Two-Level Rule Compiler

Kenneth R. Beesley and Lauri Karttunen

March 5, 2003

1 Introduction to twolc

twolc, for **Two-L**evel **C**ompiler, is a high-level language for describing morphological alternations as in *fly:flies*, *swim:swimming* and *wiggle:wiggling*. The **twolc** syntax is based on the declarative system of rule constraints, known as TWO-LEVEL RULES, proposed in Kimmo Koskenniemi's 1983 dissertation (Koskenniemi, 1983; Koskenniemi, 1984).

Like the replace rules described in Chapter 3 of (Beesley and Karttunen, 2003), **twolc** rules denote regular relations. But **twolc** rules have a distinct syntax and semantics, and they require the linguist to adopt a different mindset and grammar-writing approach. This chapter describes the syntax of **twolc** source files, the **twolc** compiler interface, and the use of the results within finite-state systems.

At **Xerox**, where developers have a choice, they are increasingly avoiding **twolc** and using replace rules. For the time being, some developers will have to learn **twolc** as well as replace rules, especially those who need to support legacy systems.

twolc rules are always written in a file using a text editor like **xemacs**, **emacs** or **vi**. The **twolc** compiler interface, written in C, provides commands for the syntactic checking of source files, compilation, testing, and writing of the results to file. Two distinct formats of output can be generated: 1) standard **Xerox** binary format suitable for composition, inside **lexc**, with lexicons and other transducers produced with the **Xerox** finite-state tools, and 2) TABULAR format, suitable for use by traditional two-level or "KIMMO" systems, in particular those written with the Lingsoft **TwoL** implementation and Evan Antworth's **PC-KIMMO** (Antworth, 1990), which is available from the Summer Institute of Linguistics (SIL). [1]

The introduction will continue with a bit of history, laying the groundwork for understanding why **twolc** rules were invented and how they work. Special attention will be paid to visualizing how **twolc** rules can fit into an overall grammar.

1.1 History

1.1.1 The origins of finite-state morphology

Traditional phonological grammars, formalized in the 1960s by Noam Chomsky and Morris Halle (Chomsky and Halle, 1968), consisted of an ordered sequence of REWRITE RULES that converted abstract phonological representations into surface forms through a series of intermediate representations. Such rewrite rules have the general form $\alpha \rightarrow \beta / \gamma \_ \delta$ where α, β, γ, and δ can be arbitrarily complex strings or feature-matrices. The rule is read "α is rewritten as β in the environment between γ and δ". In mathematical linguistics (Partee et al., 1993), such rules are called CONTEXT-SENSITIVE REWRITE RULES, and they are more powerful than regular expressions or context-free rewrite rules.

In 1972, C. Douglas Johnson published his dissertation, *Formal Aspects of Phonological Description* (Johnson, 1972), wherein he showed that phonological rewrite

[1] `http://www.sil.org/computing/catalog/pc-kimmo.html`

rules are actually much less powerful than the notation suggests. Johnson observed that while the same context-sensitive rule could be applied several times recursively to its own output, phonologists have always assumed implicitly that the site of application moves to the right or to the left of the string after each application. For example, if the rule $\alpha \rightarrow \beta / \gamma \_ \delta$ is used to rewrite the string $\gamma\alpha\delta$ as $\gamma\beta\delta$, any subsequent application of the same rule must leave the β part unchanged, affecting only γ or δ. Johnson demonstrated that the effect of this constraint is that the pairs of inputs and outputs produced by a phonological rewrite rule can be modeled by a finite-state transducer. Unfortunately, this result was largely overlooked at the time and was rediscovered by Ronald M. Kaplan and Martin Kay around 1980 (Kaplan and Kay, 1981; Kaplan and Kay, 1994). Putting things into a more algebraic perspective than Johnson, Kaplan and Kay showed that phonological rewrite rules describe REGULAR RELATIONS. By definition, a regular relation can be represented by a finite-state transducer.

Johnson was already aware of an important mathematical property of finite-state transducers (Schützenberger, 1961): there exists, for any pair of transducers applied sequentially, an equivalent single transducer. Any cascade of rule transducers can in principle be composed into a single transducer that maps lexical forms directly into the corresponding surface forms, and vice versa, without any intermediate representations. Later, Kaplan and Kay had the same idea, illustrated in Figure 1.

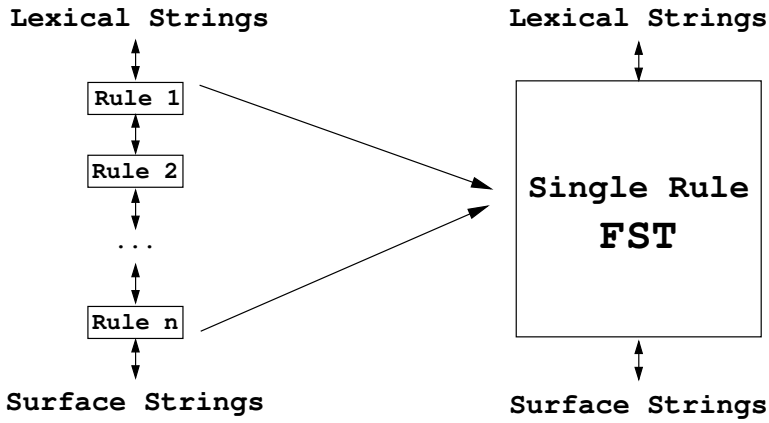

Figure 1: A Cascade of Finite-State Rewrite Rules Composed into an Single Equivalent **FST**

These theoretical insights did not immediately lead to practical results. The development of a compiler for rewrite rules turned out to be a very complex task. It became clear that building a compiler required as a first step a complete implementation of basic finite-state operations such as union, intersection, complementation,

and composition. Developing a complete finite-state calculus was a challenge in itself on the computers that were available at the time.

Another reason for the slow progress may have been that there were persistent doubts about the practicality of the approach for morphological *analysis*. Traditional phonological rewrite rules describe the correspondence between lexical forms and surface forms as a uni-directional, sequential mapping from lexical forms to surface forms. Even if it was possible to model the *generation* of surface forms efficiently by means of finite-state transducers, it was not evident that it would lead to an efficient analysis procedure going in the reverse direction, from surface forms to lexical forms.

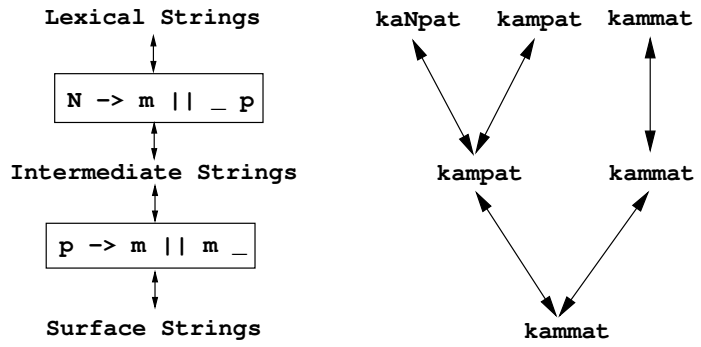

Figure 2: Rules Mapping *kammat* to *kaNpat, kampat, kammat*

The *kaNpat* exercise (see Beesley&Karttunen (2003), Section 3.5.3) is a simple illustration of the problem. The transducers compiled from the two **xfst** replace rules,

```
N -> m || _ p
```

and

```
p -> m || m _
```

map the lexical form "`kaNpat`" unambiguously down to "`kammat`", with "`kampat`" as the intermediate representation (see Beesley&Karttune (2003), Figure 3.12). However if we apply the same transducers in the upward direction to the input "`kammat`", we get the three results "`kaNpat`", "`kampat`" and "`kammat`" shown in Figure 2. The reason is that the surface form "`kammat`" has two potential sources on the intermediate level; the downward application of the `p -> m || m _` rule maps both "`kampat`" and "`kammat`" to the same surface form. The intermediate form "`kampat`" in turn could come either from "`kampat`" or from "`kaNpat`" by the downward application of the `N -> m || _ p` rule. The two rule transducers

are unambiguous when applied in the downward direction but ambiguous when applied in the upward direction.

This asymmetry is an inherent property of the generative approach to phonological description. If all the rules are deterministic and obligatory and if the order of the rules is fixed, then each lexical form generates only one surface form. But a surface form can typically be generated in more than one way, and the number of possible analyses grows with the number of rules involved. Some of the analyses may eventually turn out to be invalid because the putative lexical forms, say "`kammat`" and "`kampat`" in this case, might not exist in the language. But in order to look them up in the lexicon, the system first had to complete the analysis, producing all the phonological possibilities. The lexicon was assumed to be a separate module that was used subsequently to accept or reject the possible analyses. Depending on the number of rules involved, a surface form could easily have dozens or hundreds of potential lexical forms, even an infinite number in the case of certain deletion rules.

Although the generation problem had been solved by Johnson, Kaplan and Kay, at least in principle, the problem of efficient morphological analysis in the Chomsky-Halle paradigm was still seen as a formidable challenge. As counterintuitive as it was from a psycholinguistic point of view, it appeared that analysis was much harder computationally than generation. Composing all the rule transducers into a single one would not solve the "over-analysis" problem. Because the resulting single transducer is equivalent to the original cascade, the ambiguity remains.

The solution to the over-analysis problem should have been obvious: to formalize the lexicon itself as a finite-state transducer and compose the lexicon with the rules. With the lexicon included in the composition, all the spurious ambiguities produced by the rules are eliminated at compile time. The runtime analysis becomes more efficient because the resulting single transducer contains only lexical forms that actually exist in the language.

The idea of composing the lexicon and the rules together is not mentioned in Johnson's book or in the early **Xerox** work. Although there obviously had to be some interface relating a lexicon component to a rule component, these were traditionally thought of as different types of objects. Furthermore, rewrite rules were seen as applying to individual word forms; the idea of applying them simultaneously to a lexicon as a whole required a new mindset and computational tools that were not yet available.

The observation that a single finite-state transducer could encode the inventory of valid lexical forms as well as the mapping from lexical forms to surface forms took a while to emerge. When it first appeared in print (Karttunen et al., 1992), it was not in connection with traditional rewrite rules but with an entirely different finite-state formalism that had been introduced in the meantime, Kimmo Koskenniemi's TWO-LEVEL RULES (Koskenniemi, 1983).

Figure 3: An Example of Two-Level Constraints

1.1.2 Two-Level Morphology

In the spring of 1981 when Kimmo Koskenniemi came to the USA for a visit, he learned about Kaplan and Kay's finite-state discovery. (They weren't then aware of Johnson's 1972 publication.) **Xerox** had begun work on the finite-state algorithms, but they would prove to be many years in the making. Koskenniemi was not convinced that efficient morphological analysis would ever be practical with generative rules, even if they were compiled into finite-state transducers. Some other way to use finite automata might be more efficient.

Back in Finland, Koskenniemi invented a new way to describe phonological alternations in finite-state terms. Instead of cascaded rules with intermediate stages and the computational problems they seemed to lead to, rules could be thought of as statements that directly constrain the surface realization of lexical strings. Multiple rules would be applied not sequentially but in parallel. Each rule would constrain a certain lexical/surface correspondence and the environment in which the correspondence was allowed, required, or prohibited. For his 1983 dissertation, Koskenniemi constructed an ingenious implementation of his constraint-based model that did not depend on a rule compiler, composition or any other finite-state algorithm, and he called it TWO-LEVEL MORPHOLOGY. Two-level morphology is based on three ideas:

- Rules are symbol-to-symbol constraints that are applied in parallel, not sequentially like rewrite rules.
- The constraints can refer to the lexical context, to the surface context, or to both contexts at the same time.
- Lexical lookup and morphological analysis are performed in tandem.

To illustrate the first two principles we can turn back to the *kaNpat* example again. A two-level description of the lexical-surface relation is sketched in Figure 3. As the lines indicate, each symbol in the lexical string "`kaNpat`" is paired with its realization in the surface string "`kammat`". Two of the symbol pairs in Figure 3 are constrained by the context marked by the associated box. The **N:m** pair is *restricted* to the environment having an immediately following **p** on the lexical side. In fact the constraint is tighter. In this context, all other possible realizations

of a lexical **N** are *prohibited*. Similarly, the **p:m** pair requires the preceding surface **m**, and no other realization of **p** is allowed here. The two constraints are independent of each other. Acting in parallel, they have the same effect as the cascade of the two rewrite rules in Figure 2. In Koskenniemi's notation, these rules are written as `N:m <=> _ p:` and `p:m <=> :m _` , where `<=>` is an operator that combines a context restriction with the prohibition of any other realization for the lexical symbol of the pair. The **p** followed by a colon in the right context of first rule, `p:`, indicates that **p** refers to a lexical symbol; the colon preceding **m** in the left context of the second rule, `:m`, indicates that **m** is a surface symbol.

Figure 4: A Two-Level View of *y∼ie* Alternation in English

Two-level rules may refer to both sides of the context at the same time. The *y∼ie* alternation in English plural nouns could be described by two rules: one realizes **y** as **i** in front of an epenthetic **e**; the other inserts an epenthetic **e** between a lexical consonant-**y** sequence and a morpheme boundary (**+**) that is followed by an **s**. Figure 4 illustrates the **y:i** and **0:e** constraints.

Note that the **e** in Figure 4 is paired with a **0** (= zero) on the lexical level. Formally the rules are expressed as `y:i <=> _ 0:e` and `0:e <=> y: _ %+:`. From the point of view of two-level rules, zero is a symbol like any other; it can be used to constrain the realization of other symbols. In fact, all the other rules must "know" where zeros may occur. In two-level rules, the zeros are not epsilons, even though they are treated as such when two-level rules are eventually applied to strings.

Like replace rules, two-level rules describe regular relations; but there is an important difference. Because the zeros in two-level rules are in fact ordinary symbols, a two-level rule represents an *equal-length relation* (see Beesley&Karttunen (2003), Section 2.3). Counting the "hard zeros", each string and its related strings always have exactly the same length. This has an important consequence: although transducers cannot in general be intersected (see Beesley&Karttunen (2003), Section 2.3.3), equal-length transducers are a special case, and so Koskenniemi's constraint transducers *can* be intersected. In fact, when a set of two-level transducers are applied in parallel, the apply routine in a KIMMO-style system simulates the intersection of the rule automata and the composition of the input string with the virtual constraint network (see Beesley&Karttunen (2003), Section 1.6).

Figure 5 illustrates the upward application of the **N:m** and **p:m** rules sketched in Figure 3 to the input "`kammat`". At each point in the process, all lexical can-

didates corresponding to the current surface symbol are considered one by one. If both rules accept the pair, the process moves on to the next point in the input. In the situation shown in Figure 5, the pair **p:m** will be accepted by both rules. The **N:m** rule accepts the pair because the **p** on the lexical side is required to license the **N:m** pair that has tentatively been accepted at the previous step. The **p:m** rule accepts the **p:m** pair because the preceding pair has an **m** on the surface side.

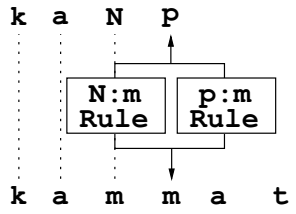

Figure 5: Parallel Application of Two-Level Rules for Analysis

When the pair in question has been accepted, the apply routine moves on to consider the next input symbol and eventually comes back to the point shown in Figure 5 to consider other possible lexical counterparts of a surface **m**. They will all be rejected by the **N:m** rule, and the apply routine will return to the previous **m** in the input to consider other alternative lexical counterparts for it such as **p** and **m**. At every point in the input the apply routine must also consider all possible deletions, that is, pairs such as **+:0** and **e:0** that have a zero on the input side.

Applying the rules in parallel does not in itself solve the over-analysis problem discussed in the previous section. The two constraints sketched above allow "`kammat`" to be analyzed as "`kaNpat`", "`kampat`", or "`kammat`". However, the problem is easy to manage in a system that has only two levels; the possible upper-side symbols are constrained at each step by consulting the lexicon, which is itself implemented as a kind of network. In Koskenniemi's two-level system, lexical lookup and the analysis of the surface form are performed in tandem. In order to arrive at the point shown in Figure 5, we must have traversed a path in the lexicon that contains the lexical string in question, see Figure 6. The lexicon thus acts as a continuous lexical filter on the analysis. The analysis routine only considers symbol pairs whose lexical side matches one of the outgoing arcs of the current state of the lexicon network.

In Koskenniemi's 1983 system, the lexicon was represented as a forest of tries (also known as letter trees), tied together by continuation-class links from leaves of one tree to roots of another tree or trees in the forest.[2] Koskenniemi's lexicon can be thought of as a partially deterministic, unminimized simple network. In the **Xerox lexc** tool, the lexicon is actually compiled into a minimized network, typi-

[2]The TEXFIN analyzer developed at the University of Texas at Austin (Karttunen et al., 1981) had the same lexicon architecture.

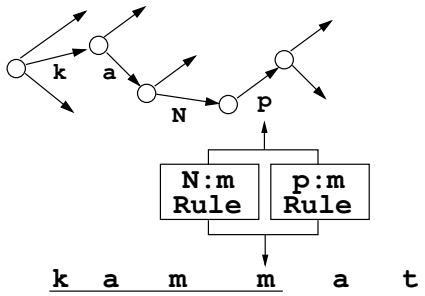

Figure 6: Following a Path in the Lexicon. The lexicon acts as a continuous filter on the analyses produced by a traditional two-level system.

cally a transducer, but the filtering principle is the same. The **lookup** utility in **lexc** matches the lexical string proposed by the rules directly against the lower side of the lexicon. It does not pursue analyses that have no matching lexical path. Furthermore, the lexicon may be composed with the rules at compile time to produce a single transducer that maps surface forms directly to lexical forms, and vice versa (see the Section 4.5.1 in Beesley&Karttunen (2003)).

Koskenniemi's two-level morphology was the first practical general model in the history of computational linguistics for the analysis of morphologically complex languages. The language-specific components, the rules and the lexicon, were combined with a universal runtime engine applicable to all languages. The original implementation was primarily intended for analysis, but the model was in principle bidirectional and could be used for generation.

1.1.3 Linguistic Issues

Although the two-level approach to morphological analysis was quickly accepted as a useful practical method, the linguistic insight behind it was not picked up by mainstream linguists. The idea of rules as parallel constraints between a lexical symbol and its surface counterpart was not taken seriously at the time outside the circle of computational linguists. Many arguments had been advanced in the literature to show that phonological alternations could not be described or explained adequately without sequential rewrite rules. It went largely unnoticed that two-level rules could have the same effect as ordered rewrite rules because two-level rules allow the realization of a lexical symbol to be constrained either by the lexical side or by the surface side. The standard arguments for rule ordering were based on the *a priori* assumption that a rule could refer only to the input context.

But the world has changed. Current phonologists, writing in the framework of OT (Optimality Theory), are sharply critical of the "serialist" tradition of ordered rewrite rules that Johnson, Kaplan and Kay wanted to formalize (Prince and Smolen-

sky, 1993; Kager, 1999; McCarthy, 2002).[3] In a nutshell, OT is a two-level theory with *ranked* parallel constraints. Many types of optimality constraints can be represented trivially as two-level rules. In contrast to Koskenniemi's "hard" constraints, optimality constraints are "soft" and violable. There are of course many other differences. Most importantly, OT constraints are meant to be universal. The fact that two-level rules can describe orthographic idiosyncrasies such as the *y*∼*ie* alternation in English with no appeal to universal principles makes the approach uninteresting from the OT point of view.[4]

1.1.4 Two-Level Rule Compilers

In his 1983 dissertation, Koskenniemi introduced a formalism for two-level rules. The semantics of two-level rules were well-defined but there was no rule compiler available at the time. Koskenniemi and other early practitioners of two-level morphology had to compile their rules *by hand* into finite-state transducers. This is tedious in the extreme and demands a detailed understanding of transducers and rule semantics that few human beings can be expected to grasp. A complex rule with multiple overlapping contexts may take hours of concentrated effort to compile and test, even for an expert human "compiler". In practice, linguists using two-level morphology consciously or unconsciously tended to postulate rather surfacy lexical strings, which kept the two-level rules relatively simple.

Although two-level rules are formally quite different from the rewrite rules studied by Kaplan and Kay, the basic finite-state methods that had been developed for compiling rewrite-rules were applicable to two-level rules as well. In both formalisms, the most difficult case is a rule where the symbol that is replaced or constrained appears also in the context part of the rule. This problem Kaplan and Kay had already solved by an ingenious technique for introducing and then eliminating auxiliary symbols to mark context boundaries. Another fundamental insight was the encoding of contextual requirements in terms of double negation. For example, a constraint such as "*p* must be followed by *q*" can be expressed as "it is not the case that something ending in *p* is not followed by something starting with *q*." In Koskenniemi's formalism, the same constraint is expressed by the rule `p => _ q`.

In the summer of 1985, when Koskenniemi was a visitor at the Center for the Study of Language and Information (CSLI) at Stanford, Kaplan and Koskenniemi worked out the basic compilation algorithm for two-level rules. The first two-level rule compiler was written in InterLisp by Koskenniemi and Karttunen in 1985-87 using Kaplan's implementation of the finite-state calculus (Koskenniemi, 1986; Karttunen et al., 1987). The current C-version of the compiler, based on Karttunen's 1989 Common Lisp implementation, was written by Lauri Karttunen, Todd Yampol and Kenneth R. Beesley in consultation with Kaplan at **Xerox PARC** in 1991-92 (Karttunen and Beesley, 1992). The landmark 1994 article by Kaplan and Kay

[3] The term SERIAL, a pejorative term in an OT context, refers to SEQUENTIAL rule application.

[4] Finite-state approaches to Optimality Theory have been explored in several recent articles (Eisner, 1997; Frank and Satta, 1998; Karttunen, 1998).

on the mathematical foundations of finite-state linguistics gives a compilation algorithm for phonological rewrite rules and for Koskenniemi's two-level rules.[5]

The **Xerox** two-level compiler has been used to compile rules for large-scale morphological analyzers for French, English, Spanish, Portuguese, Dutch, Italian and many other languages.

1.1.5 Implementations of Two-Level Morphology

The first implementation of Two-Level Morphology (Koskenniemi, 1983) was quickly followed by others. The most influential implementation was by Lauri Karttunen and his students at the University of Texas (Karttunen, 1983; Gajek et al., 1983; Dalrymple et al., 1983). Published accounts of this project inspired many copies and variations, including those by Beesley (Beesley, 1989; Beesley, 1990). A copyrighted but freely distributed implementation of classic Two-Level Morphology, called **PC-KIMMO**, available from the Summer Institute of Linguistics (Antworth, 1990), runs on PCs, Macs and Unix systems.[6]

In Europe, two-level morphological analyzers became a standard component in several large systems for natural-language processing such as the British Alvey project (Black et al., 1987; Ritchie et al., 1987; Ritchie et al., 1992), SRI's CLE Core Language Engine (Carter, 1995), the ALEP Natural Language Engineering Platform (Pulman, 1991) and the MULTEXT project (Armstrong, 1996). ALEP and MULTEXT were funded by the European Commission. The MMORPH morphology tool (Petitpierre and Russel, 1995) built at ISSCO for MULTEXT is now available under GNU Public License.[7]

Some of these systems were implemented in Lisp (Alvey), some in Prolog (CLE, ALEP), some in C (MMORPH). They were based on simplified two-level rules, the so-called PARTITION-BASED formalism (Ruessink, 1989), which was claimed to be easier for linguists to learn than the original Koskenniemi notation. But none of these systems had a finite-state rule compiler. Another difference was that morphological parsing could be constrained by feature unification. Because the rules were interpreted at runtime and because of the unification overhead, these systems were not very efficient, and two-level morphology acquired, undeservedly, a reputation for being slow. MMORPH solves the speed problem by allowing the user to run the morphology tool off-line to produce a database of fully inflected word forms and their lemmas. A compilation algorithm has since been developed for the partition-based formalism (Grimley-Evans et al., 1996), but to our knowledge there is no publicly available compiler for it.

[5] The Kaplan and Kay article appeared many years after the work on the two-level compiler was completed but before the implementation of the so-called REPLACE RULES in the current **Xerox** regular-expression compiler. The article is accurate on the former topic, but the compilation algorithm for replace rules (Karttunen, 1995; Karttunen, 1996; Kempe and Karttunen, 1996) differs in many details from the compilation method for rewrite rules described by Kaplan and Kay.

[6] `http://www.sil.org/computing/catalog/pc-kimmo.html`

[7] `http://packages.debian.org/stable/misc/mmorph.html`

A considerable amount of work has been done, and continues to be done, in the general framework of Two-Level Morphology, and the **twolc** compiler has made that work much less onerous. The newer **Xerox** replace rules, which are part of an extended regular-expression language and are compiled using the regular-expression compiler in **xfst**, have largely supplanted **twolc** rules in some applications. For the time being, some linguistic developers will have to master both replace rules and **twolc** rules.

1.2 Visualizing twolc Rules

At **Xerox**, **twolc** rules are compiled automatically into rule transducers and are typically composed on the lower side of lexicon transducers made with **lexc**. As shown in Figure 7, the two-level rules conceptually map between strings on the lower side of the lexicon and real surface strings. That is, the upper side of each **twolc** rule transducer refers to the lower-side language of the lexicon, and the lower side of each rule transducer is the surface language. The **twolc** rules are arranged in one horizontal level and apply in parallel—the order of the rules in a **twolc** file is therefore not significant.

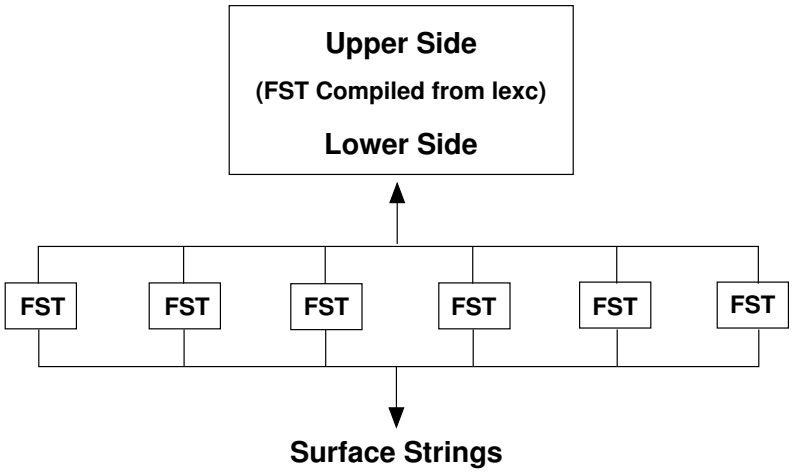

Figure 7: **twolc** Rules Relate Lower-Side Lexicon Strings and Surface Strings. **twolc** rules apply in parallel, and their relative order is not significant.

This model of parallel, simultaneously applied **twolc** rules must be sharply contrasted with the serial or cascade model of replace rules, as shown in Figure 8. Replace rules are arranged vertically, mapping strings from the lower side of the lexicon through a cascade of steps down to the surface level. The following differences must be noted and understood:

- Replace rules are organized vertically, in a cascade, and so they potentially

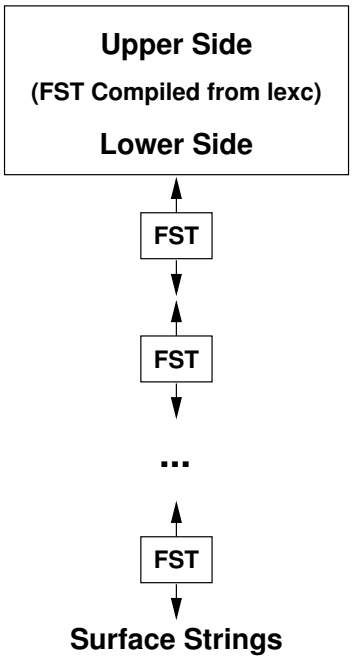

Figure 8: A Cascade of Replace Rules. In a cascade, rules can "feed" each other, and so the proper ordering of the rules is often crucial.

feed each other. In contrast, a grammar of **twolc** rules is organized horizontally; the **twolc** rules apply in parallel and do not feed each other.

- Because they can feed each other, replace rules must usually be ordered carefully. A grammar of **twolc** rules, on the other hand, can be written in any order without affecting the output at all. Rule order in a **twolc** grammar is formally insignificant.[8]

- Replace rules conceptually produce many intermediate languages (levels) when mapping between the lower side of the lexicon and the final surface language. **twolc** rules each map directly from the lower-side of the lexicon to the surface in one step.

- **twolc** rules conceptually apply simultaneously; this avoids the ordering problems but means that sets of **twolc** rules must be carefully written to avoid nasty and often mysterious conflicts among rules. Correct, non-conflicting

[8] In a **PC-KIMMO**-like system, in which each two-level rule is stored as a separate transducer and consulted individually at each step of analysis, rule order can affect the performance but not the output of a system. In particular, performance can be improved by identifying the rules that most frequently block analysis paths in practice, and ordering them early in the set so that they are checked first.

twolc rule sets are notoriously difficult to write when some of the rules perform deletion or, especially, epenthesis.

- Replace rules are compiled using the normal **xfst** regular-expression compiler, which is invoked by the **read regex** and **define** commands; replace rules are just an extension of the **Xerox** regular-expression metalanguage. **twolc** rules must be compiled using the dedicated **twolc** compiler that can be accessed only through the **twolc** interface.

A system of replace rules is relatively easy to check and modify because each rule can be applied individually. The output from one rule can be typed as input to the next rule and the effect of a whole cascade of replace rules can be checked step by step. When we move to **twolc** rules, however, the semantics of the rules demand that we conceive of them always as simultaneously applied constraints on the relation between the lexical language and the surface language. Because **twolc** rules are designed to apply in parallel, it is difficult to test them individually.

Written correctly, grammars of unordered **twolc** rules can perform the same mappings that require carefully ordered cascades of replace rules. Conversely, any **twolc** grammar can be rewritten as a grammar of replace rules; this formal equivalence is guaranteed by the fact that both formalisms are just metalanguages for describing regular relations. The practical choice is therefore one of notational perspicuity, human ease of use, and human taste. **Xerox** developers, given a choice between **twolc** rules and replace rules, are increasingly choosing replace rules.

1.3 Plan of Attack

The presentation will continue as follows:

- Section 2 (Basic **twolc** Syntax) describes the basic syntax and interface commands needed to get started, consolidating the concepts with some practical exercises.
- Section 3 (Full **twolc** Syntax) completes the formal description.
- Section 4 (The Art and Craft of Writing **twolc** Grammars) discusses some useful tricks and idioms.
- Section 5 (Debugging **twolc** Rules) explains why **twolc** rules conflict with each other and how to understand and resolve the clashes. The troublesome epenthesis rules and diacritics, as well as classic programming errors, are also discussed.
- Section 6 (Final Reflections on Two-Level Rules) looks at the real choice between using **twolc** or replace rules in various applications. The possibility of upward-oriented two-level rules is also explored.

2 Basic twolc Syntax

This section contains a description of the basic **twolc** syntax that you need to get started. **twolc** source files are created using a text editor such as **emacs**, **xemacs** or **vi**. Each **twolc** file consists of named SECTIONS, two of which, the `Alphabet` section and the `Rules` section, are obligatory.

2.1 Alphabet

The `Alphabet` keyword introduces the obligatory Alphabet section, which must appear at the top of the **twolc** source file. The Alphabet section must contain at least one declared *upper*:*lower* symbol pair and is terminated by a semicolon.

```
Alphabet  a:a ;
```

Figure 9: A Minimal Alphabet Section. The Alphabet section is required and must appear at the top of the **twolc** source file.

twolc grammars operate relative to an alphabet of character pairs such as **a:a**, **a:0**, **0:u**, **e:i**, etc. In **twolc** rules, the notation **z:y** must always have a single symbol on the left and a single symbol on the right of the colon. As in regular expressions, the symbol to the left of the colon is the upper-side symbol, and the symbol on the right of the colon is the lower-side symbol. When only a single symbol **a** is written, as in Figure 10, it is automatically interpreted by **twolc** as a shorthand notation for **a:a**.

```
Alphabet  a ;
```

Figure 10: Declaration of **a** is Equivalent to **a:a**

The alphabet in a **twolc** grammar always consists of SYMBOL PAIRS, sometimes called FEASIBLE PAIRS in two-level systems. **twolc** uses the **u:d** notation to designate a symbol pair with **u** on the upper side and **d** on the lower side. Both **u** and **d** must be single symbols in a **twolc** grammar.

By default, **twolc** assumes that the identity symbol pairs, **a:a**, **b:b**, **c:c**, **d:d**, etc., are part of the alphabet; they do not normally have to be declared. Alphabet pairs

like **e:i** and **h:0**, representing phonological or orthographical alternations, can be declared explicitly in the `Alphabet` section, or they can be declared implicitly simply by using them in a rule.

The assumptions about default identity pairs, such as **a:a**, **b:b**, **c:c**, etc., being in the alphabet are overridden by the explicit "mentioning" of particular symbols. For example, the declaration of the symbol pair **a:e** in Figure 11 involves mentioning both **a** and **e**, and so it overrides the default assumption that **a:a** and **e:e** are possible character pairs in the alphabet.

```
Alphabet a:e  ;
```

Figure 11: Declaring **a:e** Suppresses the Default Declaration of **a:a** and **e:e**

In practice, declaring **a:e** by itself causes **twolc** to conclude that **a** can appear only on the lexical side and that **e** can appear only on the surface side. If you wish to declare **a:e** and yet retain **a:a** and **e:e** as possible symbol pairs, you must then declare **a:a** and **e:e** as well as in Figure 12; alternatively **a:a** and **e:e** are declared implicitly if they are used anywhere in a rule.

```
Alphabet a:e a e ;
```

Figure 12: Regaining **a:a** and **e:e** by Overt Declaration

In practice, the alphabet in a **twolc** grammar is often a source of errors and mystery, especially when a mistake in a rule inadvertently declares an unintended symbol pair.

2.2 Basic Rules

The `Rules` keyword introduces the obligatory Rules section, which must contain at least one rule.

2.2.1 Rule Syntax

The most commonly used **twolc** rules are built on the template shown in Figure 13. Each **twolc** rule must be preceded by a unique name in double quotes. The *Center*

```
"Unique Rule Name"
  Center <=> LeftContext _ RightContext ;
```

Figure 13: The Most Commonly Used **twolc** Rule Template

part on the left side of a **twolc** rule typically consists of a single symbol pair like **u:d**, where **u** is the upper-level symbol and **d** is the lower-level symbol.

```
Rules

  "Rule 1"
    s:z <=> Vowel _ Vowel ;
```

Figure 14: A Simple Rules section with one Two-Level Rule. The keyword *Rules* introduces the section, which must contain one or more rules. Each rule must have a unique name.

The *Center* may also be a union of two or more symbol pairs, e.g. `[u1:d1 | u2:d2]`, that are subject to identical constraints. For example, to indicate that the voiced stops **b**, **d**, and **g** are all realized as unvoiced in the same context (e.g. at the end of the word, as in some Germanic languages), one would write the *Center* as `[ b:p | d:t | g:k ]` as in Figure 15.

```
  "Rule 2"
    [ b:p | d:t | g:k ] <=>  _  .#. ;
```

Figure 15: A Two-Level Rule with a Complex Left-Hand Side

After the *Center* comes an operator; the most commonly used is the <=> or double-arrow operator, typed as a left angle-bracket, an equal sign, and a right angle-bracket. After the operator, the *LeftContext* and *RightContext* are arbitrarily complex **twolc** regular expressions that surround an underscore (_) indicating the environment in which the *Center* relation is constrained. The *LeftContext* and/or the *RightContext* may be empty. There may be multiple contexts, and each is terminated with a semicolon as shown in Figure 16.

One or more rules may appear in the `Rules` section. The order of the rules in a **twolc** file has no formal effect on the functioning of the grammar.

```
"Rule 3"
    n:m <=>  .#. _   ;
               u _ i ;
               i _ u ;
```

Figure 16: A Two-Level Rule with a Multiple Contexts. Each context must be terminated with a semicolon.

2.2.2 twolc Rule Operators

In addition to the double-arrow <=> **twolc** rule operator, which is by far the most frequently used in practice, there are also the right-arrow =>, the left arrow <=, and the negated left-arrow /<= operators. All four operators are exemplified and explained in Table 1.

The key to success in reading, writing, and debugging **twolc** rules is to know the semantics of the rule operators by heart. As unfashionable as it may seem, the student is urged to commit Table 1 to memory. There is no avoiding the fact that the mastery of **twolc** rules is difficult for many students, but it's impossible for those who do not learn the semantics of the rule operators.

Figure 17 shows examples of rules with the four different operator types and the string pairs that they allow and block; the blocked string pairs are crossed out. After learning the semantics of **twolc** rule operators, you should be able to explain why the crossed-out string pairs are blocked by each rule, and why the other string pairs are allowed.

Note that in **twolc** rules the left-arrow and right-arrow constraints are not symmetrical.

2.2.3 twolc Rule Contexts

Complex left contexts and right contexts are built using a syntax that resembles **xfst** regular expressions in many, but not all, ways. The following are the main points of **twolc** regular expressions:

- A notation `u:d` is a regular expression that denotes the relation of upper-side **u** to lower-side **d**. In **twolc** the colon notation `u:d` must always have at most a single symbol on each side of the colon.
- **twolc** contexts are always two-level. If a rule context includes a symbol `x` written alone, it is interpreted as `x:x`. The context written `p:` matches any

	Positive Reading	***Negative Reading***
a:b <=> l _ r ;	1. If the symbol pair a:b appears, it must be in the context l _ r. 2. If lexical a appears in the context l _ r, then it must be be realized on the surface as b.	1. If the symbol pair a:b appears outside the context l _ r, FAIL. 2. If lexical a appears in the context l _ r and is realized as anything other than b, FAIL.
a:b => l _ r ;	If the symbol pair a:b appears, it must be in the context l _ r.	If the symbol pair a:b appears outside the context l _ r, FAIL.
a:b <= l _ r ;	If lexical a appears in the context l _ r, it must be realized on the surface as b.	If lexical a appears in the context l _ r and is realized as anything other than b, FAIL.
a:b /<= l _ r ;	Lexical a is never realized as b in the context l _ r .	If lexical a is realized as b in the context l _ r, FAIL.

Table 1: **twolc** Rule Operator Semantics

symbol pair in the alphabet having **p** on the upper side. The context written `:q` matches any symbol pair in the alphabet having **q** on the lower side. The notation `:`, i.e. a colon with a space on each side, matches any symbol pair in the alphabet. The notation `?` also matches any symbol pair in the alphabet.

- The left side of the left context and the right side of the right context are extended to infinity by concatenating the relation `:*` on the appropriate side of each context, as shown in Figure 18.

- Bracketing can be used as in **xfst** regular expressions: `[b]` is equivalent to `b`. An empty pair of brackets, `[]`, denotes the empty relation that maps the empty string to itself.

- The `0` (zero) in **twolc** rules denotes not the empty string (as in **xfst**) but a special "hard zero" symbol. More will be said about this below. (As part of a larger expression like `007`, the zero is not hard.) The expression `b:0` denotes the relation of the upper-side symbol **b** to the lower-side hard-zero symbol **0**, which, within **twolc** rules, is an ordinary symbol just like **b**.

- The concatenation of regular expressions `X` and `Y` is notated `X Y`, i.e. separated by white space and without any overt operator.

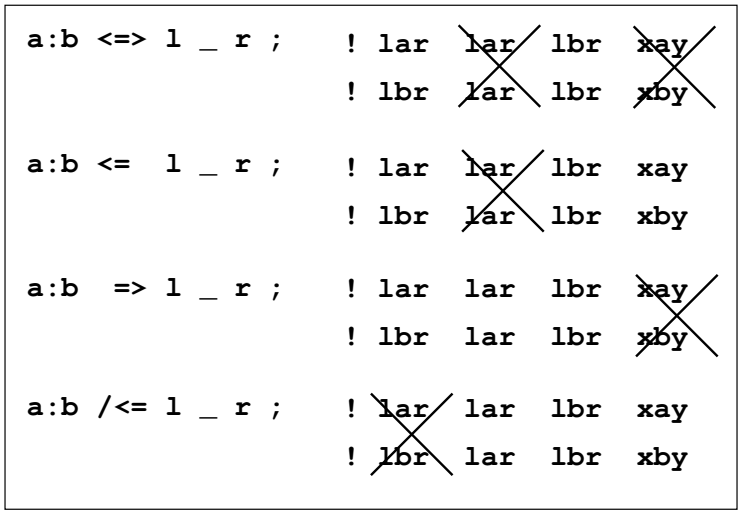

Figure 17: Rule Constraints Allow Certain String Pairs and Disallow Others. Review the semantics of two-level rules until you understand why each rule blocks and allows what it does.

- The union of regular expressions `X` and `Y` is notated `X | Y` . In **twolc**, both curly brackets and square brackets can be used to surround unioned expressions.
- Optionality is indicated by surrounding an expression with parenthesis: `(X)` is equivalent to `[ X | [] ]`.
- The Kleene Star (`*`), meaning zero or more iterations, can be postfixed to an expression, e.g. `X*`.
- The Kleene Plus (`+`), meaning one or more iterations, can be postfixed to an expression, e.g. `X+`.
- The notation `X^n`, where *n* is an integer, denotes *n* iterations: e.g. `X^3` is equivalent to `[X X X]`.
- The notation `X^n,m`, where *n* and *m* are integers, denotes *n* to *m* iterations.[9]
- The notation `X/Y` denotes the language `X`, ignoring any intervening strings from language `Y`.
- `\X` denotes the union of all the symbol pairs in the alphabet excluding pairs that belong to `X`. E.g. `\s:z` will match any single symbol pair in the alphabet except **s:z**.

[9] In **xfst** regular expressions, the notation is slightly different: `X^{n,m}`.

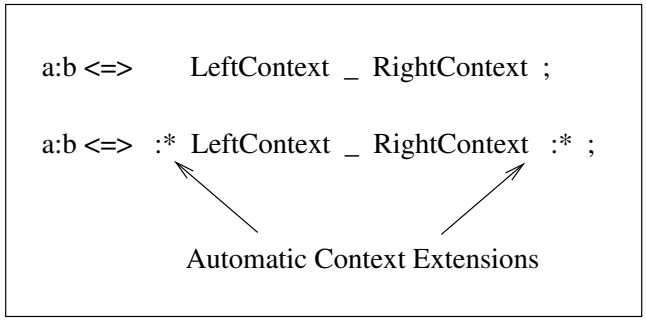

Figure 18: The Left and Right Contexts are Automatically Expanded by the Compiler with `:*`, the Universal Language over the Alphabet

- `$X` denotes the relation of all string pairs containing `X`. `$a:b` is equivalent to `[?* a:b ?*]`.
- Punctuation symbols normally interpreted as regular-expression operators can be unspecialized by preceding them with a percent sign; e.g. `%+` denotes the literal plus sign rather than the Kleene Plus operator. The notation `%0` denotes the literal digit symbol zero (parallel to `1`, `2`, `3`, etc.) rather than the hard zero.
- The notation `.#.` denotes the absolute beginning or absolute end of a string. This notation can appear in rule contexts, but it is not a symbol per se.

Alphabets and contexts in **twolc** grammars are always two-level.

In **twolc** rules, contexts are always two-level, matching strings on both the lexical (upper) and surface (lower) sides. Although two-level regular expressions denote relations, and although relations cannot usually be intersected, complemented or subtracted, **twolc** relations are a special case that do allow these operations. The following notations are therefore valid in **twolc** regular expressions.

- Where `X` and `Y` denote **twolc** relations, `X - Y` denotes the relation `X` not including any relation in `Y`.
- Where `X` and `Y` denote **twolc** relations, `X & Y` denotes the intersection of `X` and `Y`.
- Where `X` denotes a **twolc** relation, `~X` denotes the complement of that relation.

These operations are legal in the **twolc** regular-expression language because the hard zero in pairs such as **e:0** is treated as an ordinary symbol. Consequently **twolc** rules always denote regular equal-length relations, which are closed even under complementation, intersection, and subtraction.

2.3 Thinking in twolc

The way to go about writing **twolc** rules is to

1. Write out a lexical string. This will typically be a string from the lower side of a lexicon transducer created with **lexc**. In our *kaNpat* example, the lexical string is "`kaNpat`" itself (see Beesley&Karttunen (2003), Section 3.5.3).

   ```
   kaNpat
   ```

2. Then write out, under the lexical string, the ultimate surface string that you want to generate using the entire grammar of rules.

   ```
   Lexical:  kaNpat
   Surface:  kammat
   ```

3. Align the two strings, symbol by symbol. Pad out the strings with hard zeros if necessary until they are of *exactly* the same length. Where you put the zeros is ultimately up to you, but be consistent and maximize the linguistic motivations for your choices. You will end up with a pair of strings consisting of a sequence of symbol pairs. These are the two levels of a **twolc** or any two-level morphology. The following lineup might be appropriate for a language where an underlying "`banay+at`" string is realized as "`banat`" on the surface.

   ```
   Lexical:  banay+at
   Surface:  ban000at
   ```

4. Identify the ALTERNATIONS or discrepancies between the two strings. Write **twolc** rules as necessary to allow and constrain the symbol pairs that account for the alternations. The *banay+at* example just above would require rules to constrain where **a:0**, **y:0** and **+:0** can and must occur. The *kaNpat* example will require two rules to constrain where **N:m** and **p:m** can and must occur.

Of course, for most real natural languages, there are many alternations between the lexical and surface strings, and you will have to write out and align many pairs of strings as part of your grammar planning. If you don't write out and line up your lexical and surface strings, identify your symbol pairs, and then write suitable **twolc** rules, in that order, you're doing it wrong.

There are often several reasonable ways to line up the strings, with hard zeros in different places, that lead to equivalent results. For example, the "`banay+at`" string might be aligned as

```
Lexical:  banay+at
Surface:  bana000t
```

However, any grammar of compatible rules will depend on the examples being lined up consistently. It is highly recommended that you document your rules with comments showing some sample pairs of lexical and surface strings to remind yourself how you originally decided to line up the symbol pairs. Changing the lineup conventions in the middle of development may require changes to multiple rules—remember that all the rules apply in parallel and must agree in how the two levels line up.

> Plan and document the lexical-surface lineup of all symbol pairs in your examples. This lineup must be motivated and consistent so that a coherent set of two-level rules can be written.

twolc comments are preceded by an exclamation mark and continue to the end of the line.

```
! twolc comments extend to the End of Line

! banay+at    comment alignments for future reference
! ban000at

! kaNpat
! kammat

! Comments are Good Things, use lots of them
```

> Use plenty of comments in your **twolc** files, including examples that show how the lexical and surface strings are lined up symbol by symbol.

twolc rules differ from replace rules both in their syntax and semantics, and this is a common source of confusion. The following points need to be emphasized:

1. Basic **twolc** rules constrain a single pair of symbols. The left side of a **twolc** rule consists of a single symbol pair **u:d**, with one single symbol **u** on the upper side and another single symbol **d** on the lower side. Multicharacter symbols are possible, and as in **xfst** they are declared implicitly by writing several

symbols together: e.g. **%+Noun:o**. As the plus sign is a special character in **twolc** rules, literalize it where necessary by preceding it with a percent sign (%).[10]

2. **twolc** contexts are always interpreted to be two-level. That is, in **twolc** rules, the contexts always refer to both the upper and lower sides of the relation. You can leave the lower side unspecified by leaving the right (lower) side of a pair empty, as in the `u:` notation. You can leave the upper side unspecified, as in the `:l` notation. A colon by itself, i.e. `:`, refers to any single symbol pair in the alphabet.

3. **twolc** grammars always work relative to an alphabet of symbol pairs. The alphabet for a **twolc** rule transducer is the collection of overtly and implicitly declared symbol pairs.

4. Within **twolc** rules, the `0` (hard zero) character is a real character and should be treated as such when aligning examples and writing rule contexts. The hard zeros of **twolc** are therefore different from the zeros in **xfst** and **lexc**, where `0` simply denotes an epsilon or empty string.

5. **twolc** rule arrows have their own semantics, as shown in Table 1. The left-arrow and right-arrow restrictions are not symmetrical, and both are quite different from the semantics of **xfst** replace rules.

2.4 Basic twolc Interface Commands

In order to attempt the first exercise below, you will need to know the basic commands in the **twolc** interface to read and compile your **twolc** source file. The **twolc** interface is invoked by entering **twolc** at the command line.

```
unix> twolc
```

[10] In **twolc** regular expressions, special characters cannot be literalized by surrounding them in double quotes, as in **xfst**.

twolc will respond with a welcome banner, a menu of commands, and a **twolc** prompt.

```
  ********************************************************
  *            Two-Level Compiler 3.2.1 (8.0.0)          *
  *                      created by                      *
  *             Lauri Karttunen, Todd Yampol,            *
  *         Kenneth R. Beesley, and Ronald M. Kaplan.    *
  *   Copyright (c) 1991-2002 by the Xerox Corporation.  *
  *                  All Rights Reserved.                *
  ********************************************************


Input/Output ------------------------------------------------
    Rules:        read-grammar.
    Transducers:  install-binary, save-binary, save-tabular.
    Lexicon:      install-lexicon, uninstall-lexicon.
Operations --------------------------------------------------
    Compilation:  compile, redo.
    Intersection: intersect.
    Testing:      lex-test, lex-test-file, pair-test,
                     pair-test-file.
    Switches:     closed-sigma, quit-on-fail, resolve, time,
                     trace.
Display -----------------------------------------------------
    Result:       labels, list-rules, show, show-rules.
    Misc:         banner, storage, switches.
    Help:         completion, help, history, ?.
Type 'quit' to exit.
twolc>
```

You can cause the menu of commands to be redisplayed at any time by entering a question mark. You will need to know just a few utilities from the **twolc** interface to get started.

- **read-grammar *filename*** reads in your source *filename* and checks for purely syntactic errors.
- **compile** causes a successfully read-in **twolc** source file to be compiled into finite-state transducers, one transducer for each rule. Compilation is therefore a two-step process of **read-grammar** followed by **compile**.
- **lex-test** allows you to test your compiled rules immediately by manually inputting lexical strings for generation.
- **save-binary** saves the compiled rule transducers to file. Each individual transducer is written into the file separately, they are not automatically combined into a single network by intersection.
- **save-tabular** is used to save compiled **twolc** rules in the tabular format required by **TwoL** and **PC-KIMMO**.
- **quit** exits from **twolc**.

2.5 Exercises

2.5.1 The kaNpat Exercise

The first exercise, just to get used to using basic **twolc** commands, is to redo the *kaNpat* example using **twolc** rules. The first step, as usual, is to write out the relevant examples as string pairs, with the lexical string on the upper side and the surface string on the lower side.

Lexical:	kaNpat	kampat	kammat
Surface:	kammat	kammat	kammat

That is, we want lexical "kaNpat" to map to surface "kammat", lexical "kampat" also to map to surface "kammat", and in addition lexical "kammat" will map to itself. A little study of these examples will show that in some precise environments a lexical **N** has to map to a surface **m**, and in other precise environments a lexical **p** must map to a surface **m**. In addition we want to allow **m:m** and **p:p** in other environments.

Type in the following **twolc** source file using a text editor and save it to file as kaNpat-twolc.txt.

```
!  This is a comment.  Use lots of them.

Alphabet m p ;

Rules

"N:m rule"
N:m  <=>  _  p:  ;

"p:m rule"
p:m  <=>  :m  _  ;
```

Enter **twolc** and use **read-grammar** and **compile** to compile the rules.

```
twolc> read-grammar kaNpat-twolc.txt
twolc> compile
```

If **read-grammar** reports syntax errors, re-edit the source file until it reads in cleanly. During compilation, ignore any warning messages about the left-arrow restriction of the N:m rule being redundant. Any genuine error message indicates a typing error that you should correct before proceeding. Test the grammar using **lex-test**, inputting the lexical strings "kaNpat", "kampat" and "kammat".

```
twolc> lex-test kaNpat
```

They should all have the same surface output "`kammat`". Input other strings like "`book`" and "`monkey`" that are not affected by the rules; they should all be mapped to themselves.

Edit the source file, reverse the order of the rules, recompile and retest. Such reordering has no effect in a **twolc** grammar because the rules are applied in parallel.

In this example, **m** and **p** are declared in the Alphabet section; this is equivalent to declaring **m:m** and **p:p**. These declarations are required because we "mention" the symbols **p** and **m** in the pairs **p:m** and **N:m** in the rules. Unless we go back and explicitly declare **m:m** and **p:p**, **twolc** will assume that **p** can appear only on the lexical side and that **m** can appear only on the surface side. We don't declare **N:N**, in this case, because **N** stands for an abstract underspecified morphophoneme that should never reach the surface as **N**. In a more complete grammar, **N:n** and **N:NG** (where **NG** is the name of a single symbol) may also be possible realizations in other environments, but **N** itself should never reach the surface level. Any UNKNOWN symbol that does not appear anywhere in the rules is treated as an identity pair. If you try the input "`hello`" in **lex-test** on your grammar, you will see that the output string is "`hello`". Because the symbols **h**, **e**, **l**, and **o** are not affected by any rule in the *kaNpat* grammar, the compiler does not impose any constraints on them.

Note that the N:m rule has a right context `p:` which matches all symbol pairs in the alphabet that have **p** on the upper side. Because the parallel p:m rule is relating this same upper-side **p** to a lower-side **m**, why is it important for the N:m rule to specify a right context of `p:` rather than `p:p` or `p`?

Similarly, the p:m rule has a left context `:m`, which matches all symbol pairs in the alphabet having **m** on the lower-side. Why is it important in this case to specify a left context of `:m` rather than `m:m` or `m`?

Try editing the source file to make the contexts more specific and see if the rules still produce the correct output. You will find in writing **twolc** rules that making contexts too specific is just as dangerous as not making them specific enough.

2.5.2 Brazilian Portuguese Pronunciation

The next exercise is to redo the Brazilian-Portuguese Pronunciation exercise using **twolc** rules. Refer to Beesley&Karttunen (2003), page 147, for the facts to be captured.

The first step, as always with **twolc** rules, is to line up many examples as lexical/surface string pairs. Match the lexical and surface symbols one-to-one as economically, consistently and beautifully as possible. Use hard zeros to pad out the lexical and surface strings so that each pair of strings has the same length. Use zeros consistently in parallel examples. Table 2 shows our recommended way to line up the Portuguese-pronunciation examples, but it is ultimately up to each linguist to define how the levels line up. Another way of saying this is that it is up to the linguist to decide where the zeros are. It is also up to the linguist to be consistent in placing the zeros.

The fully lined-up string pairs will identify the symbol alternations between the

me mi	disse Jis0i	tarde tarJi	partes parCis	verdade verdaJi
do du	tio Ciu	filho fiL0u	ninho niN0u	carro kaR0u
caro karu	camisa kamiza	vez ves	zebra zebra	casa kaza
peruca piruka	braço brasu	chato $0atu	chatinho $0aCiN0u	interesse interes0i
homem 0omem	rápido Rápidu	peru piru	braços brasus	hostil 0osCil
antes anCis	paredes pareJis	livros livrus	cada kada	cedilha seJiL0a
pedaço pedasu	parte parCi	parede pareJi	sabe sabi	simpático simpáCiku
filhos fiL0us	case kazi	ninhos niN0us	cantar kantar	bicho bi$0u
time Cimi	fortes forCis	usar uzar	dente denCi	tempo tempu
dia Jia	rato Ratu	cimento simentu	cases kazis	luz lus
cachorro ka$0oR0u	vermelho vermeL0u	diferentes JiferenCis	sonho soN0u	e i

Table 2: Pairs of Strings for the Portuguese Pronunciation Example. The **twolc** rules should accept the upper-side strings as input and produce the lower-side strings as output.

lexical and surface levels. Write the necessary **twolc** rules to constrain the relation between lexical and surface strings. Keep the semantics of **twolc** rules constantly in mind. Remember that you are writing rules that will apply in parallel, and remember also to treat zeros as real symbols inside **twolc** rules.

Here are a few hints:

- There should be only one rule to constrain each symbol pair such as d:J and r:R.
- A rule may have multiple contexts, each context terminated with a semicolon.
- The dollar sign (**$**) is a special character in **twolc**, as in other **Xerox** regular expressions. To use it as a literal character representing the phoneme /ʃ/, it must be made un-special by preceding it with the literalizing percent sign (**%**).
- Remember not to make your contexts overly specific. In several rules you will need to specify a context only on the upper side, e.g. `h:`. In others, you will need to specify only a lower-side context, e.g. `:i`.
- Once the rules have been successfully compiled, try testing a few examples manually using the command **lex-text**.
- For batch testing, type the upper-side strings into a file, one word to a line, and use the command **lex-test-file** to generate all of them. Edit and retest your rules until all the examples are generated correctly.

3 Full twolc Syntax

3.1 Header Sections

Each **twolc** source file must start with an Alphabet section and include a Rules section as described above. In addition, the following optional sections may appear between the Alphabet and Rules sections.

3.1.1 Sets

To aid in writing rules, you can optionally define sets of characters. The keyword `Sets` is followed by assignment entries:

```
Sets
    Vowels = a e i o u ;
    Conson = b c d f g h j k l m n p q r s t v w x y z ;
```

Each entry consists of a set name on the left side, an equal sign, and a list of symbols; and each entry is terminated with a semicolon. Because the semicolon indicates the end of each set, you can continue the listing over multiple lines.

Set definitions can include previous Set definitions.

```
Sets
    Vowels = a e i o u ;
    Conson = b c d f g h j k l m n p q r s t v w x y z ;
    ExtendedVowels = Vowels w y ;
```

These defined set names can then be used in rule contexts and in rule-scoped variables (to be presented below). A bug (or "feature") of sets is that all the symbols mentioned in a set must be listed explicitly in the Alphabet section.

3.1.2 Definitions

If multiple rules require the same left or right context (or a significant part of a context) it may be wise and convenient to define the context in the optional Definitions section. The syntax is similar to that in the Sets section, but the right side of each assignment is a **twolc** regular expression. Definitions can make use of previously declared definitions.

```
Definitions
   XContext = [ p | t | k | g:k ] ;
   YContext = [ m | n | n g ] ;
   ZContext = [ a | e | i | o | u ]* XContext ;
```

3.1.3 Diacritics

Diacritics, explained below, are a deprecated feature of **twolc**. Do not confuse **twolc** Diacritics with the very different Flag Diacritics explained in Beesley&Karttunen (2003), Chapter 7.

3.2 Full twolc Rule Syntax

3.2.1 Multiple Contexts

A single rule can have multiple contexts, e.g.

```
"Rule 1"
s:z <=> Vowel _ Vowel ;
        Vowel _ [ m | n ] ;
```

Each context must end with a semicolon. Left-arrow restrictions are interpreted conjunctively, imposing the constraint that a lexical **s** must be realized as **z** in all of the indicated contexts. Right-arrow restrictions are interpreted disjunctively, limiting the symbol pair, here **s:z**, to appearing in any of the indicated contexts (but in no other contexts).

3.2.2 Rule-Scoped Variables

A **twolc** rule may contain any number of local VARIABLES that range over a set of simple symbols. Variables and the assignment of values to them are specified in a `where` clause that follows the last context of the rule. A `where` clause consists of (1) the keyword `where` followed by (2) one or more variable declarations, and (3) an optional keyword (`matched` or `mixed`) that specifies the mode of value assignment. A variable declaration contains (1) a local-variable name chosen by the programmer (2) the keyword `in`, and (3) a range of values. The range may be either a defined set name or a list enclosed in parentheses containing symbols or set names, for example.

```
"foo rule"
Cx:Cy <=>     _  .#. ;
          where Cx in (b d c  g)
                Cy in (p t c2 k)
          matched ;
```

In this rule, the two local variables, Cx and Cy, have four possible values. The keyword `matched` means that the assignment of values to the two variables is done in tandem so that when Cx takes its nth value, Cy has its nth value as well. The compiler interprets this type of rule as an intersection of four independent subrules:

```
b:p  <=>  _ .#. ;
d:t  <=>  _ .#. ;
c:c2 <=>  _ .#. ;
g:k  <=>  _ .#. ;
```

If the keyword `matched` is not present at the end of a `where` clause, the assignment of values to variables is not coordinated. In this case, assigning values freely to the two variables would create 16 subrules with all possible pairings of values from the two sets. The keyword `mixed` indicates such a free assignment overtly.

Matched variables are also convenient when there is a dependency between the correspondence and context parts of the rule. A case of that sort is a rule that creates a geminate consonant at a morpheme boundary:

```
"gemination"
%+:Cx <=>  Cy  _  Vowel ;
           where Cy in (b c d f g k l m n p r s t v z)
                 Cx in (b k d f g k l m n p r s t v z)
           matched ;
```

For example, the rule would realize lexical forms such as "`big+er`", "`picnic+ed`" and "`yak+ing`" as the appropriate English surface forms "`bigger`", "`picnicked`" and "`yakking`".

Variables can also be used to encode dependencies between the left and right parts of the context. For example, if HighLabial is defined as

```
  HighLabial = u y ;
```

the following rule realizes **k** as **v** in two contexts. (Assume that ClosedOffset is a regular expression defined in the Definitions section.)

```
"Gradation of k between u/y" ! k weakens to v between
                             ! u's and y's
k:v <=>  Cons Vx  _  Vx  ClosedOffset ;
         where Vx in HighLabial ;
```

In this case no variable occurs in the correspondence part of the rule; the compiler only needs to expand the context part as follows:

```
k:v <=>  Cons  u  _  u  ClosedOffset ;
         Cons  y  _  y  ClosedOffset ;
```

Because the rule contains only one variable, the interpretation of the "Gradation of k between u/y" rule is the same regardless of whether the keyword `matched` is present or absent.

3.3 Full twolc Interface

3.3.1 Command-Line Flags

If you launch **twolc** from the command line with the **-h** flag, it prints a short usage message:

```
unix> twolc -h

usage: twolc [-h | -help | -q | -v]
```

The command **twolc -help** gives more verbose information:

```
TWOLC COMMAND-LINE OPTIONS:
-h       print a cryptic 'usage' message and exit.
-help    print this help message and exit.
-q       operate quietly.  Don't print unnecessary messages.
-v       print twolc version number and exit.
```

3.3.2 Utilities for Rule Testing and Intersection

In addition to the basic **twolc** operations described above, there are also the following utilities that may prove useful. Use **help** for more information.

install-binary

> **install-binary** is a command to read a binary file from disk for use in the **twolc** interface. A binary file is one that has already been compiled, typically by previous invocations of **read-grammar**, **compile** and then **save-binary**.

intersect

When **twolc** compiles a set of rules, each rule is stored as a transducer, and the transducers are kept separate by default. To force **twolc** to intersect the various transducers into a single transducer, invoke **intersect** after invoking **compile**.

```
twolc> read-grammar yourlang-twolc.txt
twolc> compile
twolc> intersect
twolc> save-binary yourlang-twolc.fst
```

Keeping the transducers separate by default is important for rule testing with **pair-test** (see below). If all the rule transducers were automatically intersected into a single transducer, then **pair-test** could not identify which of the component rules was responsible for blocking a mapping between two strings.

lex-test

The **lex-test** utility allows the linguist to test a set of compiled rules by manually typing in lexical strings for generation. **lex-test** outputs the surface form or forms dictated by the rules.

Note that **lex-test** applies the rule transducers in a downward direction, performing generation on the input strings. Testing bare rules in the opposite direction, i.e. analysis, is not generally possible because most practical rule sets produce an infinite number of analyses when those analyses are not constrained by a lexicon.

lex-test-file

lex-test-file is like **lex-test** except that you specify an input file (consisting of a list of strings, written one to a line, to be generated from) and an output file for the results. The interface prompts you for the filenames.

pair-test

pair-test is a useful utility that allows you to identify which rule in your rule set is blocking a desired mapping. Suppose that you have written a set of 50 rules, and that one of the desired and expected mappings is the following:

```
Lexical:  simpático
Surface:  simpáCiku
```

If, in fact, you input "`simpático`" to **lex-test** and there is no output, or if the output is not the desired "`simpáCiku`", then identifying the villain in a set of 50 rules may be anything but trivial. The solution is to invoke **pair-test**,

which will prompt you for the lexical string and the surface string that you think should be generated from it. Then **pair-test** will run the two strings, symbol pair by symbol pair, through the rule set and identify by which rule, and at which position in the string, the desired mapping is being blocked.

Note that for **pair-test** the lexical string and the surface string must contain exactly the same number of symbols, including hard zeros positioned in exactly the right places. It's at times like this, testing for blocked derivations, that you will be especially grateful for your own documentation showing how lexical and surface strings are supposed to line up.

pair-test-file

The **pair-test-file** utility is like **pair-test** except that it takes its input (pairs of strings) from a file you specify and outputs its result to another file that you specify. The input file should be formatted as follows, with a blank line between the pairs of strings:

```
simpático
simpáCiku

partes
parCis

vermelho
vermeL0u

ninho
niN0u

carro
kaR0u
```

If you maintain a file of valid lexical-surface pairs, i.e. mappings that the rules should perform, then re-running that file periodically through **pair-test-file** is a valuable form of regression testing.

install-lexicon

install-lexicon allows you to load a lexicon (or pseudo lexicon) into the **twolc** interface for filtering out conflict reports. When compiling two-level rules, **twolc** is extremely good (sometimes dismayingly good) at identifying and reporting possible conflicts between rules. But **twolc** assumes that *any* possible string, i.e. the universal language, could be applied for generation—if the conflicts occur only for input strings that cannot, in fact, occur in the natural language in question, then the rule writer may find the conflict hard to

understand, and fixing the rules may seem pointless or at least a profound nuisance.

Consider the following two rules.

```
"Rule 1"
x:y <=>    _  a ;

"Rule 2"
x:z <=> i  _    ;
```

During compilation, **twolc** will dutifully report a left-arrow conflict between the two rules, recognizing that when generating from a possible input string like "`ixa`", Rule 1 would insist on realizing the **x** as **y**, yielding "`iya`"; whereas the second rule would insist on realizing the same **x** as **z**, yielding "`iza`". This is a fatal left-arrow conflict, assuming that a string containing "`ixa`" might indeed be input for generation.

However, it may be the case in the language in question that the sequence "`ixa`" simply never occurs; and therefore the potential conflict between the two rules just doesn't matter. It is in such cases that **install-lexicon** can prove useful. Let us assume that the linguist has already built a lexicon transducer, perhaps using **lexc**, and that this transducer has been saved to file as `lex.fst`. Let us also assume that the lower-side language of `lex.fst` naturally contains no strings containing "`ixa`". After **install-lexicon** has been invoked to read this `lex.fst` into **twolc**'s lexicon buffer, **twolc** will subsequently consult the lower side of the lexicon to filter out rule-conflict messages that just don't matter. For the two rules above, the potential conflict will be ignored, and no conflict message will be reported, when **twolc** finds that the lower-side language of the lexicon includes no string containing the problematic "`ixa`" sequence.

The lexicon loaded using **install-lexicon** need not be a real lexicon, and it need not even be two-level. In the case above, the linguist could compile, using **xfst**, a simple regular expression

```
~$[ i x a ] ;
```

denoting the language of all strings that do not contain "`ixa`". Installing this pseudo-lexicon will be sufficient to suppress the left-arrow conflict report for the cited rules. **install-lexicon** can therefore use either a real lexicon or a pseudo-lexicon, and the pseudo-lexicon could of course be arbitrarily complex, tailored to filter out all kinds of error messages that are not relevant for the language being modeled.

uninstall-lexicon

The **uninstall-lexicon** utility removes and discards a lexicon previously installed using **install-lexicon**. Once uninstalled, the lexicon will no longer be used for filtering out conflict messages.

3.3.3 Miscellaneous twolc Utilities

In addition, **twolc** provides more utilities for timing, tracing, examining compiled rules, etc. Most of these utilities are more useful to **twolc** maintainers and debuggers than to the average user.

You can cause the full menu of **twolc** utilities to be displayed at any time by entering a question mark (?) at the prompt. Use the **help** utility to see some short documentation for each command.

3.4 Exercises

3.4.1 Monish Vowel Harmony

Review the facts of the mythical Monish language starting on page 162 in Beesley&Karttunen (2003). Then redo the exercise using **lexc** and **twolc**.

- Create a **lexc** file called `monish-lex.txt`, compile it using **lexc**, and save the result to file as `monish-lex.fst`.

- Create a **twolc** file for Monish vowel-harmony called monish-rul.txt, compile it using **twolc**, and save the result as `monish-rul.twol`. The trick is to write a grammar that realizes each underspecified vowel as front or back depending on the frontness or backness of the previous vowel.

- Compose the lexicon and rules using the **lexc** interface, and save the result to file as `monish.fst`, i.e.

```
lexc>  read-source monish-lex.fst
lexc>  read-rules monish-rul.twol
lexc>  compose-result
lexc>  save-result monish.fst
```

- Test the resulting system using **lookup** and **lookdown** in **lexc**. It should analyze and generate examples like the following.

yääqin+Perf+2P+Pl yääqinenémerä
feséén**g**+Opt+False+1P+Pl+Incl feséén**g**iddéqääbigä
bunoots+Int+Perf+2P+Sg bunootsuukonóma
tsarlók+Opt+False+1P+Sg tsarlókuddóqaaba
ntonól+Imperf+1P+Pl+Excl ntonólómbaabora

- Test for bad data as well. Your system should not be able to analyze the following ill-spelled Monish words.

```
yääqinenémorä    [contains 'o' in a front-harmony word]

tsarlókuddóqaabe [contains 'e' in a back-harmony word]
```

3.5 Understanding twolc and Networks

When you compile a two-level rule grammar with **twolc** and save the result as a binary file using **save-binary**, the transducers are saved into the file one-by-one. They are not intersected into a single network, unless you have explicitly already done so by invoking the **intersect** command.

A file that contains several compiled two-level rules can be loaded back into **twolc** and **lexc**, but in general it is not advisable to load such a file into **xfst**. In **xfst**, the multiple rule networks are pushed onto The Stack as usual; but any subsequent stack operation on them, such as composition, is not likely to produce the intended result. Intersection of two-level networks in **xfst** works correctly but prints a flurry of warning messages if the rules contain one-sided epsilon pairs because intersection is not in general applicable to such networks.

xfst is a general tool for computing with finite-state networks, and the transducers that represent compiled two-level rules have no special status within **xfst**. Let us recall, for example, that the **apply up** and **apply down** commands in **xfst** apply only the top network on The Stack to the input. If the stack contains several rule transducers that were intended to be applied in parallel, the result will not be as expected.

In **lexc**, you use the command **compile-source** or **read-source** to put a network in the **lexc** SOURCE register, and you use **read-rules** to read one or more rule networks, compiled previously by **twolc**, into the RULES register; when you then invoke **compose-result**, **lexc** uses a special INTERSECTING COMPOSITION[11] algorithm that composes the SOURCE with the RULES and puts the resulting single network into the RESULT register. For example, if you have a lexicon `my-lex.fst`

[11] As its name suggests, the intersecting composition algorithm performs the intersection and composition of the rules simultaneously.

previously compiled and saved in **lexc**, and if you have a file `my-rul.twol` that you compiled and saved in **twolc**, the following series of **lexc** commands will compose them and write out the single result to the file `result.fst`.

```
lexc> read-source my-lex.fst
lexc> read-rules  my-rul.twol
lexc> compose-result
lexc> save-result result.fst
```

Note here that **read-source** is used to read in an already compiled transducer from file. If all you have is the `my-lex.txt` source file, do the following instead to compile the **lexc** source file and put the lexicon network in the SOURCE register. Note that the rule file `my-rul.twol` must always be compiled separately by **twolc** and stored using **save-binary**; you cannot compile a source file of **twolc** rules from inside **lexc**.

```
lexc> compile-source my-lex.txt
lexc> read-rules  my-rul.twol
lexc> compose-result
lexc> save-result result.fst
```

There are good practical reasons for storing the results **twolc** by default as a set of separate rule networks. These rule transducers are not intersected, unless you specifically tell **twolc** to do so using **intersect**, for four reasons.

1. **twolc** allows you to **show** the transducer corresponding to an individual rule, which can be useful for debugging. The command **show** is followed by the name of a rule.

   ```
   twolc> show unique_rule_name
   ```

2. The **pair-test** utility applies the rules individually to a pair of strings and tells you which rule is preventing the mapping; it can't identify the individual rule blocking the mapping if all the rule transducers are intersected into a single transducer.

3. **twolc** also offers **save-tabular** output, which outputs the rule transducers in a state-table format suitable for use by **PC-KIMMO** and the **TwoL** implementation of Lingsoft.[12] Here again, separate rule transducers are needed for debugging, and an intersected rule transducer might grow too big for practical use on small machines.

[12] If you hand-compile your two-level rules for **PC-KIMMO** they must be typed in this same state-table format.

4. The intersection of large rule systems may be impossible or impractical because the size of the resulting network may be very large. In the worst case, the intersection of two networks, one with k states, other with n states, may produce a network with the product $k \times n$ states.

The recipe to compile and intersect the rules within **twolc** looks like this:

```
twolc> read-grammar yourlang-rul.txt
twolc> compile
twolc> intersect
twolc> save-binary yourlang-rul.fst
```

If the rule networks can be intersected into a single network, all further processing, such as the composition with an already compiled source lexicon, can be done with the generic **xfst** interface. The intersection should be done within **twolc** because it will be done more efficiently and without provoking the warning messages that **xfst** would generate.

The composition of an entire lexicon transducer with a full set of **twolc** rules can blow up in size, at least temporarily during the computation. The intersecting-composition algorithm in **lexc** was once the only way to handle such large compositions, and it may still be the only way in certain cases. In the meantime, the general composition algorithm, as found in **xfst**, has become more efficient and may be used directly in most or all cases in which the intersection of the rules is practical.

4 The Art and Craft of Writing twolc Grammars

4.1 Using Left-Arrow and Right-Arrow Rules

In most practical **twolc** rule systems, the vast majority of the rules are, and should be, double-arrow (<=>) rules. Attempts by beginners to use left-arrow (<=) and right-arrow (=>) rules are usually mistakes, revealing a poor grasp of the semantics of the rule operators (see Table 1, page 18).

One case in which the use of single-arrow rules is justified and necessary is when the realization of a lexical symbol, e.g. **e**, is obligatorily one thing, e.g. **i**, in context A, but either **e** or **i** in a different context B.

Modifying the Portuguese Pronunciation data, let us assume a dialect in which the pair **e:i** can appear at the end of a word, and in this context, a lexical **e** must be realized as **i**. That would normally be expressed with a double-arrow rule as in Rule 1.

```
"Rule 1"
e:i <=>  _ .#.
```

Let us further assume that the pair **e:i** is also possible in the context `.#. p _ r`, but that the realization of lexical **e** in this context is not obligatory: it could be realized as either **i** or as **e**. If the Alphabet supports only **e:e** and **e:i** as possible realization of **e**, then we would notate these phenomena with a right arrow (=>) as in Rule 2.

```
"Rule 2"
e:i => .#. p _ r ;
```

The problem here is that Rule 1 and Rule 2 are in right-arrow conflict; Rule 1 states that the pair **e:i** can occur only in the context `_ .#.`, and Rule 2 states that the same pair can occur only in the context `.#. p _ r`. The solution that avoids all conflicts is the following pair of rules, one using the left arrow, and one the right arrow.

```
"Rule 1'"
e:i <=  _ .#.

"Rule 2'"
e:i => .#. p _ r ;
           _ .#. ;
```

Rule 1' now indicates that if a lexical **e** appears at the end of a word, it must be realized as **i**, but this left-arrow rule places no constraints on where the pair **e:i** can appear. Rule 2' now indicates that the pair **e:i** can appear only in the two contexts indicated, without forcing lexical **e** to be realized as **i** in either of them. By combining the left-arrow and right-arrow constraints of the two rules, the desired behavior is achieved. The lexical input "`sabe`" will be generated as "`sabi`", and the lexical input "`peru`" will be generated as both "`peru`" and "`piru`".

Such alternations that are obligatory in one context but optional in another are fairly rare, at least in standard orthographies. The single-arrow rules might be more useful when modeling phonological alternations.

4.2 Multiples Levels of Two-Level Rules

Koskenniemi-style Two-Level Morphology and the general two-level OT approaches to phonology and morphology claim that it is not only possible but virtuous to have a single level of simultaneous rules that directly constrain lexical-surface mappings. Proponents insist that the intermediate languages suggested by sequential (cascaded) rules do not exist and have no place in the notation.

However, given that sets of two-level rules compile into transducers, there is no formal restriction against writing a system that includes multiple levels of two-level rules, or even mixtures of sets of two-level rules with sequential replace rules. The multi-level system outline shown in Figure 19, including two sets of two-level

rules, Set A and Set B, is not only possible but is typical of some commercial morphological analyzers that were built at **Xerox** in the early 1990s, before replace rules became available.

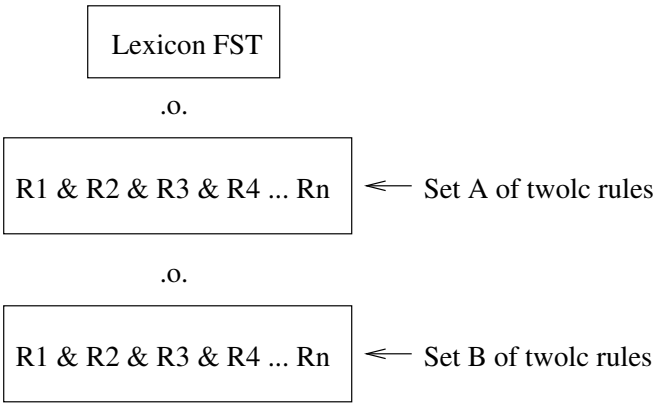

Figure 19: A System with Multiple Levels of **twolc** Rules. Here Set A of rules is compiled, intersected and then composed on the lower side of the lexicon transducer. Then Set B is compiled, intersected and composed on the bottom of the previous result. There might be any number of levels, even using two-level rule notation.

In such a system, the upper side of the Set A rules matches the lower side of the LexiconFst; and the lower-side of the Set A rules is not the final surface language but an intermediate language I. Similarly, the upper side of the Set B rules matches I, and the lower side is, finally, the real surface language. So in practice, there may be any number of levels and intermediate languages in a system, even when using the **twolc** rule notation. Of course, once the composition is performed, the result is a single transducer, and all the intermediate languages disappear. To computational linguists who are more impressed by formal power than by the superficial expression of the grammar, the prohibition against multiple levels seems more than a bit religious.

It is often convenient to compose filtering or modifying transducers on the upper side of a lexicon, and in the days before replace rules this was done with **twolc** rules as well. However, **twolc** rules assume a downward orientation, with the input on the upper side; so to write rules that map upward from the upper side of the lexicon transducer to a modified upper-side language, the trick is to write the rules (we'll call them Set C) upside down, compile them into a transducer, and then invert the transducer before composition. Again, there may be any number of transducers, compiled from **twolc** rules and inverted, to be composed on the upper side of the lexicon. Such inversions and compositions are easily performed in **xfst**, as shown in Figure 20.

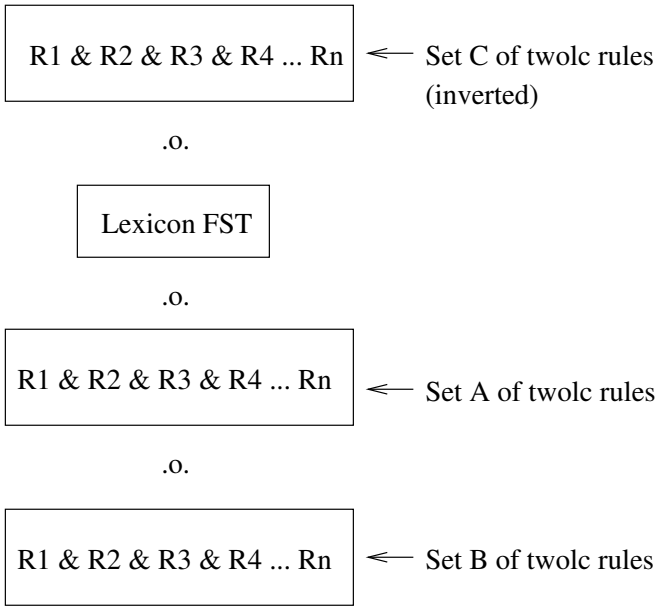

Figure 20: A System with Inverted **twolc** Rules Composed on the Upper Side of the Lexicon Transducer. There might be any number of levels of rules composed on the upper side, even using a two-level rule notation.

Using the **lexc** interface instead of **xfst**, the composition of multiple rule transducers on the lower side of a lexicon, and the composition of inverted transducers on the top of a lexicon, are a bit more difficult. The **lexc** interface has built into it the KIMMO-like assumption that there is but one lexicon, stored in the SOURCE register, and one set of rules, read into the RULES register; and the **lexc** command **compose-result** composes the rules on the lower side of the lexicon and stores the result in the RESULT register.

However, there are some **lexc** interface tricks that allow inversions and multiple compositions. Assuming that Sets A, B, and C of **twolc** rules have been precompiled and saved in files `A.fst`, `B.fst` and `C.fst`, and assuming that the lexicon is stored in `lexicon.fst`, the **lexc**-interface idiom for performing the

various inversions and compositions shown in Figure 20 is

```
lexc> read-source lexicon.fst

lexc> read-rules A.fst
lexc> compose-result
lexc> result-to-source

lexc> read-rules B.fst
lexc> compose-result
lexc> result-to-source

lexc> invert-source

lexc> read-rules C.fst
lexc> compose-result
lexc> invert-result
```

Notice here that to compose rules on the upper side of the lexicon, the lexicon is first inverted so that the upper side and lower side are switched. The **twolc** interface can then compose `C.fst` on the new lower side (old upper side) of the lexicon to create a result that is upside down. The result is then inverted to get the final transducer. This is obviously a bit awkward, and most **Xerox** developers now avoid the **lexc** interface, and very often **twolc** rules themselves, in favor of **xfst** and replace rules.

4.3 Moving Tags

Occasionally linguists feel a need to "move" a tag that was most naturally put in an undesirable position by a **lexc** grammar. Such moving is accomplished by introducing a new tag via an epenthesis rule in the desired new location, and then deleting the original tag, mapping it to the empty string.

Such "moving" of tags is not always recommendable, as it may result in the copying of large portions of your networks. The following little **lexc** grammar (`adj-lexc.txt`) illustrates the problem:

```
! adj-lexc.txt

Multichar_Symbols Neg+  +Adj

LEXICON Root
        Adjectives ;

LEXICON Adjectives
        NegPrefix ;
        AdjRoots ;
```

```
LEXICON NegPrefix
Neg+:un AdjRoots ;

LEXICON AdjRoots
healthy Adj ;
worthy  Adj ;
likely  Adj ;

LEXICON Adj
+Adj:0  # ;
```

In the **lexc** grammar, the *un-* prefix is naturally paired with the `Neg+` tag on the lexical side, yielding string pairs like the following (showing epsilons overtly as zeros).

```
Upper:  healthy+Adj             Neg+0healthy+Adj
Lower:  healthy0                u   nhealthy0
```

Now suppose that we really want to have lexical strings that always begin with a root, with all the tags concatenated after it. That is, in the negative case we might want something more like

```
Upper:  00healthy+Adj+Neg
Lower:  unhealthy0   0
```

with all the tags, including the `+Neg`, on the end. This will almost double the size of the transducer, because it creates a separated dependency (see Sections 4.4.2 and 7.3 in Beesley&Karttunen (2003)), but it makes an instructive exercise.

4.3.1 Tag-Moving Exercise 1

Take the **lexc** grammar as shown above, compile it using **lexc**, and store the binary result as `adj-lexc.fst`. Then "move" the tags by composing **xfst** replace rules on the top. One rule should map a single empty string [..] upwards to `+Neg` at the very end of the word, if the string is currently marked with a `Neg+`; and the other should map the original tag `Neg+` upwards to epsilon. In essence, the moving of tags will involve the insertion of one tag and the deletion of another. Do the composition on the **xfst** stack, or using regular expressions, arranging the order of the networks carefully.

4.3.2 Tag-Moving Exercise 2

Do the same using **twolc** rules and composing the lexicon with the rules inside **lexc**. This will be a bit tricky. **twolc** rules have a built-in downward orientation, and the usual assumption is that they will be composed on the *lower side* of a source lexicon

inside **lexc**. To modify the upper side of a transducer, as in this case, the trick is to write the **twolc** rules upside-down. Then, inside **lexc**, read the lexicon network and *invert* it before composing the **twolc** rules (which were written upside-down). Then *invert* the result. The **lexc** interface includes the necessary inversion commands.

```
lexc> read-source adj-lexc.fst
lexc> invert-source
lexc> read-rules my-rul.twol
lexc> compose-result
lexc> invert-result
```

4.3.3 Tag-Moving Exercise 3

Do the same exercise using **twolc** rules, but doing the composition on the **xfst** stack. Compile and intersect the rules in **twolc**, save the resulting network into a file, and load the file into **xfst**. Next, invert the rule network and compose it on upper side of the lexicon using **compose net** utility of **xfst**.

The problem with all of these solutions is that the size of the transducer jumps from 17 states, 21 arcs and 6 paths to 31 states, 35 arcs for the same 6 paths. That's a high price to pay for a cosmetic change in the spelling of the lexical strings.

5 Debugging twolc Rules

5.1 Rule Clashes

Because all the rules in a **twolc** grammar apply in parallel, simultaneously, there are abundant opportunities for rules to get in each other's way and conflict. Rule conflicts are discovered and reported by the rule compiler, and some of them are even resolved automatically, with appropriate warning messages. However, all rule conflicts are technically errors, and some of them cannot be resolved automatically, so every **twolc** user needs to understand rule conflicts and learn the idioms for resolving them. As a practical matter, always read the error and warning messages carefully.

There are two basic kinds of rule conflict:

1. Right-Arrow Conflicts

2. Left-Arrow Conflicts

The key to understanding rule conflicts is understanding the semantics of the **twolc** left and right arrows. Review the semantics on page 18 as necessary.

5.1.1 Right-Arrow Conflicts

It is possible for two rules to be in a right-arrow conflict. Right-arrow conflicts are usually benign. Here is a simple example.

```
"Rule 1"
a:b  <=>  l  _  r ;

"Rule 2"
a:b  <=>  x  _  y ;
```

From the right-arrow point of view, Rule 1 states that the pair **a:b** can occur only in the environment `l _ r`; and at the same time, Rule 2 states that the same pair **a:b** can occur only in a different environment `x _ y`. Wherever one rule succeeds, the other will fail; they are in mortal conflict.

Fortunately, such conflicts are resolved quite easily. If both contexts are in fact valid, then simply collapse the two rules into one rule with two contexts.

```
"Rule 3"
 a:b  <=>  l  _  r ;
           x  _  y ;
```

The **twolc** compiler will then constrain **a:b** to appear *either* in the first context `l _ r` *or* in the second context `x _ y`.

The rule writer may prefer to leave the resolution of right-arrow conflicts to the compiler itself. The **twolc** interface has a special switch that can be toggled off and on with the command **resolve**. The default position is ON. When the **resolve** switch is ON, the compiler looks for and tries to resolve rule conflicts. In the case of a right-arrow conflict, it assumes that the rule writer intends to allow the particular realization in all of the contexts that are mentioned in the rules. In the case at hand, the effect is that both Rule 1 and Rule 2 are in fact compiled as Rule 3, and the compiler prints the message:

```
>>> Resolving a => conflict with respect to 'a:b'
    between "Rule 1"
        and "Rule 2"
```

If the **resolve** flag is OFF and a right arrow conflict is not resolved manually, the effect is that lexical **a** cannot be realized as **b** in either of the two environments.

> Right-arrow conflicts occur when two right-arrow or double-arrow rules constrain the same feasible pair to occur in different environments. The conflict is usually resolvable by collapsing the two rules into one rule with two contexts. The compiler will do this automatically by default.

5.1.2 Left-Arrow Rule Conflicts

Left-arrow conflicts are more difficult to detect and resolve correctly. Consider the grammar in Figure 21. According to the semantics of **twolc** rule operators, the left-arrow constraint of Rule 4 states that if lexical **a** is between a left context that ends with **l** and a right context that begins with **r**, it must be realized as **b**. Simultaneously, the left-arrow constraint of Rule 5 states that a lexical **a** in this same environment must be realized as **c**. The two rules impose contradictory requirements: wherever Rule 4 matches and succeeds, Rule 5 will fail, and vice versa. Because of the conflict, a lexical **a** has no valid realization in the context `l _ r`.

> Two rules are in left-arrow conflict when they each impose a left-arrow restriction, have overlapping contexts, and try to realize the same lexical symbol in two different ways.

```
Alphabet a a:b a:c b c ;

Rules

"Rule 4"
 a:b <=>  l  _  r  ;

"Rule 5"
 a:c <=>  l  _  r  ;
```

Figure 21: Rule 4 and Rule 5 are in Left-Arrow Conflict

If the **resolve** switch is ON, the compiler looks for and tries to resolve left-arrow conflicts as best it can. It tries to determine which of the rules has a more specific context. In the case of Rules 4 and 5, there is no difference; the contexts are identical. In that case, the compiler will arbitrarily give precedence to whichever rule comes first. Here Rule 4 will prevail over Rule 5. The compiler reports:

```
>>> Resolving a <= conflict with respect to 'a:b' vs. 'a:c'
    between "Rule 4"
        and "Rule 5"
    by giving precedence to "Rule 4"
```

In effect, Rule 5 becomes completely irrelevant. In the context `l _ r`, the preferred Rule 4 requires **a** to be realized as **b**. Because the right-arrow restriction of Rule 5 does not allow the **c** realization in any other contexts, the result is that **a** can never

be realized as **c**. In a case such as this one, automatic conflict resolution is as good as no resolution at all. There is a logical error that has to be corrected by the rule writer. The only real benefit is that the conflict is noticed and reported.

```
Alphabet a a:b a:c b c ;

Rules
"Rule 6"
a:b <=>     _  r  ;

"Rule 7"
a:c <=>  l  _  r  ;
```

Figure 22: Rule 6 and Rule 7 are in Left-Arrow Conflict

A more interesting case of a left-arrow conflict is demonstrated by Rules 6 and 7 in Figure 22. Rule 6 and Rule 7 have the same right context. Because the left context of Rule 6 is empty, the rule requires that a lexical **a** be realized as **b** in front of a following **r** regardless of what precedes it on the left. Rule 7 is more specific. It requires **a** to be realized as **c** only when it is both preceded by **l** and followed by **r**. Note that every instance of the context `l _ r` is also an instance of the context `_ r`. Formally speaking, the context of Rule 7 is SUBSUMED by the more general context of Rule 6. Figure 23 illustrates the relationship between the contexts of the two conflicting rules.

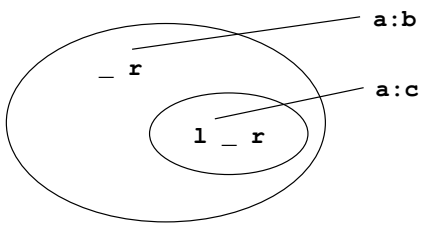

Figure 23: An Automatically Resolvable Left-Arrow Conflict Between a General and a More Specific Rule

In such cases, the compiler resolves the conflict by giving precedence to the more specific rule. It interprets the <= part of Rule 6 as if the rule writer had written it as `a:b | a:c <= _ r`. This does no harm because in a two-level grammar all the rules work together. Rule 6 can be relaxed to allow **a** to have either realization in its context because Rule 7 will prevent the **c** realization outside its own specific

context. The combined effect is that **a** is realized as **c** between **l** and **r** and as **b** in all other cases when **r** follows.

In a system of replace rules, there are no conflicts between rules, but the rule writer has to order the rules in a proper way. In particular, a specific rule always has to be ordered before a more general rule. In phonological literature (Chomsky and Halle, 1968) this is called DISJUNCTIVE ORDERING. The way in which the **twolc** compiler resolves left-arrow conflicts is motivated by this tradition.

There are, however, left-arrow conflicts that cannot be resolved in a principled way. When **twolc** discovers and reports an unresolvable left-arrow conflict, read the error message carefully, identify the rules involved, review the semantics of **twolc** left arrows if necessary, and make sure that you understand *why* the rules are in conflict. DO NOT start making changes until you understand the conflict completely. Beginners too often start changing arrows, fiddling with contexts, and flailing about aimlessly. Take the time to understand first.

After you understand what each rule is intended to do, and why they are in conflict, then one of them will need to be changed intelligently to resolve the conflict. If one of the rules is simply in gross error, then fixing it should be easy. Where both of the rules seem right, then the usual problem is that one needs to be made more specific in its contextual requirements. Consider this simple case:

```
"Rule 8"
a:b <=>   l  _     ;

"Rule 9"
a:c <=>      _  r  ;
```

Rule 8 and Rule 9 are in left-arrow conflict because both left contexts can match a string ending with **l**, and both right contexts can match a string beginning with **r**. The rules are in mortal conflict with respect to how strings such as "`lar`" should be realized, and the compiler cannot resolve the conflict for you. Very often in such cases, one of the rules is the general case and the other is a more specific exception. If so, the error can be fixed by simply making one of the rules more specific than the other. That is, the context of the more specific rule should be completely subsumed by the context of the more general rule instead of just partially overlapping with it. As we showed above, the compiler then resolves such conflicts automatically.

If this is truly a case of partial overlap between two rules, the rule writer has to decide which of the two rules should apply in the context where they are in conflict. The general idiom for fixing such left-arrow conflicts is called CONTEXT SUBTRACTION. In the case above, if we want to give Rule 8 precedence over Rule 9, with respect to strings such as "`lar`", we must subtract the left context of Rule 8, here any string ending in **l**, from the left context of Rule 9, which is the univeral relation. Recall that the rule compiler automatically extends each context with the universal relation over the feasible pairs of the alphabet, as shown in Figure 24. The new non-conflicting version, Rule 9' in Figure 25, has the same right context

```
"Rule 8"
a:b <=>   :* l  _  :*   ;

"Rule 9"
a:c <=>   :*    _  r  :*  ;
```

Figure 24: Automatic Context Extension. The **twolc** rule compiler automatically extends the left context to the left, and the right context to the right, with the universal relation over the alphabet of feasible pairs.

```
"Rule 8"
a:b <=>  l  _  ;

"Rule 9'"
a:c <=>  .#. [:* - [:* l]]   _  r  ;
```

Figure 25: Resolving a Left-Arrow Conflict, Giving Precedence to Rule 8, by Context Subtraction. Whereas Rule 9 matches anything for the left context, Rule 9' matches any left context, except (minus) left contexts that end with **l**. The subtracted left context, here **l**, might be an arbitrarily complex regular expression.

as the original rule, but the left context is modified by removing, i.e. subtracting, all strings that match the left context of the competing rule. Because the complement of language L, written ~L, is equivalent to `:* - L`, i.e. to the universal relation minus the strings in L, Rule 9' could be written equivalently as in Figure 26.

While context subtraction and the equivalent context complementation make perfect sense, they are notoriously difficult for students to grasp. Indeed, even experienced developers have had to reinvent the idiom several times. The key to understanding context subtraction is to remember that rules are compiled automatically with `:*` added on the left of the left context and `:*` on the right of the right context, as shown in Figure 24 and in Figure 18 on page 20. The full left context of Rule 8 is therefore `[:* l]` and the full left context of the original Rule 9 is `:*`, the Kleene-star relation on the pair alphabet. The expression [:* - [:* l]] in the left context of Rule 9' subtracts the full left context of Rule 8 from the full left context of the original Rule 9.

Note that the initial **a** in strings such as "ar" should be mapped to **c** by these rules; Rule 9' must be written to allow the possibility of an empty-string left context. The subtraction would not have the intended effect without indicating the ex-

```
"Rule 8"
a:b <=>  l  _  ;

"Rule 9'"
a:c <=>  .#. ~[:* l]  _  r  ;
```

Figure 26: Resolving a Left-Arrow Conflict, Giving Precedence to Rule 8, by Context Complementation. Whereas Rule 9 matched anything for the left context, Rule 9' matches any left context that does not end in **l**.

plicit word-boundary marker `.#.` in the left context of 9' because the relation `[:* - [:* l]]` includes the empty string. Of course, the equivalent `~[:* l]` also includes the empty string. As we have seen, the compiler always concatenates the relation `:*` to the left side of whatever the rule writer has specified. The concatenation of a Kleene-star relation to anything that contains the empty string is equivalent to the Kleene-star relation itself.[13] That is, `:* [:* - [:* - l]]` and `:* ~[:* l]` denote the same relation as `:*`. Without the initial word boundary `.#.` explicitly specified, the left context of Rule 9' would be exactly equivalent to the left context of the original rule. The intent of Rule 9' is to restrict the left context to "all sequences of symbol pairs that do not end with `l`". Because this includes the empty context, there is no way to formally express the idea without the boundary symbol.

The boundary symbol `.#.` is necessary in all cases where a specific context (some subset of `:*`) includes the empty string. Without the initial `.#.`, a rule such as `a:c <=> .#. :* - [:* l] _` or `a:c <=> .#. ~[:* l] _` is vacuous. Without the final `.#.`, the optional right context in a rule like `a:b <=> _ (r :*) .#.` has no effect.

In the example at hand, where the original context consists of just one symbol, it might seem that there is a simpler solution using \, the symbol-complement operator. If, as in Rule 9 Not-Quite-Right, we change the left context of Rule 9 to be `\l`, that is any single symbol or symbol pair except for **l**, then the conflict is resolved favoring Rule 8.

```
"Rule 8"
a:b <=>  l  _  ;
```

[13] See the discussion of the Restriction operator in Section 2.4.1 in Beesley&Karttunen (2003).

```
"Rule 9 Not-Quite-Right"
a:c <=>  \l  _  r  ;
```

However, whereas the original Rule 9 maps the initial **a** in strings such as "ar" to **c**, the Not-Quite-Right modified version does not; it requires at least one symbol of left context. To get the correct result, we must specifically handle the case of the empty context as shown below.

```
"Rule 9''"
a:c <=>  [.#. | \l]  _  r  ;
```

All the versions of Rule 9' and 9" are equivalent. They all require **a** to be realized as **c** at the beginning of a string or after any symbol or symbol pair other than **l**. Note that the backslash idiom, even with the left word-boundary specified, cannot resolve clashes where the left context is over one symbol long. Context subtraction, or context complementation, is the general idiom for such conflict resolution.

Let's look now at the opposite case: If we want to resolve the conflict between Rule 8 and Rule 9 in favor of the latter one, we must subtract the right context of Rule 9 from the right context of Rule 8 as shown in Figure 27. The new right context `[:* - [r :*]]` denotes any string, except those beginning with **r**. The trailing `.#.` in the modified Rule 8' is necessary for the reasons discussed above.

```
"Rule 8'"
a:b <=>  l  _  [:* - [r :*]] .#. ;

"Rule 9"
a:c <=>     _  r  ;
```

Figure 27: Resolving a Left-Arrow Conflict, Giving Precedence to Rule 9, by Context Subtraction of the Right Context. The right context of Rule 8' now matches all possible strings, including the empty string, except for those strings that begin with **r**.

An equivalent result is obtained via context complementation, shown in Rule 8" in Figure 28.

The context-subtraction idiom also works where the overlapping contexts are more complex, as in Figure 29. Let us assume that the intent of the rule writer is that the more general Rule 11 is to apply whenever the left context ends in a consonant, except when the left context ends with the more specific context `[ l e f t ]`. That is, Rule 10 expresses an exception to the more general rule. The context subtraction idiom works perfectly in this case too, but it is not necessary to use it because the compiler will automatically resolve the conflict in the desired way. That

```
"Rule 8''"
a:b <=>  l  _  ~[r :*] .#. ;

"Rule 9"
a:c <=>     _  r  ;
```

Figure 28: Resolving a Left-Arrow Conflict, Giving Precedence to Rule 9, by Context Complementation of the Right Context. The right context shown here as **r** could be an arbitrarily complex regular expression. The right word-boundary .#. must be explicitly indicated as shown.

is, it automatically relaxes the left-arrow restriction of Rule 10 to allow a lexical **a** to be realized as either **b** or **c**.

```
Alphabet a b c d f g h j k l
           m n p q r s t v y;

Sets
   Conson = b c d f g h j k l
            m n p q r s t v y ;
Rules

"Rule 10"
a:b  <=>  l e f t  _  r  ;

"Rule 11"
a:c  <=>  Conson  _  r  ;
```

Figure 29: Complex Overlapping Contexts

If the rule writer opts for the manual solution by context subtraction, the left context of Rule 11 should be modified as shown below in Rule 11'.

```
"Rule 11'"
 a:c  <=>  .#. [[ :* Conson] - [:* l e f t ]] _  r  ;
```

The automatic resolution of the conflict will produce a transducer for Rule 11 that is different from the transducer compiled from the manually modified Rule 11'. However, the intersection of the "relaxed" version of Rule 11 with Rule 10 is identical to

the transducer resulting from the intersection of Rule 10 and Rule 11'. That is, considering the rule system as a whole, the automatic resolution of left-arrow conflicts and the manual method of context subtraction produce exactly the same result.

5.1.3 Spurious Compiler Warnings

Although the compiler correctly detects most left-arrow conflicts, it is not perfect. In some cases the compiler issues spurious warnings. A simple example is shown in Figure 30. The two rules in Figure 30 allow a lexical **a** to be realized as either **b** or **c** but require that whichever way is chosen in a particular case, it must be chosen consistently. That is, "`aa`" can be realized as "`bb`" or as "`cc`" but not as "`ac`" or "`ca`".

```
Alphabet a:b a:c;

Rules

"Rule 12"
a:b  <=>  $[a:b] _ ;

"Rule 13"
a:c  <=>  $[a:c]  _ ;
```

Figure 30: **twolc** is Not Perfect. In spite of warnings from the compiler, there is no real <= conflict between these two rules.

The compiler compiles the rules correctly but gives a spurious warning about a left-arrow conflict.

```
*** Warning: Unresolved <= conflict with respect to
   'a:b' vs. 'a:c'
  between "Rule 12"
      and "Rule 13"

**** Conflicting pair(s) of contexts:
$a:b _  ;
$a:c _  ;
Left context example:     a:c a:b
```

The reason is that the compiler cannot "see ahead" what effects the rule or rules it is working on are going to have at the end. At the point where the compiler is comparing the context where **a** must be realized as **b** and the context where **a** must be realized as **c**, it sees a potential conflict in the case where a string containing two

as has been seen and one of the **a**s has been realized as **b** and the other as **c**. At the point where the compiler is comparing the two rules, it cannot yet deduce that the combined effect of the two rules is to disallow the problematic case.

In real life, examples of this sort are very rare. Because the message includes an example of the problematic context, the rule writer can decide whether to take the warning seriously or to ignore it. In our experience, if the compiler warns about an unresolved left-arrow conflict, it is nearly always right.

5.1.4 Why Context Subtraction Works

The context-subtraction and context-complementation idioms perform subtraction and complementation on relations, which is not generally possible. As pointed out above on page 20, the equal-length relations denoted by **twolc** rules are a special case.

From another point of view, the transducers of **twolc** and two-level morphology in general are reduced to simple **FSM**s denoting languages that consist of strings that in turn consist of letters like a:a, b:b, y:0, 0:i, etc. The basic alphabet always consists of such feasible pairs. The zeros in such pairs are the "hard zeros" of **twolc**, as real as **b** and **c**, and not equal (inside rules) to the empty string. The transducers of **twolc** and two-level morphology are therefore really one-level networks that are cleverly interpreted as if they were transducers.

5.2 Epenthesis Rules

Epenthesis or insertion rules are a challenge for any grammar, and they are surprisingly hard to get right in a finite-state system (where your rules are compiled and performed literally by a computer). Consider the following bad **twolc** rule:

```
Alphabet  0:h  %+Neg:0 ;

Rules

! this is a bad epenthesis rule

"Lenition"
0:h  <=>  .#.  [ b | t | s ] _   :*  %+Neg: ;
```

The intent of this rule is to perform the following mapping:

```
Lexical:  b0riseann+Neg
Surface:  bhriseann0
```

where "`briseann+Neg`" is a string from the lexicon. As usual, **twolc** rules will match and impose their constraints everywhere they can. The problem is that the colon (:) matches any feasible pair in the alphabet, including 0:h, so the left-arrow

portion of the rule continues to match and tries to insert an infinite number of 0:h pairs.

```
Lexical:  b0000...0000riseann+Neg
Surface:  bhhhh...hhhhriseann0
```

However, the right-arrow restriction of the same rule prohibits any 0:h insertions that are not immediately after an initial consonant, and so the rule is in conflict with itself.

One way to fix this particular rule is to specify something more specific than just :* as the start of the right context, e.g.

```
"Lenition"
 0:h  <=>  .#.  [ b | t | s ] _   \0:h*  %+Neg: ;
```

This new rule specifies that 0:h is inserted after a **b**, **t** or **s** at the beginning of a word, where the right context consists of any number of symbols that *are not* **0:h** and then a lexical `+Neg`. This prevents the infinite insertion and allows the intended insertion of a single **0:h**.

A similar problem occurs whenever a left or right context is left out entirely, e.g.

```
"bad epenthesis rule"
 0:i  <=>  c k _  ;
```

This rule is written so that any right context will match, so if the intent is to do the following

```
Upper: rack0
Lower: racki
```

0:i is also a valid right context for the rule, and it will match and require another insertion.

```
Upper: rack00
Lower: rackii
```

And in fact it will match repeatedly and require an infinite number of insertions, but once again the rule will be in conflict with itself.

```
Upper: rack000...00000
Lower: rackiii...iiiii
```

The way to fix the rule is the specify a right context that blocks the infinite insertion.

```
"good epenthesis rule"
 0:i  <=>  c k _  [ \0:i | .#. ] ;
```

A similar problem is found in **xfst** replace rules (see Beesley&Karttunen (2003), Section 3.5.5, page 175), where the bad epenthesis rule

```
! this is a bad xfst epenthesis rule

[] -> i  ||  l  _  r  ;
```

would try to insert an infinite number of **i**s in between **l** and **r** in the following example:

```
Lexical:  l000...0000r
Surface:  liii...iiiir
```

An infinite number of empty strings exist between **l** and **r** on the lexical side, and the rule tries to map them all into **i**.

For replace rules, the solution is provided in the [..] or "dotted-bracket" operator, which constrains strings to be interpreted as having exactly one empty string between each symbol. The following replace rule has exactly one output.

```
[..] -> i || l _ r

Lexical:  l0r
Surface:  lir
```

5.3 Diacritics

In a **twolc** file, an optional section allows you to declare diacritical characters, e.g.

```
Diacritics    %^FOO  %^DOUBLE  %^DEL   ;
```

Of all the features of **twolc**, the Diacritics have always been the worst documented and the most poorly understood. They also complicate the methods described above to resolve rule clashes. Diacritics are now a deprecated feature of **twolc**, but they are described here for completeness.

> **twolc** Diacritics should not be confused with the Flag Diacritics described in Beesley&Karttunen (2003), Chapter 7.

5.3.1 The Raw Facts about Diacritics

When a **twolc** grammar has Diacritics declared, the rules in the grammar are compiled in such a way that

- The Diacritics are always mapped to epsilon (zero) on the lower side, and
- The rules, by default, simply ignore the presence of the Diacritics in the lexical strings

However, if a rule explicitly "mentions" or refers to a Diacritic in its context, then that rule is compiled in such a way that it "notices" that Diacritic symbol.

The notion of Diacritics applies only in **twolc** files, and a declaration of Diacritics only affects the way in which the rules of the file are compiled into transducers. Inside the grammar as a whole, the Diacritic symbols are just symbols, typically multicharacter symbols, like any other.

5.3.2 The Motivation for Diacritics

To understand **twolc** diacritics, recall that the classical approach to finite-state morphology is to build a lexicon, typically using the **lexc** language, and to write a separate grammar of rules, typically using **twolc**, to map the abstract strings from the lexicon into surface strings. The rules are traditionally composed on the lower side of the lexicon such that the lower-side language of the lexicon is the upper-side language of the **twolc** rules.

In a lexicon, there is often a need to inject a feature symbol into certain strings which eventually, when the **twolc** rules are applied, either triggers the firing of a particular rule or blocks the firing of a particular rule. Such a symbol automatically becomes part of the alphabet of the lexicon, and normally all **twolc** rules would have to be written to use or at least allow the presence of such feature symbols in the strings.

However, writing all the rules to be aware of all the features is often not desirable, especially if only a couple of rules need to be aware of the feature symbols. The problem is especially acute in a relatively mature lexicon where the linguist feels a need to inject some new feature characters to capture behaviors that were not anticipated at the beginning. The injection of new feature symbols in certain strings may require the review and rewriting of all the rules in the grammar to allow the presence of the new symbols.

The motivation behind Diacritics was to let linguists inject feature symbols freely into their lexical strings (typically in a **lexc** program), but to allow them, in **twolc**, to declare such characters as Diacritics so that they would be ignored by all rules except those rules that overtly refer to them.

5.3.3 Noticing and Ignoring Diacritics

twolc rules are compiled, by default, in such a way that they ignore the symbols declared to be Diacritics. There are in fact two ways to override the default:

- Any rule that specifically mentions a Diacritic symbol in its context will be compiled in such a way that it notices (i.e. does not ignore) that symbol.
- Another overt way to force a particular rule to notice or pay attention to a Diacritic is to include an Attention statement immediately after the rule name.

The following example shows the use of the Attention statement, where the word *Attention* is followed by an equal sign and a list of Diacritic symbols (one or more) to be noticed.

```
Diacritics  %^MF  %^SG  %^PL  ;

Rules

"my rule"
Attention = ( %^MF )

0:x <=> l  _  r  ;
```

Such a rule would normally be compiled in such a way that it ignores completely the symbol `^MF`, declared in the same file to be a Diacritic; but with the overt Attention statement, the default is overridden and the rule will be compiled so that it treats `^MF` like any other symbol.

5.3.4 Diacritics and Rule Conflicts

Although the automatic resolution of left-arrow conflicts usually works correctly even when diacritics are involved, they present difficult problems if the conflict has to be resolved by the brute-force context-subtraction method. This is one of the major reasons why Diacritics are currently deprecated.

Most **twolc** writers eventually need context subtraction (see Section 5.1.2, page 48) to clean up rule conflicts in their grammars. If you feel a strong need to use context subtraction and Diacritics at the same time, you can often get away with it by overtly forcing the rules in conflict to notice all the declared Diacritics, as in the following example.

```
Alphabet A:0 ;   ! a token alphabet declaration

Diacritics %^FOO ;  ! rules will ignore ^FOO by default

Rules
```

```
"Rule 1"
Attention = ( %^FOO )
a:e <=>  .#. ~[:* y]  _  z  ;  ! with Diacritics declared,
                               ! and the rules forced to
"Rule 2"                       ! pay attention to them,
Attention = ( %^FOO )          ! the rules are no longer
a:o <=>  y _  z  ;             ! in conflict.
```

5.4 Classic twolc Errors

When writing **twolc** grammars, beginners often fall victim to some classic errors. The following points are worth reviewing.

1. You should always start a **twolc** analysis by writing out numerous two-level examples with the lexical string above the surface string, and with **twolc** hard zeros inserted at appropriate places to pad out the strings to the same length. The failure to "line up your strings", to document the lineup, and to be consistent in the lining up of your examples is a fundamental methodological error. Two-level rules constrain the appearance of feasible pairs in two-level strings, and you cannot write a workable set of rules unless you know what you're trying to accomplish.

2. **twolc** rules map strings from an upper level to a lower level, and the upper level of the rules is typically the lower level of the lexicon. Sometimes, linguists write multiple levels of **twolc** rules, mapping through a cascade of levels. The failure to plan and understand your own levels is a fundamental methodological error. You should think of each level of intermediate strings as a language and be able to characterize what the strings look like at each level.

3. The **twolc** compiler prints out error messages and warnings when it compiles a source file. Beginners tend to ignore the messages; experts study them carefully, fixing syntactic errors and rewriting rules to resolve conflicts that are not resolved automatically by the compiler.

4. **twolc** rules are deontic statements of fact that constrain where particular feasible pairs can appear in a pair of lexical/surface strings. A simple rule states constraints on a single feasible pair like **x:y**, while a compound rule with `where` clauses is just an abbreviation for multiple single rules that constrain a single feasible pair.

   ```
   "unique rule name"
   r:l  <=>  i  _  .#.  ;

   ! A typical rule constrains where a feasible pair
   ```

```
! like r:l can appear in a valid pair of lexical-
! surface strings
```

Attempts to make **twolc** rules map strings of symbols into strings of symbols, which is possible in replace rules, is a common error that betrays a poor grasp of the syntax and semantics of **twolc** rules. The following **twolc** rule, which might appear to map `+Participle` into the symbols **i**, **n** and **g**, will in fact implicitly declare a single multicharacter symbol named **ing** and constrain where the feasible pair **%+Participle:ing** can appear.

```
! declaring multicharacter symbols like ing is
! almost always an error

"bad ing rule"
%+Participle:ing  <=>   l  _  r  ;
```

In **twolc**, multiple letters written together, such as `%+Noun` or `%+Pl` or **ing**, are automatically interpreted, and implicitly declared, as single symbols with multicharacter print names. Where these names contain special characters like the plus sign or the circumflex, the special characters must be literalized with the preceding percent sign.[14] When the intent is to represent a genuine string of concatenated alphabetical characters such as **i** concatenated to **n** concatenated to **g**, it is almost always a mistake to use a single multicharacter symbol such as **ing** in **twolc**.

5. **twolc** always works within a fixed alphabet of feasible pairs, and in the earliest versions of the language the linguist was forced to declare all the feasible pairs overtly in the Alphabet section. This was tedious and led to mysterious errors when the linguist forgot to declare simple identity pairs such as **t:t** or **u:u**.

   In an attempt to remedy this problem, the current versions of **twolc** automatically assume that identity pairs such as **a:a**, **b:b**, **t:t**, etc. are in the alphabet, *unless* the overtly "mentioned" symbols suggest otherwise. For example, if the user overtly declares the pair **a:e** or writes a rule to control **a:e**, or even writes a rule where **a:e** appears in the context, then the compiler will assume that **a** can appear only on the lexical side and that **e** can appear only on the surface side. That is, "mentioning" **a:e** will override the default assumption that **a:a** and **e:e** are in the alphabet; if the linguist wants to retain **a:a** and **e:e** after declaring **a:e**, then they will have to be declared overtly in the Alphabet section.

[14] Note that special characters in **twolc** cannot be literalized by surrounding them in double quotes, which is possible in **xfst** regular expressions.

It is an open question whether the new alphabetic assumptions are any better than the old ones. In any case, the failure to understand what the alphabet is leads to many mysterious errors for beginners. Be aware that alphabetic symbols can be declared overtly, in the Alphabet section, or implicitly via their use in rules.

6. Often a linguist will declare feasible pairs like **%+Pl:s** and **s:s**, intending to limit `%+Pl` only to the upper-side rule language. Then, unwittingly, the linguist will write a rule like the following,

```
0:e  <=>  z:  _  %+Pl  ;
```

intending to insert a surface **e** into plurals like the Spanish "`vez+Pl:veces`". However, the right context of the rule, written simply as `%+Pl`, is automatically interpreted as `%+Pl:%+Pl`, and this feasible pair is then added automatically to the alphabet. What the linguist should have written is one of the following:

```
0:e  <=>  z:  _  %+Pl:  ;

! or

0:e  <=>  z:  _  %+Pl:s  ;
```

This common error does not show up in error messages because the compiler has no way of knowing whether you really mean to declare `%+Pl:%+Pl` or not. The typical sign of this problem is when you generate, via **lex-test**, a string like "`vez+Pl`" and you get multiple output strings like "`vece+Pl`", "`vezs`" and "`vez+Pl`" instead of the single solution "`veces`" that was expected. Experts learn to expect and check for problems in the **twolc** alphabet.

7. Finally, the little `where` clauses that define rule-scoped variables cause their share of problems. Linguists often write a rule like the following

```
"rule with a where clause"
XX:YY <=>  l  _  r  ;
           where XX in ( i e )
                 YY in ( u o ) matched;
```

and then try to add a new context on the end of the rule, e.g.

```
! a where clause must appear after
! all the contexts for a rule
```

```
"rule with a where clause"
XX:YY <=>  l  _  r  ;
            where XX in ( i e )
                  YY in ( u o ) matched;
           m  _  n  ;    ! ERROR, ERROR
```

However, if a `where` clause appears, it must be after the very last context. Unfortunately, the **twolc** rule compiler does not give an error message but simply stops parsing at the end of the `where` clause and ignores the misplaced new context and the remainder of the file. Once again, it is important to look carefully at the messages returned by the compiler, including the listing of the rules that were successfully compiled.

6 Final Reflections on Two-Level Rules

6.1 Upward-Oriented Two-Level Rules

It will be noticed (see Table 1, page 18), that the right-arrow and left-arrow restrictions of two-level rules are not symmetrical. The right-arrow => restriction, as in

```
a:b => l _ r ;
```

states that the symbol *pair* **a:b** is restricted to appear in the indicated context; the rule will therefore block the appearance of the pair **a:b** in any other context. Notice that there is no directionality in the right-arrow restriction—it is the pair **a:b** that is restricted to appear in a certain context or contexts.

In contrast, the left-arrow <= restriction has a clear downward orientation.

```
a:b <= l _ r ;
```

It states that if the lexical symbol **a** occurs in the indicated context, then this **a** must be realized on the surface as **b**. Any other realization of lexical **a** in the indicated context or contexts is blocked by the rule. Significantly, the rule does *not* say that a surface **b** in the indicated context must be a realization of lexical **a**; the **twolc** left-arrow restriction is downward oriented, from lexical to surface.

The double arrow <=> straightforwardly combines the non-directional right-arrow constraints and the downward-oriented left-arrow constraints.

The downward-orientation of the <= restriction suggests that one might define a new kind of two-level rule with an upward-oriented semantics. Let the left arrow with a single bar <- be a new operator such that the rule

```
a:b <-  l  _  r
```

states that if the surface symbol **b** appears in the indicated context, then it must be mapped to **a** on the lexical side; the rule would therefore block any other lexical source for surface **b** in the indicated context.

Such upward-oriented rules can certainly be compiled by hand and included in a set of two-level rules, but to our knowledge such upward-oriented two-level rules have never been supported by automatic rule compilers or seriously investigated for their usefulness.

6.2 Comparing Sequential and Parallel Approaches

The application of a set of replace rules to an input string involves a cascade of transductions, that is, a sequence of compositions that yields a relation mapping the input string to one or more surface realizations. The application of a set of **twolc** rules involves a combination of intersection and composition.

A set of rules of either type can in principle always be combined into an equivalent single transducer. Replace rules are merged by composition, **twolc** rules by intersection. The final outcome can be the same, as the following example shows.

Yokuts vowels (Kisseberth, 1969) are subject to three types of alternations. (The period, **.**, is used here to indicate that the preceding vowel is long.)

- Underspecified suffix vowels are rounded in the presence of a round stem vowel of the same height: *dub+hIn* → *dubhun*, *bok'+Al* → *bok'ol*.
- Long high vowels are lowered: *?u.t+It* → *?o.tut*, *mi.k+It* → *me.kit*.
- Vowels are shortened in closed syllables: *sa.p* → *sap*, *go.b+hIn* → *gobhin*.

Because of examples such as *?u.t+hIn* → *?othun*, the rules must be applied in the order shown. Rounding must precede lowering because the suffix vowel in *?u.t+hIn* emerges as *u*. Shortening must follow lowering because the stem vowel in *?u.t+hIn* would otherwise remain high giving *?uthun* rather than *?othun* as the final output.

Kisseberth's analysis of this data can be formalized straightforwardly as regular replace expressions in **xfst**.

```
define Cons [b|d|f|g|h|k|k'|l|m|n|p|q|r|s|t|v|w|%?];
define Rounding [I -> u || u  $%+  _ ,,
                 A -> o || o  $%+  _ ];
define Lowering [i -> e, u -> o || _ %. ];
define Shortening [%. -> 0 || _ Cons [Cons | .#.]] ;
define Defaults  A -> a, I -> i, %+ -> 0;
```

The composition `[Rounding .o. Lowering .o. Shortening .o. Defaults]` yields a 63-state single transducer that maps Yakut lexical forms to surface forms, and vice versa.

The same data can also be analyzed in **twolc** terms. The **twolc** Shortening and Lowering rules are simple notational variants of their **xfst** counterparts. The **twolc**

Rounding rule, however, is different from its **xfst** counterpart in one crucial respect: the left context of the alternation `[Vz: $%+: ]` requires a rounded lexical vowel without specifying its surface realization. Thus the lowering of the stem vowel in *?u.t+hIn* does not prevent it from triggering the rounding of the suffix vowel. This is one of the many cases in which *under-specification* in two-level rules does the same work as rule ordering in the old Chomsky-Halle phonological model.

```
Alphabet
 %+:0 %.:0 %. i i:e u u:o o e i I:u I:i A:a A:o a
 b d f g h k k' l m n p q r s t v w %? ;
Sets
  Cons =  b d f g h k k' l m n p q r s t v w %? ;
Rules
  "Rounding"
     Vx:Vy <=> Vz: $%+: _ ; where Vx in (I A)
                                  Vy in (u o)
                                  Vz in (u o)
                            matched;
   "Lowering"
      i:e | u:o <=> _ %.: ;
   "Shortening"
      %.:0 <=> _ Cons [Cons | .#.];
```

The intersection of the **twolc** rules for Yakut also yields a 63-state transducer that is virtually identical to its **xfst** counterpart. The only difference is that the two rule systems make a different prediction about the realization of hypothetical Yakut forms such as *?u.t+hI.n* in which the underspecified suffix vowel is long. In this case, the **xfst** Yakut rules predict that it would undergo all three processes yielding *?othon* as the final output whereas the **twolc** Yakut rules produce *?othun* in this case. With a little tinkering, it would be easy to make two rule systems completely equivalent, if one knew what the right outcome would be in that unattested case.

From a mathematical and computational point of view, **twolc** and **xfst** rule systems are equivalent in the sense that they describe regular relations. The sequential replace rules and the parallel **twolc** rules decompose, in a different way, a complex relation between lexical and surface forms into smaller subrelations that we can understand and manipulate. They represent different intuitions of what the best way is to accomplish the task. In practice, the two approaches are often combined, for example, by composing a set of intersected **twolc** rules with networks that represent constraints, replace rules, or another set of intersected **twolc** rules.

Depending on the application, one approach may have an advantage over the other. A cascade of replace rules is well-suited for cases in which an upper language string is typically very different from its lower language counterpart. In such cases we are often interested only in string-to-string mappings and do not care about how the strings are exactly aligned with respect to each other. But if it is important to describe the relation in terms of exact symbol-to-symbol correspondences, **twolc** rules

may be a more natural choice because that is precisely what they were designed to do.

References

Antworth, E. L. (1990). *PC-KIMMO: a two-level processor for morphological analysis*. Number 16 in Occasional publications in academic computing. Summer Institute of Linguistics, Dallas.

Armstrong, S. (1996). Multext: Multilingual text tools and corpora. In Feldweg, H. and Hinrichs, E. W., editors, *Lexikon und Text*, pages 107–112. Max Niemeyer Verlag, Tuebingen.

Beesley, K. R. (1989). Computer analysis of Arabic morphology: A two-level approach with detours. In *Third Annual Symposium on Arabic Linguistics*, Salt Lake City. University of Utah. Published as Beesley, 1991.

Beesley, K. R. (1990). Finite-state description of Arabic morphology. In *Proceedings of the Second Cambridge Conference on Bilingual Computing in Arabic and English*. No pagination.

Beesley, K. R. and Karttunen, L. (2003). *Finite State Morphology*. CSLI Publications, Palo Alto, CA.

Black, A., Ritchie, G., Pulman, S., and Russell, G. (1987). Formalisms for morphographemic description. In *Proceedings of the Third Conference of the European Chapter of the Association for Computational Linguistics*, pages 11–18.

Carter, D. (1995). Rapid development of morphological descriptions for full language processing systems. In *Proceedings of the Seventh Conference of the European Chapter of the Association for Computational Linguistics*, pages 202–209.

Chomsky, N. and Halle, M. (1968). *The Sound Pattern of English*. Harper and Row, New York.

Dalrymple, M., Doron, E., Goggin, J., Goodman, B., and McCarthy, J., editors (1983). *Texas Linguistic Forum, Vol. 22*. Department of Linguistics, The University of Texas at Austin, Austin, TX.

Eisner, J. (1997). Efficient generation in primitive Optimality Theory. In *Proceedings of the 35th Annual ACL and 8th EACL*, pages 313–320, Madrid. Association for Computational Linguistics.

Frank, R. and Satta, G. (1998). Optimality theory and the generative complexity of constraint violability. *Computational Linguistics*, 24(2):307–316.

Gajek, O., Beck, H. T., Elder, D., and Whittemore, G. (1983). LISP implementation. In Dalrymple, M., Doron, E., Goggin, J., Goodman, B., and McCarthy, J., editors, *Texas Linguistic Forum, Vol. 22*, pages 187–202. Department of Linguistics, The University of Texas at Austin, Austin, TX.

Grimley-Evans, E., Kiraz, G. A., and Pulman, S. G. (1996). Compiling a partition-based two-level formalism. In *COLING'96*, Copenhagen. cmp-lg/9605001.

Johnson, C. D. (1972). *Formal Aspects of Phonological Description*. Mouton, The Hague.

Kager, R. (1999). *Optimality Theory*. Cambridge University Press, Cambridge, England.

Kaplan, R. M. and Kay, M. (1981). Phonological rules and finite-state transducers. In *Linguistic Society of America Meeting Handbook, Fifty-Sixth Annual Meeting*, New York. Abstract.

Kaplan, R. M. and Kay, M. (1994). Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378.

Karttunen, L. (1983). KIMMO: a general morphological processor. In Dalrymple, M., Doron, E., Goggin, J., Goodman, B., and McCarthy, J., editors, *Texas Linguistic Forum, Vol. 22*, pages 165–186. Department of Linguistics, The University of Texas at Austin, Austin, TX.

Karttunen, L. (1995). The replace operator. In *ACL'95*, Cambridge, MA. cmp-lg/9504032.

Karttunen, L. (1996). Directed replacement. In *ACL'96*, Santa Cruz, CA. cmp-lg/9606029.

Karttunen, L. (1998). The proper treatment of optimality in computational phonology. In *FSMNLP'98. International Workshop on Finite-State Methods in Natural Language Processing*, Bilkent University, Ankara, Turkey. cmp-lg/9804002.

Karttunen, L. and Beesley, K. R. (1992). Two-level rule compiler. Technical Report ISTL-92-2, Xerox Palo Alto Research Center, Palo Alto, CA.

Karttunen, L., Kaplan, R. M., and Zaenen, A. (1992). Two-level morphology with composition. In *COLING'92*, pages 141–148, Nantes, France.

Karttunen, L., Koskenniemi, K., and Kaplan, R. M. (1987). A compiler for two-level phonological rules. In Dalrymple, M., Kaplan, R., Karttunen, L., Koskenniemi, K., Shaio, S., and Wescoat, M., editors, *Tools for Morphological Analysis*, volume 87-108 of *CSLI Reports*, pages 1–61. Center for the Study of Language and Information, Stanford University, Palo Alto, CA.

Karttunen, L., Uszkoreit, H., and Root, R. (1981). Morphological analysis of Finnish by computer. In *Proceedings of the 71st Annual Meeting of SASS*. Albuquerque, New Mexico.

Kempe, A. and Karttunen, L. (1996). Parallel replacement in finite-state calculus. In *COLING'96*, Copenhagen. cmp-lg/9607007.

Kisseberth, C. (1969). On the abstractness of phonology. *Papers in Linguistics*, 1:248–282.

Koskenniemi, K. (1983). Two-level morphology: A general computational model for word-form recognition and production. Publication 11, University of Helsinki, Department of General Linguistics, Helsinki.

Koskenniemi, K. (1984). A general computational model for word-form recognition and production. In *COLING'84*, pages 178–181.

Koskenniemi, K. (1986). Compilation of automata from morphological two-level rules. In Karlsson, F., editor, *Papers from the Fifth Scandinavian Conference on Computational Linguistics*, pages 143–149.

McCarthy, J. J. (2002). *The Foundations of Optimality Theory*. Cambridge University Press, Cambridge, England.

Partee, B. H., ter Meulen, A., and Wall, R. E. (1993). *Mathematical Methods in Linguistics*. Kluwer, Dordrecht.

Petitpierre, D. and Russel, G. (1995). MMORPH — the multext morphology program. Technical report, Carouge, Switzerland. Multext Deliverable 2.3.1.

Prince, A. and Smolensky, P. (1993). Optimality Theory: Constraint interaction in generative grammar. Technical report, Rutgers University, Piscataway, NJ. RuCCS Technical Report 2. Rutgers Center for Cognitive Science.

Pulman, S. (1991). Two level morphology. In Alshawi, H., Arnold, D., Backofen, R., Carter, D., Lindop, J., Netter, K., Pulman, S., Tsujii, J., and Uskoreit, H., editors, *ET6/1 Rule Formalism and Virtual Machine Design Study*, chapter 5. CEC, Luxembourg.

Ritchie, G., Black, A., Pulman, S., and Russell, G. (1987). The Edinburgh/Cambridge morphological analyser and dictionary system (version 3.0) user manual. Technical Report Software Paper No. 10, Department of Artificial Intelligence, University of Edinburgh.

Ritchie, G., Russell, G., Black, A., and Pulman, S. (1992). *Computational Morphology: Practical Mechanisms for the English Lexicon*. MIT Press, Cambridge, Mass.

Ruessink, H. (1989). Two level formalisms. In *Utrecht Working Papers in NLP*, volume 5.

Schützenberger, M.-P. (1961). A remark on finite transducers. *Information and Control*, 4:185–196.