρ. We do this in two stages. First, we use simple relations to construct a *Prologue* operator that freely introduces the context markers in m:

$$Prologue = Intro(m)$$

An output string of *Prologue* is just like the corresponding input except that brackets appear in arbitrary positions. The relation $Prologue^{-1}$ removes all brackets that appear on its input.

Second, we define more complex identity relations that pair a string with itself if and only if those markers appear in the appropriate contexts. The *P-iff-S* operator is the key component of these context-identifying predicates. The condition we must impose for the left context is that the left-context bracket $<$ appears if and only if it is immediately preceded by an instance of λ. This basic requirement is satisfied by strings in the regular language *P-iff-S*$(\Sigma^*\lambda, <\Sigma^*)$. The situation is slightly more complicated, however, because of two special circumstances.

An instance of λ may have prefixes that are also λ instances. If λ is the expression ab^*, then $a<b<$ is an acceptable marking but $ab<$ and $a<b$ are not because the two

λ-instances are not both followed by $<$. The brackets that necessarily follow such prefixes must not prevent the longer instances from also being identified and marked, and right-context brackets also must not interfere with left-context identification. The ignore operators in the expression $P\text{-}iff\text{-}S(\Sigma^*{}_<\lambda_<, <\Sigma^*{}_<)_>$ allow for these possibilities. This disregards slightly too many brackets, however: since an instance of $\lambda_<$ followed by an $<$ is also an instance of $\lambda_<$, it must be followed by another bracket, and so on. The only (finite) strings that belong to this language are those that contain no instances of λ at all! To correctly identify and mark left-contexts, the bracket following a $\lambda_<$ instance must not be ignored. Thus, the requisite set of strings is the regular language $Leftcontext(\lambda, <, >)$, where the *Leftcontext* operator is defined as follows:

$$Leftcontext(\lambda, l, r) = P\text{-}iff\text{-}S(\Sigma_l^*\lambda_l - \Sigma_l^* l, l\Sigma_l^*)_r$$

We parameterize this operator for the left-context pattern and the actual brackets so that it can be used in other definitions below.

The other complication arises in rules intended to insert or delete material in the string, so that either ϕ or ψ includes the empty string ϵ. Consider the left-to-right rule

$$a \rightarrow \epsilon \ / \ b$$

Iterated applications of this rule can delete an arbitrary sequence of a's, converting strings of the form $baaaa\ldots a$ into simply b. The single b at the beginning serves as left-context for applications of the rule to each of the subsequent a's. This presents a problem for the constructions we have developed so far: The *Replace* relation requires a distinct $<$ marker for each application of the rule. The $<$ that sanctions the deletion of the leftmost a in the string is therefore not available to delete the next one. However, the *Leftcontext* operator as defined disallows two left-context brackets in a row. Our solution is to insert an explicit character 0 to represent the deleted material. If *Leftcontext* ignores this character in λ, 0 will always be followed by another left bracket and thus another rule application is possible.

The auxiliary symbol 0 is not in Σ or in the set of context brackets. It will substitute for the empty strings that might appear in the center of rules (in ϕ or ψ), but it is a genuine symbol in an expanded alphabet which, unlike the normal ϵ, actually appears as a distinct element in character strings. The *Prologue* relation is extended to freely introduce 0 as well as the brackets in m:

$$Prologue = Intro(m \cup \{0\})$$

We then construct alternative versions of ϕ and ψ in which this special symbol replaces the true empty strings. We define

$$\phi^0 = \begin{cases} \phi & \text{if } \epsilon \notin \phi \\ [\phi - \epsilon] \cup 0 & \text{otherwise} \end{cases}$$

which contains exactly the same strings as ϕ except that the singleton string 0 is included instead of the empty string that otherwise might be in the language. ψ^0 is defined similarly, and then the replacement operator is expressed in terms of these new regular languages:

$$Replace = [Id(\Sigma^*{}_{m\,0})\ Opt(Id(<)\ \phi_m^0 \times \psi_m^0\ Id(>))]^*$$

Now we can complete our definition of the left-context identifier:

$$Leftcontext(\lambda, l, r) = P\text{-}iff\text{-}S(\Sigma^*{}_{l\,0}\ \lambda_{l\,0} - \Sigma^*{}_{l\,0}\ l,\ l\ \Sigma^*{}_{l\,0})_r$$

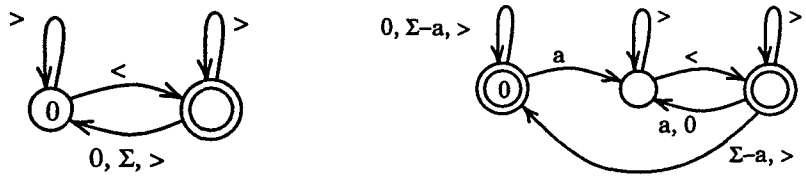


Figure 10
Left-context identifiers.

As desired, the regular language denoted by this operator includes strings if and only if every substring belonging to λ (ignoring l, r, and 0) is immediately followed by a bracket l. This effect is illustrated by the state-transition diagrams in Figure 10. The machine on the left is a minimal-state acceptor for the empty-context language $Leftcontext(\epsilon, <, >)$. It accepts strings that have at least one $<$, and every 0 or Σ symbol must be followed by a $<$. The $>$-labeled transitions represent the fact that $>$ is being ignored. The machine on the right accepts the language $Leftcontext(a, <, >)$; it requires $<$ to appear after every a or after any 0 that follows an a. This particular machine is nondeterministic so that its organization is easier to understand.

An operator for identifying and marking right-context strings can be defined symmetrically:

$$Rightcontext(\rho, l, r) = P\text{-}iff\text{-}S(\Sigma^*_{r\,0}\ r,\ \rho_{r\,0}\ \Sigma^*_{r\,0} - r\ \Sigma^*_{r\,0})_l$$

Thus $Rightcontext(\rho, <, >)$ includes strings if and only if every substring belonging to ρ (with appropriate ignoring) is immediately *preceded* by a right-context bracket $>$. Alternatively, taking advantage of the fact that the reversal of a regular language is also a regular language, we can define *Rightcontext* in terms of *Leftcontext*:

$$Rightcontext(\rho, l, r) = Rev(Leftcontext(Rev(\rho), r, l))$$

These context identifiers denote appropriate string-sets even for rules with unspecified contexts, if the vacuous contexts are interpreted as if the empty string had been specified. The empty string indicates that adjacent symbols have no influence on the rule application. If an omitted λ is interpreted as ϵ, for example, every *Leftcontext* string will have one and only one left-context bracket at its beginning, its end, and between any two Σ symbols, thus permitting a rule application at every position.

5.2 Directional and Simultaneous Application

We now have components for freely introducing and removing context brackets, for rejecting strings with mislocated brackets, and for representing the rewrite action of a rule between appropriate context markers. The regular relation that models the optional application of a rule is formed by composition of these pieces. The order of composition depends on whether the rule is specified as applying iteratively from left to right or from right to left.

As noted in Section 4, the difference is that for left-to-right rules, the left-context expression λ can match against the output of a previous (that is, leftward) application of the same rule, but the right-context expression ρ must match against the as yet unchanged input string. These observations are directly modeled by the order in which the various rule components are combined. For a left-to-right rule, the right context is checked on the input (ϕ) side of the replacement, while the left context is checked on the output (ψ) side. The regular relation and corresponding transducer for a left-

to-right optional rule is therefore defined by the following sequence of compositions:

$$\begin{gathered} \mathit{Prologue} \circ \\ \mathit{Id}(\mathit{Rightcontext}(\rho, <, >)) \circ \\ \mathit{Replace} \circ \\ \mathit{Id}(\mathit{Leftcontext}(\lambda, <, >)) \circ \\ \mathit{Prologue}^{-1} \end{gathered}$$

Both left- and right-context brackets are freely introduced on input strings, strings in which the right-context bracket is mislocated are rejected, and the replacement takes place only between the now-constrained right-context brackets and the still free left-context markers. This imposes the restriction on left-context markers that they at least appear before replacements, although they may or may not freely appear elsewhere. The left-context checker ensures that left-context markers do in fact appear only in the proper locations on the output. Finally, all brackets are eliminated, yielding strings in the output language.

The context-checking situation is exactly reversed for right-to-left rules: the left-context matches against the unchanged input string while the right-context matches against the output. Right-to-left optional application can therefore be modeled simply by interchanging the context-checking relations in the cascade above, to yield

$$\begin{gathered} \mathit{Prologue} \circ \\ \mathit{Id}(\mathit{Leftcontext}(\lambda, <, >)) \circ \\ \mathit{Replace} \circ \\ \mathit{Id}(\mathit{Rightcontext}(\rho, <, >)) \circ \\ \mathit{Prologue}^{-1} \end{gathered}$$

The transducer corresponding to this regular relation, somewhat paradoxically, models a right-to-left rule application while moving from left to right across its tapes.

Simultaneous optional rule application, in which the sites of all potential string modifications are located before any rewriting takes place, is modeled by a cascade that identifies both left and right contexts on the input side of the replacement:

$$\begin{gathered} \mathit{Prologue} \circ \\ \mathit{Id}(\mathit{Leftcontext}(\lambda, <, >) \cap \mathit{Rightcontext}(\rho, <, >)) \circ \\ \mathit{Replace} \circ \\ \mathit{Prologue}^{-1} \end{gathered}$$

5.3 Obligatory Application

These compositions model the optional application of a rule. Although all potential application sites are located and marked by the context checkers, these compositions do not force a ϕ-ψ replacement to take place for every instance of ϕ appearing in the proper contexts. To model obligatory rules, we require an additional constraint that rejects string pairs containing sites where the conditions of application are met but the replacement is not carried out. That is, we must restrict the relation so that, disregarding for the moment the effect of overlapping applications, every substring of the form $\lambda\phi\rho$ in the first element of a pair corresponds to a $\lambda\psi\rho$ in the second element of that pair. We can refine this restriction by framing it in terms of our context-marking brackets: the *Replace* relation must not contain a pair with the substring $<\phi>$ in one element corresponding to something distinct from $<\psi>$ in the other.

We might try to formulate this requirement by taking the complement of a relation that includes the undesired correspondences, as suggested by the expression

$$\overline{\mathit{Id}(\Sigma^*_{m\,0})\ \mathit{Id}(<)\ \phi^0_m \times \overline{\psi^0_m}\ \mathit{Id}(>)\ \mathit{Id}(\Sigma^*_{m\,0})}$$

This expression might be taken as the starting point for various augmentations that would correctly account for overlapping applications. However, pursuing this line of attack will not permit us to establish the fact that obligatory rules also define regular mappings. First, it involves the complement of a regular relation, and we observed above that the complement of a regular relation (as opposed to the complement of a regular language) is not necessarily regular. Second, even if the resulting relation itself turned out to be regular, the obvious way of entering it into our rule composition is to intersect it with the replacement relation, and we also know that intersection of relations leads to possibly nonregular results.

Proving that obligatory rules do indeed define regular mappings requires an even more careful analysis of the roles that context-brackets can play on the various intermediate strings involved in the rule composition. A given left-context bracket can serve in the *Replace* relation in one of three ways. First, it can be the start of a rule *application*, provided it appears in front of an appropriate configuration of ϕ, ψ, and right-context brackets. Second, it can be ignored during the *identity* portions of the strings, the regions between the changes sanctioned by the replacement relation. Third, it can be ignored because it comes in the middle or *center* of another rule application that started to the left of the bracket in question and extends further to the right. Suppose we encode these three different roles in three distinct left-bracket symbols $\underset{a}{<}$, $\underset{i}{<}$, and $\underset{c}{<}$ and also provide for a similar set of distinct right-context brackets $\underset{a}{>}$, $\underset{i}{>}$, and $\underset{c}{>}$. Wherever a rule is properly applied, the input side of the replacement relation will contain a substring of the form

$$\underset{a}{<} \phi^0 \underset{c}{<>} \underset{a}{>}$$

The crucial difference in the case where an obligatory left-to-right rule incorrectly fails to apply is that the left-context preceding the ϕ^0 is marked with $\underset{i}{<}$ instead of $\underset{a}{<}$, since it is part of an identity sequence. This situation is undesirable no matter what types of brackets are ignored in the ϕ^0 pattern or mark the right-context of this potential application. Whether those brackets are in the center or at the boundary of replacements that are carried out further to the right of the offending situation, the leftward application marked by the $\underset{i}{<}$ should have taken precedence.

The symbols $<$ and $>$ were previously used as auxiliary characters appearing in intermediate strings. With a slight abuse of notation, we now let them act as cover symbols standing for the sets of left and right brackets $\{\underset{i}{<}, \underset{a}{<}, \underset{c}{<}\}$ and $\{\underset{i}{>}, \underset{a}{>}, \underset{c}{>}\}$ respectively, and we let m be the combined set $< \cup >$. A substring on the input side of the replacement is then a missed left-to-right application if it matches the simple pattern $\underset{i}{<}\phi^0_m>$. Thus, we can force obligatory application of a left-to-right rule by requiring that the strings on the input side of its replacement contain no such substrings, or, to put it in formal terms, that the input strings belong to the regular language *Obligatory*$(\phi, \underset{i}{<}, >)$, where *Obligatory* is defined by the following operator:

$$\textit{Obligatory}(\phi, l, r) = \overline{\Sigma^*_{m\,0}\ l\ \phi^0_m\ r\ \Sigma^*_{m\,0}}$$

By symmetry, a missed application of a right-to-left rule matches the pattern $<\phi^0_m\underset{i}{>}$, and *Obligatory*$(\phi, <, \underset{i}{>})$ is the appropriate input filter to disallow all such substrings. Note that the obligatory operator involves only regular languages and not relations so that the result is still regular despite the complementation operation.

We must now arrange for the different types of brackets to appear on the input to *Replace* only in the appropriate circumstances. As before, the context identifiers must

ensure that none of the brackets can appear unless preceded (or followed) by the appropriate context, and that every occurrence of a context is marked by a bracket freely chosen from the appropriate set of three. The *Leftcontext* and *Rightcontext* operators given above will have exactly this effect when they are applied with the new meanings given to $<$, $>$, and m. The *Replace* operator must again be modified, however, because it alone distinguishes the different roles of the context brackets. The following final definition chooses the correct brackets for all parameters of rule application:

$$Replace = [Id(\Sigma^*_{\substack{<\\i}\ \substack{>\\i}}\ 0)\ Opt(Id(\underset{a}{<})\ \phi^0_{\substack{<\\c}\ \substack{>\\c}} \times \psi^0_{\substack{<\\c}\ \substack{>\\c}}\ Id(\underset{a}{>}))]^*$$

The behavior of obligatory rules is modeled by inserting the appropriate filter in the sequence of compositions. Left-to-right obligatory rules are modeled by the cascade

$$\begin{gathered} Prologue \circ \\ Id(Obligatory(\phi, \underset{i}{<}, >)) \circ \\ Id(Rightcontext(\rho, <, >)) \circ \\ Replace \circ \\ Id(Leftcontext(\lambda, <, >)) \circ \\ Prologue^{-1} \end{gathered}$$

and right-to-left obligatory rules are modeled by:

$$\begin{gathered} Prologue \circ \\ Id(Obligatory(\phi, <, \underset{i}{>})) \circ \\ Id(Leftcontext(\lambda, <, >)) \circ \\ Replace \circ \\ Id(Rightcontext(\rho, <, >)) \circ \\ Prologue^{-1} \end{gathered}$$

We remark that even obligatory rules do not necessarily provide a singleton output string. If the language ψ contains more than one string, then outputs will be produced for each of these at each application site. Moreover, if ϕ contains strings that are suffixes or prefixes (depending on the direction of application) of other strings in ϕ, then alternatives will be produced for each length of match. A particular formalism may specify how such ambiguities are to be resolved, and these stipulations would be modeled by additional restrictions in our formulation. For example, the requirement that only shortest ϕ matches are rewritten could be imposed by ignoring only one of $\underset{c}{<}$ or $\underset{c}{>}$ in the mapping part of *Replace*, depending on the direction of application.

There are different formulations for the obligatory application of simultaneous rules, also depending on how competition between overlapping application sites is to be resolved. Intersecting the two obligatory filters, as in the following cascade, models the case where the longest substring matching ϕ is preferred over shorter overlapping matches:

$$\begin{gathered} Prologue \circ \\ Id(Obligatory(\phi, \underset{i}{<}, >) \cap Obligatory(\phi, <, \underset{i}{>})) \circ \\ Id(Rightcontext(\rho, <, >) \cap Leftcontext(\lambda, <, >)) \circ \\ Replace \circ \\ Prologue^{-1} \end{gathered}$$

The operators can be redefined and combined in different ways to model other regimes for overlap resolution.

5.4 Boundary Contexts

A rule contains the special boundary marker # when the rewriting it describes is conditioned by the beginning or end of the string. The boundary marker only makes sense when it appears in the context parts of the rule; specifically, when it occurs at the left end of a left-context string or the right end of a right-context string. No special treatment for the boundary marker would be required if # appeared as the first and last character of every input and output string and nowhere else. If this were the case, the compositional cascades above would model exactly the intended interpretation wherein the application of the rule is edge-sensitive. Ordinary input and output strings do not have this characteristic, but a simple modification of the *Prologue* relation can simulate this situation. We defined *Prologue* above as $Intro(m \cup \{0\})$. We now augment that definition:

$$Prologue = Intro(m \cup \{0\}) \circ [\epsilon{:}\# \; Id(\overline{\Sigma^*_{m\,0} \; \# \; \Sigma^*_{m\,0}}) \; \epsilon{:}\#]$$

We have composed an additional relation that introduces the boundary marker at the beginning and end of the already freely bracketed string, and also rejects strings containing the boundary marker somewhere in the middle. The net effect is that strings in the cascade below the *Prologue* are boundary-marked; bracketed images of the original input strings and the context identifiers can thus properly detect the edges of those strings. The inverse *Prologue* at the bottom of the cascade removes the boundary marker along with the other auxiliary symbols.

5.5 Batch Rules

It remains to model the application of a set of rules collected together in a single batch. Recall that for each position in the input string each rule in a batch set is considered for application independently. As we have seen several times before, there is a straightforward approach that approximates this behavior. Let $\{R_1, \ldots, R_n\}$ be the set of regular relations for rules that are to be applied as a batch and construct the relation $[\cup_k R_k]^*$. Because of closure under union, this relation is regular and includes all pairs of strings that are identical except for substrings that differ according to the rewriting specified by at least one of the rules. But also as we have seen several times before, this relation does not completely simulate the batch application of the rules. In particular, it does not allow for overlap between the material that satisfies the application requirements of one rule in the set with the elements that sanction a previous application of another rule. As usual, we account for this new array of overlapping dependencies by introducing a larger set of special marking symbols and carefully managing their occurrences and interactions.

A batch rule is a set of subrules $\{\phi^1 \rightarrow \psi^1/\lambda^1 \_\_ \rho^1, \ldots, \phi^n \rightarrow \psi^n/\lambda^n \_\_ \rho^n\}$ together with a specification of the standard parameters of application (left-to-right, obligatory, etc.). We use superscripts to distinguish the components of the different subrules to avoid (as much as possible) confusion with our other notational conventions. A crucial part of our treatment of an ordinary rule is to introduce special bracket symbols to mark the appearance of its left and right contexts so that its replacements are carried out only in the proper (possibly overlapping) environments. We do the same thing for each of the subrules of a batch, but we use a different set of brackets for each of them. These brackets permit us to code in a single string the context occurrences for all the different subrules with each subrule's contexts distinctively marked.

Let $<^k$ be the set $\{\underset{i}{<^k}, \underset{a}{<^k}, \underset{c}{<^k}\}$ of left-context brackets for the k^{th} subrule $\phi^k \rightarrow \psi^k/\lambda^k \_\_ \rho^k$ of the batch, let $>^k$ be the corresponding set of right-context brackets,

and let m^k be the set $<^k \cup >^k$. We also redefine the generic cover symbols $<$, $>$, and m to stand for the respective collections of all brackets: $< = \cup_k <^k$, $> = \cup_k >^k$, $m = < \cup >$. Note that with this redefinition of m, the *Prologue* relation as defined above will now freely introduce all the brackets for all of the subrules. It will also be helpful to notate the set of brackets *not* containing those for the k^{th} subrule: $m^{-k} = m - m^k$.

Now consider the regular language $Leftcontext(\lambda^k, <^k, >^k)_{m^{-k}}$. This contains strings in which all instances of the k^{th} subrule's left-context expression are followed by one of the k^{th} left-context brackets, and those brackets appear only after instances of λ^k. The k^{th} right-context brackets are freely distributed, as are all brackets for all the other subrules. Occurrences of all other left-context brackets are restricted in similarly defined regular languages. Putting all these bracket-restrictions together, the language

$$\bigcap_k Leftcontext(\lambda^k, <^k, >^k)_{m^{-k}}$$

has each subrule's left-context duly marked by one of that subrule's left-context brackets. This leaves all right-context brackets unconstrained; they are restricted to their proper positions by the corresponding right-context language

$$\bigcap_k Rightcontext(\rho^k, <^k, >^k)_{m^{-k}}$$

These intersection languages, which are both regular, will take the place of the simple context identifiers when we form the composition cascades to model batch-rule application. These generalized context identifiers are also appropriate for ordinary rules if we regard each of them as a batch containing only one subrule.

A replacement operator for batch rules must also be constructed. This must map between input and output strings with context-brackets properly located, ensuring that any of the subrule rewrites are possible at each properly marked position but that the rewrite of the k^{th} subrule occurs only between $\underset{a}{<^k}$ and $\underset{a}{>^k}$. The complete set of possible rewrites is encoded in the relation

$$\bigcup_k [Id(\underset{a}{<^k})\ \phi^{0k}_{\underset{c}{<}\ \underset{c}{>}} \times \psi^{0k}_{\underset{c}{<}\ \underset{c}{>}}\ Id(\underset{a}{>^k})]$$

where the generic symbol $\underset{c}{<}$ now stands for $\{\underset{c}{<^1} \ldots \underset{c}{<^k}\}$, the set of all left-center brackets, and the generic $\underset{c}{>}$ is assigned a corresponding meaning. We incorporate this relation as the rewrite part of a new definition of the *Replace* operator, with the generic $\underset{i}{<}$ and $\underset{i}{>}$ now representing the sets of all left and right identity brackets:

$$Replace = [Id(\Sigma^*_{\underset{i}{<}\ \underset{i}{>} 0})\ Opt(\bigcup_k [Id(\underset{a}{<^k})\ \phi^{0\ k}_{\underset{c}{<}\ \underset{c}{>}} \times \psi^{0\ k}_{\underset{c}{<}\ \underset{c}{>}}\ Id(\underset{a}{>^k})])]^*$$

This relation allows for any of the appropriate replacements separated by identity substrings. It is regular because of the union-closure property; this would not be the case, of course, if intersection or complementation had been required for its construction.

A model of the left-to-right application optional application of a batch rule is obtained by substituting the new, more complex definitions in the composition cascade for ordinary rules with these application parameters:

$$\begin{gathered} Prologue \circ \\ Id(\bigcap_k Rightcontext(\rho^k, <^k, >^k)_{m^{-k}}) \circ \\ Replace \circ \\ Id(\bigcap_k Leftcontext(\lambda^k, <^k, >^k)_{m^{-k}}) \circ \\ Prologue^{-1} \end{gathered}$$

Optional right-to-left and simultaneous batch rules are modeled by similar substitutions in the corresponding ordinary-rule cascades. Obligatory applications are handled by combining instances of the *Obligatory* operator constructed independently for each subrule. $Obligatory(\phi^k, \underset{i}{<^k}, >^k)$ excludes all strings in which the k^{th} subrule failed to apply, moving from left to right, when its conditions of application were satisfied. The intersection of the obligatory filters for all subrules in the batch ensures that at least one subrule is applied at each position where application is allowed. Thus the behavior of a left-to-right obligatory batch rule is represented by the composition

$$\begin{gathered} Prologue \circ \\ Id(\underset{k}{\cap} Obligatory(\phi^k, \underset{i}{<^k}, >^k)) \circ \\ Id(\underset{k}{\cap} Rightcontext(\rho^k, <^k, >^k)_{m-k}) \circ \\ Replace \circ \\ Id(\underset{k}{\cap} Leftcontext(\lambda^k, <^k, >^k)_{m-k}) \circ \\ Prologue^{-1} \end{gathered}$$

Again, similar substitutions in the cascades for ordinary obligatory rules will model the behavior of right-to-left and simultaneous application.

5.6 Feature Matrices and Finite Feature Variables

Using only operations that preserve the regularity of string sets and relations, we have modeled the properties of rewriting rules whose components are regular languages over an alphabet of unanalyzable symbols. We have thus established that every such rule denotes a regular relation. We now extend our analysis to rules involving regular expressions with feature matrices and finite feature variables, as in the Turkish vowel harmony rule discussed in Section 4:

$$\begin{bmatrix} +\text{syllabic} \\ -\text{consonantal} \end{bmatrix} \rightarrow [\ \alpha\text{back}\ ] \,/\, \begin{bmatrix} \alpha\text{back} \\ +\text{syllabic} \\ -\text{consonantal} \end{bmatrix} [\ +\text{consonantal}\ ]^* \ \_\_$$

We first translate this compact feature notation, well suited for expressing linguistic generalizations, into an equivalent but verbose notation that is mathematically more tractable. The first step is to represent explicitly the convention that features not mentioned in the input or output matrices are left unchanged in the segment that the rule applies to. We expand the input and output matrices with as many variables and features as necessary so that the value of every output feature is completely specified in the rule. The center-expanded version of this example is

$$\begin{bmatrix} +\text{syllabic} \\ -\text{consonantal} \\ \beta\text{back} \\ \beta_1\text{round} \\ \beta_2\text{high} \\ \dots \\ \beta_n\text{f}_n \end{bmatrix} \rightarrow \begin{bmatrix} +\text{syllabic} \\ -\text{consonantal} \\ \alpha\text{back} \\ \beta_1\text{round} \\ \beta_2\text{high} \\ \dots \\ \beta_n\text{f}_n \end{bmatrix} / \begin{bmatrix} \alpha\text{back} \\ +\text{syllabic} \\ -\text{consonantal} \end{bmatrix} [\ +\text{consonantal}\ ]^* \ \_\_$$

The input and output feature matrices are now fully specified, and in the contexts the value of any unmentioned feature can be freely chosen.

A feature matrix in a regular expression is quite simple to interpret when it does not contain any feature variables. Such a matrix merely abbreviates the union of all

segment symbols that share the specified features, and the matrix can be replaced by that set of unanalyzable symbols without changing the meaning of the rule. Thus, the matrix [+consonantal] can be translated to the regular language {p, t, k, b, d...} and treated with standard techniques. Of course, if the features are incompatible, the feature matrix will be replaced by the empty set of segments.

A simple translation is also available for feature variables all of whose occurrences are located in just one part of the rule, as in the following fictitious left context:

$$[\ \alpha\text{high}\ ]\ [\ +\text{consonantal}\ ]^*\ [\ -\alpha\text{round}\ ]$$

If α takes on the value +, then the first matrix is instantiated to [+high] and denotes the set of unanalyzable symbols, say {e, i, ...}, that satisfy that description. The last matrix reduces to [−round] and denotes another set of unanalyzable symbols (e.g. {a, e, i, ...}). The whole expression is then equivalent to

$$\{\text{e, i, ...}\}\ \{\text{p, t, k, b, d ...}\}^*\ \{\text{a, e, i, ...}\}$$

On the other hand, if α takes on the value −, then the first matrix is instantiated to [−high] and denotes a different set of symbols, say {a, o...}, and the last one reduces to [+ round]. The whole expression on this instantiation of α is equivalent to

$$\{\text{a , o, ...}\}\ \{\text{p, t, k, b, d...}\}^*\ \{\text{o, u, ...}\}$$

On the conventional interpretation, the original expression matches strings that belong to either of these instantiated regular languages. In effect, the variable is used to encode a correlation between choices from different sets of unanalyzable symbols.

We can formalize this interpretation in the following way. Suppose θ is a regular expression over feature matrices containing a single variable α for a feature whose values are drawn from a finite set V, commonly the set $\{+, -\}$. Let $\theta[\alpha \rightarrow v]$ be the result of substituting $v \in V$ for α wherever it occurs in θ, and then replacing each variable-free feature matrix in that result by the set of unanalyzable symbols that satisfy its feature description. Then the interpretation of θ is given by the formula

$$\bigcup_{v \in V} \theta[\alpha \rightarrow v]$$

This translation produces a regular expression that properly models the choice-correlation defined by α in the original expression. Rule expressions containing several locally occurring variables can be handled by an obvious generalization of this substitution scheme. If $\alpha_1 \ldots \alpha_n$ are the local variables in θ whose values come from the finite sets $V_1 \ldots V_n$, the set of n-tuples

$$I = \{<\alpha_1 \rightarrow v_1, \ldots, \alpha_n \rightarrow v_n> \mid v_1 \in V_1 \ldots v_n \in V_n\}$$

represents the collection of all possible value instantiations of those variables. If we let $\theta[i]$ be the result of carrying out the substitutions indicated for all variables by some i in I, the interpretation of the entire expression is given by the formula

$$\bigcup_{i \in I} \theta[i]$$

When all local variables are translated, the resulting expression may still contain feature matrices with nonlocal variables, those that also occur in other parts of the rule.

Indeed, the input and output expressions will almost always have variables in common, because of the feature variables introduced in the initial center-expansion step.

Variables that appear in more than one rule part clearly cannot be eliminated from each part independently, because the correlation between feature instantiations would be lost. A feature-matrix rule is to be interpreted as scanning in the appropriate direction along the input string until a configuration of symbols is encountered that satisfies the application conditions of the rule instantiated to one selection of values for all of its variables. The segments matching the input are then replaced by the output segments determined by that same selection, and scanning resumes until another configuration is located that matches under possibly a different selection of variables values. This behavior is modeled as the batch-mode application of a set of rules each of which corresponds to one variable instantiation of the original rule.

Consider a center-expanded rule of the general form $\phi \rightarrow \psi / \lambda \_\_ \rho$, and let I be the set of possible value instantiations for the feature-variables it contains. Then the collection of instantiated rules is simply

$$\{\phi[i] \rightarrow \psi[i] / \lambda[i] \_\_ \rho[i] \mid i \in I\}$$

The components of the rules in this set are regular languages over unanalyzable segment symbols, all feature matrices and variables having been resolved. Since each instantiated rule is formed by applying the same substitution to each of the original rule components, the cross-component correlation of symbol choices is properly represented. The behavior of the original rule is thus modeled by the relation that corresponds to the batch application of rules in this set, and we have already shown that such a relation is regular.

5.7 Summary

This completes our examination of individual context-sensitive rewriting rules. We have modeled the input-output behavior of these rules according to a variety of different application parameters. We have expressed the conditions and actions specified by a rule in terms of carefully constructed formal languages and string relations. Our constructions make judicious use of distinguished auxiliary symbols so that crucial informational dependencies can be string-encoded in unambiguous ways. We have also shown how these languages and relations can be combined by set-theoretic operations to produce a single string relation that simulates the rule's overall effect. Since our constructions and operations are all regularity-preserving, we have established the following theorem:

Theorem

For all the application parameters we have considered, every rewriting rule whose components describe regular languages denotes a regular string relation.

This theorem has an immediate corollary:

Corollary

The input-output string pairs of every such rewriting rule are accepted by some finite-state transducer.

This theoretical result has important practical consequences. The mathematical analysis that establishes the theorem and its corollary is constructive in nature. Not only do we know that an appropriate relation and its corresponding transducer exist, we also

know all the operations to perform to construct such a transducer from a particular rule. Thus, given a careful implementation of the calculus of regular languages and regular relations, our analysis provides a general method for compiling complicated rule conditions and actions into very simple computational devices.

6. Grammars of Rewriting Rules

The individual rules of a grammar are meant to capture independent phonological generalizations. The grammar formalism also specifies how the effects of the different rules are to be combined together to account for any interactions between the generalizations. The simplest method of combination for rewriting rule grammars is for the rules to be arranged in an ordered sequence with the interpretation that the first rule applies to the input lexical string, the second rule applies to the output of the first rule, and so on. As we observed earlier, the typical practice is to place specialized rules with more elaborate context requirements earlier in the sequence so that they will override more general rules appearing later.

The combined effect of having one rule operate on the output of another can be modeled by composing the string relations corresponding to each rule. If the string relations for two rules are regular, we know that their composition is also regular. The following result is then established by induction on the number of rules in the grammar:

Theorem
If $G = \langle R_1, \ldots, R_n \rangle$ is a grammar defined as a finite ordered sequence of rewriting rules each of which denotes a regular relation, then the set of input-output string-pairs for the grammar as a whole is the regular relation given by $R_1 \circ \ldots \circ R_n$.

This theorem also has an immediate corollary:

Corollary
The input-output string pairs of every such rewriting grammar are accepted by a single finite-state transducer.

Again, given an implementation of the regular calculus, a grammar transducer can be constructed algorithmically from its rules.

We can also show that certain more complex methods of combination also denote regular relations. Suppose a grammar is specified as a finite sequence of rules but with a further specification that rules in some subsequences are to be treated as a block of mutually exclusive alternatives. That is, only one rule in each such subsequence can be applied in any derivation, but the choice of which one varies freely between derivations. The alternative choices among the rules in a block can be modeled as the union of the regular relations they denote individually, and regular relations are closed under this operation. Thus this kind of grammar also reduces to a finite composition of regular relations.

In a more intricate arrangement, the grammar might specify a block of alternatives made up of rules that are not adjacent in the ordering sequence. For example, suppose the grammar consists of the sequence $\langle R_1, R_2, R_3, R_4, R_5 \rangle$, where R_2 and R_4 constitute a block of exclusive alternatives. This cannot be handled by simple union of the block rules, because that would not incorporate the effect of the intervening rule R_3. However, this grammar can be interpreted as abbreviating a choice between two

different sequences, $\langle R_1, R_2, R_3, R_5 \rangle$ and $\langle R_1, R_3, R_4, R_5 \rangle$, and thus denotes the regular relation

$$R_1 \circ [(R_2 \circ R_3) \cup (R_3 \circ R_4)] \circ R_5$$

The union and composition operators can be interleaved in different ways to show that a wide variety of rule combination regimes are encompassed by the regular relations. There may be grammars specifying even more complex rule interactions, and, depending on the formal details, it may be possible to establish their regularity by other techniques; for example, by carefully managing a set of distinguished auxiliary symbols that code inter-rule constraints.

We know, of course, that certain methods for combining regular rules give rise to nonregular mappings. This is true, for example, of unrestricted cyclic application of the rules in a finite ordered sequence. According to a cyclic grammar specification, a given input string is mapped through all the rules in the sequence to produce an output string, and that output string then becomes a new input for a reapplication of all the rules, and the process can be repeated without bound. We can demonstrate that such a grammar is nonregular by considering again the simple optional rule

$$\epsilon \rightarrow \mathrm{ab} / \mathrm{a} \_\_ \mathrm{b}$$

We showed before that this rule does not denote a regular relation if it is allowed to rewrite material that was introduced on a previous application. Under those circumstances it would map the regular language $\{ab\}$ into the context-free language $\{a^n b^n \mid 1 \leq n\}$. But we would get exactly the same result from an unrestricted cyclic grammar whose ordered sequence consists only of this single rule. In effect, cyclic reapplication of the rule also permits it to operate arbitrarily, often on its own output. In the worst case, in fact, we know that the computations of an arbitrary Turing machine can be simulated by a rewriting grammar with unrestricted rule reapplication.

These results seem to create a dilemma for our regularity analysis. Many phonological formalisms based on ordered sets of rewriting rules provide for cyclic rule applications. The underlying notion is that words have a bracketed structure reflecting their morphological composition. For example, *unenforceable* has the structure [*un*[[*en*[*force*]]*able*]]. The idea of the cycle is that the ordered sequence of rules is applied to the innermost bracketed portion of a word first. Then the innermost set of brackets is removed and the procedure is repeated. The cycle continues in this way until no brackets remain.

The cycle has been a major source of controversy ever since it was first proposed by Chomsky and Halle (1968), and many of the phenomena that motivated it can also be given noncyclic descriptions. Even for cases where a nonrecursive, iterative account has not yet emerged, there may be restrictions on the mode of reapplication that limit the formal power of the grammar without reducing its empirical or explanatory coverage. For example, the bracket erasure convention means that new string material becomes accessible to the rules on each cycle. If, either implicitly or explicitly, there is also a finite bound on the amount of old material to which rules in the new cycle can be sensitive, it may be possible to transform the recursive specification to an equivalent iterative one. This is analogous to the contrast between center-embedding context-free grammars and grammars with only right- or left-linear rules; the latter are known to generate only regular languages. Unfortunately, phonological theories are usually not presented in enough formal detail for us to carry out such a mathematical analysis. The regularity of cyclic phonological formalisms will have to be examined on a case-by-case basis, taking their more precise specifications into account.

We have shown that every noncyclical rewriting grammar does denote a regular relation. We now consider the opposite question: Is every regular relation denoted by some noncyclic rewriting grammar? We can answer this question in the affirmative:

Theorem
Every regular relation is the set of input/output strings of some noncyclic rewriting grammar with boundary-context rules.

Proof
Let R be an arbitrary regular relation and let $T = (\Sigma, Q, q_0, F, \delta)$ be a finite-state transducer that accepts it. Without loss of generality we assume that Σ and Q are disjoint. We construct a rewriting grammar that simulates the operation of T, deriving a string y from a string x if and only if the pair $\langle x, y \rangle$ is accepted by T. There will be four rules in the grammar that together implement the provisions that T starts in state q_0, makes transitions from state to state only as allowed by δ, and accepts a string only if the state it reaches at the end of the string is in F. Let $\Sigma \cup Q \cup \{\#, \$\}$ be the alphabet of the grammar, where # is a boundary symbol not in either Σ or Q and \$ is another distinct symbol that will be used in representing the finality of a state. Our rules will introduce states into the string between ordinary tape symbols and remove them to simulate the state-to-state advance of the transducer. The first rule in the grammar sequence is the simple start rule:

$$\epsilon \rightarrow q_0 / \# \_\_\_ \quad \text{(obligatory, left-to-right)}$$

The effect of this rule is to introduce the start-state as a symbol only at the beginning of the input string, as specified in the rule by the boundary symbol #. The string *abc* is thus rewritten by this rule to q_0abc. The following sets of rules are defined to represent the state-to-state transitions and the final states of the transducer:

$$\textit{Transitions} = \{u \rightarrow vq_j / q_i \_\_\_ \mid q_i, q_j \in Q; u, v \in \Sigma^\epsilon; \text{ and } q_j \in \delta(q_i, u, v)\}$$
$$\textit{Final} = \{\epsilon \rightarrow \$ / q_i \_\_\_ \# \mid q_i \in F\}$$

The second rule of the grammar is an obligatory, left-to-right batch rule consisting of all the rules in *Transitions* $\cup$ *Final*. If the transition function carries the transducer from q_i to q_j over the pair $\langle u, v \rangle$, there will be a rule in *Transitions* that applies to the string $\ldots q_i u \ldots$ at the position just after the substring beginning with q_i and produces $\ldots q_i v q_j \ldots$ as its output. Because δ is a total function on $Q \times \Sigma^\epsilon \times \Sigma^\epsilon$, some subrule will apply at every string position in the left-to-right batch scan. The state-context for the left-most application of this rule is the start-state q_0, and subrules corresponding to start-state transitions are selected. This introduces a state-symbol that makes available at the next position only subrules corresponding to transitions at one of the start-state's successors. The batch rule eventually writes a state at the very end of the string. If that state is in F, the corresponding *Final* subrule will apply to insert \$ at the end of the string. If the last state is not in F, \$ will not be inserted and the state will remain as the last symbol in the string. Thus, after the batch rule has completed its application, an input string x will have been translated to an output string consisting of intermixed symbols from Q and Σ. We can prove by a simple induction that the string of states obtained by ignoring symbols in $\Sigma \cup \{\#, \$\}$ corresponds to a sequence of state-to-state moves that the transducer can make on the pair $\langle x, y \rangle$, where y comes from ignoring \$ and all state-symbols in the output string.

Two tasks remain: we must filter the output to eliminate any strings whose derivation does not include a *Final* subrule application, and we must remove all state-symbols and \$ to obtain the ultimate output string. If a *Final* rule did not apply, then the last element in the batch output string is a state, not the special character \$. We must formulate a rule that will "bleed" the derivation, producing no output at all if its input ends in a state-symbol instead of \$. We can achieve this with an anomalous obligatory rule whose output would be infinitely long if its input ever satisfies the conditions for its application. The following rule behaves in this way:

$$\epsilon \rightarrow \$/Q \_\_ \# \text{ (obligatory, left-to-right)}$$

It has the effect of filtering strings that do not represent transitions to a final state by forcing indefinitely many insertions of \$ when no single \$ is present. The output of this rule will be all and only the strings that came from a previous application of a *Final* rule. The last rule of the grammar is a trivial clean-up rule that produces the grammar's final output strings:

$$Q \cup \$ \rightarrow \epsilon \text{ (obligatory, left-to-right)}$$

This completes the proof of the theorem. We have constructed for any regular relation an ordered sequence of four rules (including a batch rule with finitely many subrules) that rewrites a string x to y just in case the pair $\langle x, y \rangle$ belongs to the relation. □

We remark that there are alternative but perhaps less intuitive proofs of this theorem framed only in terms of simple nonbatch rules. But this result cannot be established without making use of boundary-context rules. Without such rules we can only simulate a proper subclass of the regular relations, those that permit identity prefixes and suffixes of unbounded length to surround any nonidentity correspondences. It is interesting to note that for much the same reason, Ritchie (1992) also made crucial use of two-level boundary-context rules to prove that the relations denoted by Koskenniemi's (1985) two-level grammars are also coextensive with the regular relations. Moreover, putting Ritchie's result together with ours gives the following:

Theorem
Ordered rewriting grammars with boundaries and two-level constraint grammars with boundaries are equivalent in their expressive power.

Although there may be aesthetic or explanatory differences between the two formal systems, empirical coverage by itself cannot be used to choose between them.

7. Two-Level Rule Systems

Inspired in part by our early report of the material presented in this paper (Kaplan and Kay 1981), Koskenniemi (1983) proposed an alternative system for recognizing and producing morphological and phonological word-form variants. Under his proposal, individual generalizations are expressed directly in the state-transition diagrams of finite-state transducers, and their mutual interactions emerge from the fact that every input-output string pair must be accepted simultaneously by all these transducers. Thus, he replaced the serial feeding arrangement of the independent generalizations in a rewriting grammar with a parallel method of combination. In eliminating the intermediate strings that pass from one rewriting rule to another, he also reduced to

just two the number of linguistically meaningful levels of representation. In two-level parlance, these are usually referred to as the lexical and surface strings.

7.1 The Analysis of Parallel Automata

The lexical-surface string sets of the individual generalizations in Koskenniemi's system are clearly regular, since they are defined outright as finite-state transducers. But it is not immediately obvious that the string relation defined by a whole two-level grammar is regular. Koskenniemi gave an operational specification, not an algebraic one, of how the separate transducers are to interact. A pair of strings is generated by a two-level grammar if the pair is accepted separately by each of the transducers, and furthermore, the label on the transition taken by one fst at a particular string position is identical to the label of the transition that every other fst takes at that string position. In essence, he prescribed a transition function δ for a whole-grammar transducer that allows transitions between states in cross-product state sets just in case they are permitted by literal-matching transitions in the individual machines.

This transition function generalizes to a two-tape transducer the construction of a one-tape finite-state machine for the intersection of two regular languages. We might therefore suspect that the lexical-surface relation for a two-level grammar consisting of transducers $T_1, \ldots T_n$ is the relation $\cap_i R(T_i)$. However, what is actually computed under this interpretation is the relation $Rel(Paths(T_1) \cap Paths(T_2) \ldots Paths(T_n))$ of the form discussed in Section 3. As we observed, this may be only a proper subset of the relation $\cap_i R(T_i)$ when the component relations contain string pairs of unequal length. In this case, the literal-matching transducer may not accept the intersection, a relation that in fact may not even be regular.

The individual transducers allowed in two-level specifications do permit the expansion and contraction of strings by virtue of a null symbol 0. If this were treated just like ϵ, we would be able to say very little about the combined relation. However, the effect of Koskenniemi's literal-matching transition function is achieved by treating 0 as an ordinary tape symbol, so that the individual transducers are ϵ-free. The intersection of their same-length relations is therefore regular. The length-changing effect of the whole-grammar transducer is then provided by mapping 0 onto ϵ. Thus we embed the same-length intersection $\cap_i R(T_i)$ as a regular inner component of a larger regular relation that characterizes the complete lexical-to-surface mapping:

$$Intro(0) \circ [\underset{i}{\cap} R(T_i)] \circ Intro(0)^{-1}$$

This relation expands its lexical string by freely introducing 0 symbols. These are constrained along with all other symbols by the inner intersection, and then the surface side of the inner relation is contracted by the removal of all 0's. The entire outer relation gives an algebraic model of Koskenniemi's operational method for combining individual transducers and for interpreting the null symbol. With this analysis of the two-level system in terms of regularly-closed operations and same-length relations, we have shown that the string relations accepted by parallel two-level automata are in fact regular. We have also shown, by the way, that the two-level system is technically a four-level one, since the inner relation defines two intermediate, 0-containing levels of representation. Still, only the two outer levels are linguistically significant. In typical two-level implementations the *Intro* relations are implicitly encoded in the interpretation algorithms and do not appear as separate transducers.

7.2 Two-Level Rule Notation

Koskenniemi (1983) offered an informal grammatical notation to help explicate the intended effect of the individual transducers and make the generalizations encoded in their state-transition diagrams easier to understand and reason about. However, he proposed no method of interpreting or compiling that notation. In later work (e.g. Karttunen, Koskennienni, and Kaplan 1987; Karttunen and Beesley 1992) the mathematical techniques we presented above for the analysis of rewriting systems were used to translate this notation into the equivalent regular relations and corresponding transducers, and thus to create a compiler for a more intuitive and more tractable two-level rule notation. Ritchie (1992) summarizes aspects of this analysis as presented by Kaplan (1988). Ritchie et al. (1992) describe a program that interprets this notation by introducing and manipulating labels assigned to the states of component finite-state machines. Since these labels have no simple set-theoretic significance, such an approach does not illuminate the formal properties of the system and does not make it easy to combine two-level systems with other formal devices.

Ignoring some notational details, a grammar of two-level rules (as opposed to fsts) includes a specification of a set of "feasible pairs" of symbols that we denote by π. The pairs in π contain all the alphabet symbols and 0, but do not contain ϵ except possibly when it is paired with itself in $\epsilon{:}\epsilon$. The relations corresponding to all the individual rules are all subsets of π^*, and thus are all of the restricted same-length class (since π does not contain ϵ paired with an alphabetic symbol). For this class of relations, it makes sense to talk about a correspondence between a symbol in one string in a string-pair and a symbol in the other: in the pair $\langle abc, lmn \rangle$ for example, we can say that a corresponds to l, b to m, and c to n, by virtue of the positions they occupy relative to the start of their respective same-length strings. Symbol pairs that correspond in this way must be members of π. It also makes sense to talk about corresponding substrings, sequences of string pairs whose symbols correspond to each other in some larger string pair. Corresponding substrings belong to π^*.

The grammar contains a set of rules whose parts are also same-length regular-relation subsets of π^*. There are two basic kinds of rules, context restriction rules and surface coercion rules. A simple context restriction rule is an expression of the form

$$\tau \Rightarrow \lambda \_\_ \rho$$

where τ, λ, and ρ denote subsets of π^*. Usually τ is just a single feasible pair, a singleton element of π, but this limitation has no mathematical significance. Either of the contexts λ or ρ can be omitted, in which case it is taken to be $\epsilon{:}\epsilon$. Such a rule is interpreted as denoting a string relation whose members satisfy the following conditions: Every corresponding substring of a string pair that belongs to the relation τ must be immediately preceded by a corresponding substring belonging to the left-context λ and followed by one belonging to the right-context ρ. In other words, any appearance of τ outside the specified contexts is illegal. Under this interpretation, the relation denoted by the rule

$$a{:}b \Rightarrow c{:}d \_\_ e{:}f$$

would include the string pairs $\langle cae, dbf \rangle$ and $\langle cae, cge \rangle$, assuming that $a{:}g$ is in π along with the symbol pairs mentioned in the rule. The first string pair is included because the pair $a{:}b$ is properly surrounded by $c{:}d$ and $e{:}f$. The second belongs because it contains an instance of $a{:}g$ instead of $a{:}b$ and thus imposes no requirements on the surrounding context. The string pair $\langle cae, cbe \rangle$ is not included, however, because $a{:}b$ appears in a context not sanctioned by the rule.

A simple surface coercion rule is written with the arrow going in the other direction:

$$\tau \Leftarrow \lambda \_\_\_ \rho$$

For strings to satisfy a constraint of this form, they must meet a more complicated set of conditions. Suppose that a corresponding substring belonging to λ comes before a corresponding substring belonging to ρ, and that the lexical side of the paired substring that comes between them belongs to the domain of τ. Then that intervening paired substring must itself belong to τ. To illustrate, consider the surface coercion version of the context restriction example above:

$$a{:}b \Leftarrow c{:}d \_\_\_ e{:}f$$

The string pair $\langle cae, dbf \rangle$ satisfies this constraint because $a{:}b$ comes between the context substrings $c{:}d$ and $e{:}f$. The pair $\langle cbe, dbf \rangle$ is also acceptable, because the string intervening between $c{:}d$ and $e{:}f$ does not have a on its lexical side. However, the pair $\langle cae, dgf \rangle$ does not meet the conditions because $a{:}g$ comes between $c{:}d$ and $e{:}f$. The lexical side of this is the same as the lexical side of the τ relation $a{:}b$, but the pair $a{:}g$ itself is not in τ. Informally, this rule forces the a to be realized as a surface b when it appears between the specified contexts.

Karttunen et al. (1987) introduced a variant of a surface coercion rule called a surface prohibition. This is a rule of the form

$$\tau / \Leftarrow \lambda \_\_\_ \rho$$

and indicates that a paired substring that comes between instances of λ and ρ and whose lexical side is in the domain of τ must *not* itself belong to τ. We shall see that the mathematical properties of surface prohibitions follow as immediate corollaries of our surface coercion analysis.

The notation also permits compound rules of each type. These are rules in which multiple context pairs are specified. A compound context restriction rule is of the form

$$\tau \Rightarrow \lambda^1 \_\_\_ \rho^1; \lambda^2 \_\_\_ \rho^2; \ldots \lambda^n \_\_\_ \rho^n$$

and is satisfied if each instance of τ is surrounded by an instance of *some* λ^k-ρ^k pair. A compound surface coercion rule requires the τ surface realization in *each* of the specified contexts.

For convenience, surface coercions and context restrictions can be specified in a single rule, by using $\Leftrightarrow$ as the main connective instead of $\Leftarrow$ or $\Rightarrow$. A bidirectional rule is merely an abbreviation that can be included in a grammar in place of the two subrules formed by replacing the $\Leftrightarrow$ first by $\Leftarrow$ and then $\Rightarrow$. Such rules need no further discussion.

7.3 Context Restriction Rules ($\Rightarrow$)

To model the conditions imposed by context restriction rules, we recall the *If-P-then-S* and *If-S-then-P* operators we defined for regular languages L_1 and L_2:

$$\textit{If-P-then-S}(L_1, L_2) = \overline{L_1 \overline{L_2}} = \Sigma^* - L_1 \overline{L_2}$$

$$\textit{If-S-then-P}(L_1, L_2) = \overline{\overline{L_1} L_2} = \Sigma^* - \overline{L_1} L_2$$

These operators can be extended to apply to string relations as well, and the results will be regular if the operands are in a regular subclass that is closed under complementation. For notational convenience, we let the overbar in these extensions and in the other expressions below stand for the complement relative to π^* as opposed to the more usual $\Sigma^* \times \Sigma^*$:

$$\textit{If-P-then-S}(R_1, R_2) = \pi^* - R_1\overline{R_2} = \overline{R_1\overline{R_2}}$$

$$\textit{If-S-then-P}(R_1, R_2) = \pi^* - \overline{R_1}R_2 = \overline{\overline{R_1}R_2}$$

The conditions for a simple context restriction rule are then modeled by the following relation:

$$\textit{Restrict}(\tau, \lambda, \rho) = \textit{If-S-then-P}(\pi^*\lambda, \tau\pi^*) \cap \textit{If-P-then-S}(\pi^*\tau, \rho\pi^*)$$

The first component ensures that τ is always preceded by λ, and the second guarantees that it is always followed by ρ.

A compound context restriction rule of the form

$$\tau \Rightarrow \lambda^1 \_\_ \rho^1; \lambda^2 \_\_ \rho^2; \ldots; \lambda^n \_\_ \rho^n$$

is satisfied if all instances of τ are surrounded by substrings meeting the conditions of at least one of the context pairs independently. A candidate model for this disjunctive interpretation is the relation

$$\bigcup_k \textit{Restrict}(\tau, \lambda^k, \rho^k)$$

This is incorrect, however, because the scope of the union is too wide. It specifies that there must be some k such that *every* occurrence of τ will be surrounded by λ^k-ρ^k, whereas the desired interpretation is that each τ must be surrounded by λ^k-ρ^k, but a different k might be chosen for each occurrence. A better approximation is the relation

$$[\bigcup_k \textit{Restrict}(\tau, \lambda^k, \rho^k)]^*$$

Because of the outer Kleene * iteration, different instances of the rule can apply at different string-pair positions. But this also has a problem, one that should be familiar from our study of rewriting rules: the iteration causes the different rule instances to match on separate, successive substrings. It does not allow for the possibility that the context substring of one application might overlap with the center and context portions of a preceding one.

This problem can be solved with the auxiliary-symbol techniques we developed for rewriting rule overlaps. We introduce left and right context brackets $<^k$ and $>^k$ for each context pair λ^k-ρ^k. These are distinct from all other symbols, and since their identity pairs are now feasible pairs, they are added to π. These pairs take the place of the actual context relations in the iterative union

$$[\bigcup_k \textit{Restrict}(\tau, \textit{Id}(<^k), \textit{Id}(>^k))]^*$$

This eliminates the overlap problem. We then must ensure that these bracket pairs appear only if appropriately followed or preceded by the proper context relation. With m being the set of all bracket pairs and subscripting now indicating that identity pairs of the specified symbols are ignored, we define a two-level left-context operator

$$\textit{Leftcontext}(\lambda, l) = \textit{If-S-then-P}(\pi^*\lambda_m, \textit{Id}(l)\pi^*)$$

so that $Leftcontext(\lambda^k, <^k)$ enforces the requirement that every $<^k$ pair be preceded by an instance of λ^k. This is simpler than the rewriting left-context operator because not every instance of λ must be marked—only the ones that precede τ, and those are picked out independently by the iterative union. That is why this uses a one-way implication instead of a biconditional. As in the rewriting case, the ignoring provides for overlapping instances of λ. The right-context operator can be defined symmetrically using *If-P-then-S* or by reversing the left-context operator:

$$Rightcontext(\rho, r) = Rev(Leftcontext(Rev(\rho), r))$$

Putting the pieces together, the following relation correctly models the interpretation of a compound context restriction rule:

$$[\bigcup_k Restrict(\tau, Id(<^k), Id(>^k))]^* \cap [\bigcap_k [Leftcontext(\lambda^k, <^k) \cap Rightcontext(\rho^k, >^k)]] \begin{array}{l} Intro(m) \circ \\ \circ\ Intro(m)^{-1} \end{array}$$

Auxiliary marks are freely introduced on the lexical string. Those marks are appropriately constrained so that matching brackets enclose every occurrence of τ, and each bracket marks an occurrence of the associated context relation. The marks are removed at the end. Note that there are only same-length relations in the intermediate expression, and that all brackets introduced at the top are removed at the bottom. Thus the composite relation is regular and also belongs to the same-length subclass, so that the result of intersecting it with the same-length regular relations for other rules will be regular.

7.4 Surface Coercion Rules (⇐)

A surface coercion rule of the form

$$\tau \Leftarrow \lambda \_\_ \rho$$

imposes a requirement on the paired substrings that come between all members of the λ and ρ relations. If the lexical side of such a paired substring belongs to the domain of τ, then the surface side must be such that the intervening pair belongs to τ.

To formalize this interpretation, we first describe the set of string pairs that fail to meet the conditions. The complement of this set is then the appropriate relation. The relation $\overline{\tau} = \pi^* - \tau$ is the set of string pairs in π^* that are not in τ, because either their lexical string is not in the domain of τ or τ associates that lexical string with different surface strings. $Id(Dom(\tau)) \circ \overline{\tau}$ is the subset of these whose lexical strings *are* in the domain of τ and whose surface strings must therefore be different than τ provides for. The unacceptable string pairs thus belong to the same-length relation $\pi^*\lambda[Id(Dom(\tau)) \circ \overline{\tau}]\rho\pi^*$, and its regular complement in the *Coerce* operator

$$Coerce(\tau, \lambda, \rho) = \overline{\pi^*\lambda[Id(Dom(\tau)) \circ \overline{\tau}]\rho\pi^*}$$

contains all the string pairs that satisfy the rule.

For most surface coercions it is also the case that this contains only the pairs that satisfy the rule. But for one special class of coercions, the epenthesis rules, this relation includes more string pairs than we desire. These are rules in which the domain of τ includes strings consisting entirely of 0's, and the difficulty arises because of the dual nature of two-level 0's. They behave formally as actual string symbols in same-length

relations, but they are also intended to act as the empty string. In this way they are similar to the ϵ's in the centers of rewriting rules, and they must also be modeled by special techniques. The epenthesis rule

$$0{:}b \Leftarrow c{:}c \_\_ d{:}d$$

can be used to illustrate the important issues.

If this is the only rule in a grammar, then clearly that grammar should allow the string pair $\langle cd, cbd\rangle$ but disallow the pair $\langle cd, ced\rangle$, in which e appears instead of b between the surface c and d. It should also disallow the pair $\langle cd, cd\rangle$, in which c and d are adjacent on both sides and no epenthesis has occurred. This is consistent with the intuition that the 0 in the rule stands for the absence of explicit lexical string material, and that therefore the rule must force a surface b when lexical c and d are adjacent. In our analysis this interpretation of 0 is expressed by having the *Intro* relation freely introduce 0's between any other symbols, mimicking the fact that ϵ can be regarded as freely appearing everywhere. The pair $\langle cd, cbd\rangle$ is allowed as the composition of pairs $\langle cd, c0d\rangle$ and $\langle c0d, cbd\rangle$; the first pair belongs to the *Intro* relation and the second is sanctioned by the rule. But because 0's are introduced freely, the *Intro* relation includes the identity pair $\langle cd, cd\rangle$ as well. The *Coerce* relation as defined above also contains the pair $\langle cd, cd\rangle$ $(= \langle c\epsilon d, c\epsilon d\rangle)$, since $\epsilon{:}\epsilon$ is not in $[0{:}0 \circ \overline{0{:}b}]$. The grammar as a whole thus allows $\langle cd, cd\rangle$ as an undesired composition.

We can eliminate pairs of this type by formulating a slightly different relation for epenthesis rules such as these. We must still disallow pairs when 0's in the domain of τ are paired with strings not in the range. But we also want to disallow pairs whose lexical strings do not have the appropriate 0's to trigger the grammar's epenthesis coercions. This can be accomplished by a modified version of the *Coerce* relation that also excludes realizations of the empty string by something not in τ. We replace the $Dom(\tau)$ expression in the definition above with the relation $Dom(\tau \cup \{\epsilon{:}\epsilon\})$. The two-level literature is silent about whether or not an epenthesis rule should also reject strings with certain other insertion patterns. On one view, the rule only restricts the insertion of singleton strings and thus pairs such as $\langle cd, cbbd\rangle$ and $\langle cd, ceed\rangle$ would be included in the relation. This view is modeled by using the $Dom(\tau \cup \{\epsilon{:}\epsilon\})$ expression. On another view, the rule requires that lexically adjacent c and d must be separated by exactly one b on the surface, so that $\langle cd, cbbd\rangle$ and $\langle cd, ceed\rangle$ would be excluded in addition to $\langle cd, ced\rangle$ and $\langle cd, cd\rangle$. We can model this second interpretation by using 0^* instead of $Dom(\tau \cup \{\epsilon{:}\epsilon\})$. The relation then restricts the surface realization of any number of introduced 0's. It is not clear which of these interpretations leads to a more convenient formalism, but each of them can be modeled with regular devices.

Karttunen and Beesley (1992, p. 22) discuss a somewhat different peculiarity that shows up in the analysis of epenthesis rules where one context is omitted (or equivalently, one context includes the pair $\epsilon{:}\epsilon$). The rule

$$0 : \mathrm{b} \Leftarrow c{:}c \_\_ d{:}d$$

requires that a b corresponding to nothing in the lexical string must appear in the surface string after every $c{:}c$ pair. If we use either the $Dom(\tau \cup \{\epsilon : \epsilon\})$ or 0^* expressions in defining the coercion relation for this rule, the effect is not what we intend. The resulting relation does not allow strings in which a $\epsilon{:}\epsilon$ follows $c{:}c$, because ϵ is included in the restrictive domain expression. But $\epsilon{:}\epsilon$ follows and precedes every symbol pair, of course, so the result is a relation that simply prohibits all occurrences of $c{:}c$. If, however, we revert to using the domain expression without the $\{\epsilon{:}\epsilon\}$ union, we fall

back into the difficulty we saw with two-context epenthesis rules: the resulting relation properly ensures that nothing other than b can be inserted after $c{:}c$, but it leaves open the possibility of $c{:}c$ followed by no insertion at all.

A third formulation is necessary to model the intended interpretation of one-context epenthesis rules. This is given by the relation $\overline{\pi^*\lambda\overline{\tau}\pi^*}$ if only the left-context is specified, or $\overline{\pi^*\overline{\tau}\rho\pi^*}$ if only ρ appears. These exclude all strings where an instance of the relevant context is followed by paired substrings not in τ, either because the appropriate number of lexical 0's were not (freely) introduced or because those 0's correspond to unacceptable surface material. These two prescriptions can be brought together into the single formula $\overline{\pi^*\lambda\overline{\tau}\rho\pi^*}$ for all one-context rules, since whichever context is missing is treated as the identity pair $\epsilon{:}\epsilon$. We can bring out the similarity between this formula and the original *Coerce* relation by observing that this one is equivalent to $\overline{\pi^*\lambda[Id(Dom(\pi^*)) \circ \overline{\tau}]\rho\pi^*}$ because $Id(Dom(\pi^*)) \circ \overline{\tau}$ and $\overline{\tau}$ are the same relation.

We now give a general statement of the *Coerce* relation that models surface coercions whether they are epenthetic or non-epenthetic:

$$Coerce(\tau, \lambda, \rho) = \overline{\pi^*\lambda[Id(Dom(X)) \circ \overline{\tau}]\rho\pi^*}, \text{ where}$$

$X = \tau$ if τ has no epenthetic pairs;
$X = \tau \cup \{\epsilon{:}\epsilon\}$ (or perhaps $[0{:}0]^*$) if τ has only epenthetic pairs and neither λ nor ρ contains $\epsilon{:}\epsilon$;
$X = \pi^*$ if τ has only epenthetic pairs and one of λ or ρ does contain $\epsilon{:}\epsilon$.

This definition assumes that τ is homogeneous in that either all its string-pairs are epenthetic or none of them are, but we must do further analysis to guarantee that this is the case. In the formalism we are considering, τ is permitted to be an arbitrary same-length relation, not just the single unit-length pair that two-level systems typically provide for. If τ contains more than one string-pair, the single rule is interpreted as imposing the constraints that would be imposed by a conjunction of rules formed by substituting for τ each of its member string-pairs in turn. Without further specification and even if τ contains infinitely many pairs, this is the interpretation modeled by the *Coerce* relation, provided that τ is homogeneous. To deal with heterogeneous τ relations, we separate the epenthetic and nonepenthetic pairs into two distinct and homogeneous subrelations. We partition an arbitrary τ into the subrelations τ^0 and $\tau^{\overline{0}}$ defined as

$$\begin{aligned} \tau^0 &= Id(0^*) \circ \tau \\ \tau^{\overline{0}} &= \tau - \tau^0 \end{aligned}$$

We then recast a rule of the form $\tau \Leftarrow \lambda \_\_ \rho$ as the conjunction of the two rules

$$\begin{aligned} \tau^0 &\Leftarrow \lambda \_\_ \rho \\ \tau^{\overline{0}} &\Leftarrow \lambda \_\_ \rho \end{aligned}$$

These rules taken together represent the desired interpretation of the original, and each of them is properly modeled by exactly one variant of the *Coerce* relation.

We have now dealt with the major complexities that surface coercion rules present. The compound forms of these rules are quite easy to model. A rule of the form

$$\tau \Leftarrow \lambda^1 \_\_ \rho^1; \lambda^2 \_\_ \rho^2; \ldots; \lambda^n \_\_ \rho^n$$

is interpreted as coercing to the surface side of τ if any of the context conditions are met. Auxiliary symbols are not needed to model this interpretation, since there is no iteration to introduce overlap difficulties. The relation for this rule is given simply by the intersection of the individual relations:

$$\bigcap_k Coerce(\tau, \lambda^k, \rho^k)$$

We conclude our discussion of two-level rules with a brief mention of surface prohibitions. Recall that a prohibition rule

$$\tau / \Leftarrow \lambda \_\_ \rho$$

indicates that a paired substring must not belong to τ if it comes between instances of λ and ρ and its lexical side is in the domain of τ. We can construct a standard surface coercion rule that has exactly this interpretation by using the complement of τ restricted to τ's domain:

$$[Id(Dom(\tau)) \circ \overline{\tau}] \Leftarrow \lambda \_\_ \rho$$

As desired, the left side is the relation that maps each string in the domain of τ to all strings other than those to which τ maps it. Surface prohibitions are thus reduced to ordinary surface coercions.

7.5 Grammars of Two-Level Rules

The relation for a grammar of rules is formed just as for a grammar of parallel automata. The intersection of the relations for all the individual rules is constructed as a same-length inner relation. This is then composed with the 0 introduction and removal relations to form the outer lexical-to-surface map. Rule-based two-level grammars thus denote regular relations, just as the original transducer-based grammars do. Some grammars may make use of boundary-context rules, in which case a special symbol # can appear in contexts to mark the beginning and end of the strings. These can be modeled with exactly the same technique we outlined for rewriting rules: we compose the additional relation $[\epsilon{:}\#Id\ (\overline{\Sigma^*_{m0}\ \#\ \Sigma^*_{m0}})\ \epsilon{:}\#]$ at the beginning of the four-level cascade and compose its inverse at the end. As we mentioned before, the two-level grammars with boundary-context rules are the ones that Ritchie (1992) showed were complete for the regular relations.

In reasoning about these systems, it is important to keep clearly in mind the distinction between the outer and inner relations. Ritchie (1992), for example, also proved that the "languages" generated by two-level grammars with regular contexts are closed under intersection, but this result does not hold if a grammar's language is taken to be its outer relation. Suppose that G_1 has the set $\{a{:}b, 0{:}c\}$ as its feasible pairs and the vacuous $a{:}b \Rightarrow \_\_$ as its only rule, and that G_2 has the pairs $\{a{:}c, 0{:}b\}$ and rule $a{:}c \Rightarrow \_\_$. The domain of both outer (0-free) relations is a^*. A string a^n is mapped by G_1 into strings containing n b's with c's freely intermixed and by G_2 into strings containing n c's with b's freely intermixed. The range of the intersection of the outer relations for G_1 and G_2 thus contains strings with the same number of b's and c's but occurring in any order. This set is not regular, since intersecting it with the regular language b^*c^* produces the context-free language b^nc^n. The intersection of the two outer relations is therefore also not regular and so cannot be the outer relation of any regular two-level grammar.

We have shown how our regular analysis techniques can be applied to two-level systems as well as rewriting grammars, and that grammars in both frameworks denote only regular relations. These results open up many new ways of partitioning the account of linguistic phenomena in order to achieve descriptions that are intuitively more satisfying but without introducing new formal power or computational machinery. Karttunen, Kaplan, and Zaenen (1992), for example, argued that certain French morphological patterns can be better described as the composition of two separate two-level grammars rather than as a single one. As another option, an entire two-level grammar can be embedded in place of a single rule in an ordered rewriting system. As long as care is taken to avoid inappropriate complementations and intersections, all such arrangements will denote regular relations and can be implemented by a uniform finite-state transducer mechanism.

8. Conclusion

Our aim in this paper has been to provide the core of a mathematical framework for phonology. We used systems of rewriting rules, particularly as formulated in SPE, to give concreteness to our work and to the paper. However, we continually sought solutions in terms of algebraic abstractions of sufficiently high level to free them from any necessary attachment to that or any other specific theory. If our approach proves useful, it will only be because it is broad enough to encompass new theories and new variations on old ones. If we have chosen our abstractions well, our techniques will extend smoothly and incrementally to new formal systems. Our discussion of two-level rule systems illustrates how we expect such extensions to unfold. These techniques may even extend to phonological systems that make use of matched pairs of brackets. Clearly, context-free mechanisms are sufficient to enforce dependencies between corresponding brackets, but further research may show that accurate phonological description does not exploit the power needed to maintain the balance between particular pairs, and thus that only regular devices are required for the analysis and interpretation of such systems.

An important goal for us was to establish a solid basis for computation in the domain of phonological and orthographic systems. With that in mind, we developed a well-engineered computer implementation of the calculus of regular languages and relations, and this has made possible the construction of practical language processing systems. The common data structures that our programs manipulate are clearly states, transitions, labels, and label pairs—the building blocks of finite automata and transducers. But many of our initial mistakes and failures arose from attempting also to think in terms of these objects. The automata required to implement even the simplest examples are large and involve considerable subtlety for their construction. To view them from the perspective of states and transitions is much like predicting weather patterns by studying the movements of atoms and molecules or inverting a matrix with a Turing machine. The only hope of success in this domain lies in developing an appropriate set of high-level algebraic operators for reasoning about languages and relations and for justifying a corresponding set of operators and automata for computation.

From a practical point of view, the result of the work reported here has been a set of powerful and sometimes quite complex tools for compiling phonological grammars in a variety of formalisms into a single representation, namely a finite-state transducer. This representation has a number of remarkable advantages: (1) The program required to interpret this representation is simple almost to the point of triviality, no matter how intricate the original grammars might have been. (2) That same program can be used to generate surface or textual forms from underlying lexical representations or

to analyze text into a lexical string; the only difference is in which of the two symbols on a transition is regarded as the input and which the output. (3) The interpreter is constant even under radical changes in the theory and the formalism that informed the compiler. (4) The compiler consists almost entirely of an implementation of the basic calculus. Given the operators and data types that this makes available, only a very few lines of code make up the compiler for a particular theory.

Reflecting on the way the relation for a rewriting rule is constructed from simpler relations, and on how these are composed to create a single relation for a complete grammar, we come naturally to a consideration of how that relation should comport with the other parts of a larger language-processing system. We can show, for example, that the result of combining together a list of items that have exceptional phonological behavior with a grammar-derived relation for general patterns is still a regular relation with an associated transducer. If E is a relation for a finite list of exceptional input-output pairs and P is the general phonological relation, then the combination is given by

$$E \cup [Id(\overline{Dom(E)}) \circ P]$$

This relation is regular because E is regular (as is any finite list of pairs); it suppresses the general mapping provided by P for the exceptional items, allowing outputs for them to come from E only. As another example, the finite list of formatives in a lexicon L can be combined with a regular phonology (perhaps with exceptions already folded in) by means of the composition $Id(L) \circ P$. This relation enshrines not only the phonological regularities of the language but its lexical inventory as well, and its corresponding transducer would perform phonological recognition and lexical lookup in a single sequence of transitions. This is the sort of arrangement that Karttunen et al. (1992) discuss. Finally, we know that many language classes are closed under finite-state transductions or composition with regular relations—the images of context-free languages, for example, are context-free. It might therefore prove advantageous to seek ways of composing phonology and syntax to produce a new system with the same formal properties as syntax alone.

Acknowledgments

We are particularly indebted to Danny Bobrow for helpful discussions in the early stages of the research on rewriting systems. Our understanding and analysis of two-level systems is based on very productive discussions with Lauri Karttunen and Kimmo Koskenniemi. We would like to thank John Maxwell, Mary Dalrymple, Andy Daniels, Chris Manning, and especially Kenneth Beesley for detailed comments on earlier versions of this paper. Finally, we are also indebted to the anonymous referees for identifying a number of technical and rhetorical weaknesses. We, of course, are responsible for any remaining errors.

References

Chomsky, Noam, and Halle, Morris (1968). *The Sound Pattern of English.* Harper and Row.

Eilenberg, Samuel (1974). *Automata, Languages, and Machines.* Academic Press.

Harrison, Michael A. (1978). *Introduction to Formal Language Theory.* Addison-Wesley.

Hopcroft, John E., and Ullman, Jeffrey D. (1979). *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley.

Johnson, C. Douglas (1972). *Formal Aspects of Phonological Description.* Mouton.

Kaplan, Ronald M. (1984). "Finite-state models of phonological rule systems." Paper presented to the *First Mathematics of Language Conference*, University of Michigan.

Kaplan, Ronald M. (1985). "Finite-state models of phonological rule systems." Paper presented to the *Workshop on Finite State Morphology*, Center for the Study of Language and Information, Stanford University.

Kaplan, Ronald M. (1988). "Regular models of phonological rule systems." Paper presented to the *Alvey Workshop on Parsing and Pattern Recognition*, Oxford University.

Kaplan, Ronald M., and Kay, Martin (1981). "Phonological rules and finite-state transducers." Paper presented to the *Winter Meeting of the Linguistic Society of America*, New York.

Karttunen, Lauri, and Beesley, Kenneth (1992). "Two-level rule compiler." Technical report, Xerox Palo Alto Research Center, Palo Alto, California.

Karttunen, Lauri; Kaplan, Ronald M.; and Zaenen, Annie (1992). "Two-level morphology with composition." COLING-92, 141–148. Nantes.

Karttunen, Lauri; Koskenniemi, Kimmo; and Kaplan, Ronald M. (1987). "A compiler for two-level phonological rules." In Report No. CSLI-87-108, Center for the Study of Language and Information, Stanford University.

Kay, Martin (1987). "Nonconcatenative finite-state morphology." In *Proceedings, Third European Conference of the Association for Computational Linguistics*, 2–10. Copenhagen.

Kenstowicz, Michael, and Kisseberth, Charles (1979). *Generative Phonology: Description and Theory*. Academic Press.

Klatt, Dennis H. (1980). "Scriber and Lafs: Two new approaches to speech analysis." In *Trends in Speech Recognition*, edited by Wayne Lea, 529–555. Prentice-Hall.

Koskenniemi, Kimmo (1983). "Two-level morphology: A general computational model for word-form recognition and production." Publication No. 11, Department of General Linguistics, University of Helsinki.

Koskenniemi, Kimmo (1985). "Compilation of automata from morphological two-level rules." In *Papers from the Fifth Scandinavian Conference of Computational Linguistics*, 143–149. Helsinki, Finland.

Ritchie, Graeme D. (1992). "Languages generated by two-level morphological rules." *Computational Linguistics*, 18 (1), 41–59.

Ritchie, Graeme D.; Russell, Graham J.; Black, Alan W.; and Pulman, Stephen G. (1992). *Computational Morphology*. MIT Press.

Woods, William A.; Bates, Madeleine; Brown, Geoffrey; Bruce, Bertram C.; Cook, Craig C.; Klovstad, John W.; Makhoul, John; Nash-Webber, Bonnie; Schwartz, Richard; Wolf, Jared; and Zue, Victor (1976). "Speech understanding systems: Final report." Report No. 3438, Bolt Beranek and Newman, Inc., Cambridge, Mass.