The Replace Operator

Lauri Karttunen

- 19 December 1994 -
THE REPLACE OPERATOR

Lauri Karttunen

Rank Xerox Research Centre
Grenoble
(Karttunen@xerox.fr)

Abstract

This paper introduces to the calculus of regular expressions a replace operator and defines a set of replacement expressions that concisely encode several alternate variations of the operation. Replace expressions denote regular relations, defined in terms of other regular-expression operators. The basic case is unconditional obligatory replacement. We develop several versions of conditional replacement that allow the operation to be constrained by context. The replacement operation is included in the Xerox finite-state calculus.

0. Introduction

Linguistic descriptions in phonology, morphology, and syntax typically make use of an operation that replaces some symbol or sequence of symbols by another sequence or symbol. We consider here the replacement operation in the context of finite-state grammars. This includes many frameworks for phonological, morphological and syntactic description. Kaplan and Kay (1981, 1994) demonstrate that classical phonological rewrite rules can be implemented as finite-state transducers. The two-level model of Koskenniemi (1983) presents another finite-state formalism for constraining symbol-by-symbol replacements in morphology. The constraint grammar of Karlsson et al. (1994) has its own replacement formalism designed for morphological and syntactic disambiguation. It employs constraint rules that delete given morphological or syntactic tags in specified contexts. Finite-state syntax, proposed by Koskenniemi (1990), Koskenniemi, Taapanainen, and Voutilainen (1992), and Voutilainen (1994), accomplishes the same task by adapting the two-level formalism to express syntactic constraints. Each of these frameworks has its own rule formalism for replacement operations.

Our purpose in this paper is twofold. One is to define replacement in a more general way than is done in these formalisms, explicitly allowing replacement to be constrained by input and output contexts, as in two-level rules, but without the restriction of only single-symbol replacements. The second objective is to define replacement within a general calculus of regular expressions so that replacements can be conveniently combined with other kinds of operations, such as composition and union, to form complex expressions. We have already incorporated the new expressions into our implementation of the finite-state calculus. Thus we can construct transducers directly from replacement expressions as part of the general calculus without invoking any special rule compiler.

We start with a standard kind of regular expression language and add to it two new operators, \( \to \) and \( \overrightarrow{\to} \). These new operators can be used to describe regular relations which relate the strings of one regular language to the strings of another regular language that contain the specified replacement. The replacement may be unconditional or it may be restricted by left and right contexts. The \( \to \) operator makes the replacement obligatory, \( \overrightarrow{\to} \) makes the replacement optional. For the sake of completeness, we also define the inverse operators, \( \leftarrow \) and \( \overleftarrow{\to} \), and the corresponding bidirectional variants, \( \leftrightarrow \) and \( \leftrightarrow \). Several new types of regular expressions have been defined for these operators, and they are described below.

The replacement operators are close relatives of the rewrite-operator defined in Kaplan and Kay 1994, but they are not identical to it. We discuss their relationship in a section at the end of the paper where we also point out the differences between our replacement expressions and two-level rules (Koskenniemi 1993).

0. 1. Simple regular expressions

The replacement operators are defined by means of regular expressions. Some of the operators we use to define them are specific to Xerox implementations of the finite-state calculus,
but equivalent formulations could easily be found in other notations.

The table below describes the types of expressions and special symbols that are used to define the replacement operators.

| Symbol | Description |
|--------|-------------|
| (A)    | option (union of A with the empty string) |
| ~A     | complement (negation) |
| \A     | term complement (any symbol other than A) |
| $A     | contains (all strings containing at least one A) |
| A*     | Kleene star |
| A+     | Kleene plus |
| A/B    | ignore (A possibly interspersed with strings from B) |
| A B    | concatenation |
| A | B | union |
| A & B  | intersection |
| A - B  | relative complement (minus) |
| A .o. B | crossproduct (Cartesian product) |
| A => B _ C | restriction (A only between B and C) |

Square brackets, [], are used for grouping expressions. Thus [A] is equivalent to A while (A) is not. The order in the above table corresponds to the precedence of the operations. The prefix operators (~, \, and $) bind more tightly than the postfix operators (*, +, and /), which in turn rank above concatenation. Union, intersection, and relative complement are considered weaker than concatenation but stronger than crossproduct and composition. Our new replacement operator goes in a class between the Boolean operators and composition. Operators sharing the same precedence are interpreted left-to-right. Taking advantage of all these conventions, the fully bracketed expression

\[
[[[-[a]]* [[[b]/x]]] | c] \cdot x. \ d;
\]

can be rewritten more concisely as

\[
-a^* b/x | c \cdot x. \ d;
\]

Note that expressions that contain the crossproduct (\cdot x.) or the composition (\cdot o.) operator describe regular relations rather than regular languages. A regular relation is a mapping from one regular language to another one. Regular languages correspond to simple finite-state automata; regular relations are modeled by finite-state transducers.

In the relation A \cdot x. B, we call the first member, A, the upper language and the second member, B, the lower language. This choice of words is motivated by the linguistic tradition of writing the result of a rule application underneath the original form. In a cascade of compositions, R1 .o. R2, ..., .o. Rn, that models a linguistic derivation by rewrite-rules, the upper side of the first relation, R1, contains the "underlying lexical form," while the lower side of the last relation, Rn, contains the resulting "surface form."

We also have operators that extract one side of the regular relation: R.1 describes the upper-side language of the relation R; the language on the lower side is R.2.

Some of the operations listed above are defined only for simple regular languages (intersection, complement, relative complement, crossproduct); others apply to both languages and relations (concatenation, union, option, Kleene star and plus). See Kaplan and Kay 1994 for discussion.

To make the notation less cumbersome, we systematically ignore the distinction between the language A and the identity relation that maps every string of A to itself. Correspondingly, a simple automaton may be thought of as representing a language or as a transducer for its identity relation. For the sake of convenience, we also equate a language consisting of a single string with the string itself. Thus the expression abc may denote, depending on the context, (i) the string abc, (ii) the language consisting of the string abc, and (iii) the identity relation on that language.

We recognize two kinds of symbols: simple symbols (a, b, c, etc.) and fst pairs (a:b, y:z, etc.). An fst pair a:b can be thought of as the crossproduct of a and b, the minimal relation consisting of a (the upper symbol) and b (the lower symbol). Because we regard the identity relation on A as equivalent to A, we can and usually do write a:a as just a. There are two special symbols

0 \epsilon (the empty string).
?
\any symbol in the known alphabet and its extensions.
A third special symbol, #, is introduced in section 2.4.2. The escape character, %, allows letters that have a special meaning in the calculus to be used as ordinary symbols. Thus %& denotes a literal ampersand as opposed to &, the intersection operator; %0 is the ordinary zero symbol.

The following simple expressions appear frequently in our formulas:

- $\varepsilon$ the empty string language. The corresponding minimal automaton, the epsilon fsm, consists of a single final start state without any transitions.
- $\emptyset$ the null set. The corresponding minimal automaton, the null fsm, consists of a single nonfinal start state without any transitions.
- $?^*$ the universal ("sigma-star") language: all possible strings of any length including the empty string. Alternately, we may view it as the universal identity relation. The corresponding minimal automaton consists of a single final start state with a single looping transition labeled ?. It has no other symbols in its alphabet.

1. Unconditional replacement

To the regular-expression language described above, we add the new replacement operator. The unconditional replacement of UPPER by LOWER is written

$$\text{UPPER} \rightarrow \text{LOWER}$$

Here UPPER and LOWER are any regular expressions that describe simple regular languages. We define this replacement expression as

$$[\text{NO\_UPPER} \ [\text{UPPER} \ .\x. \text{LOWER}] ]^* \text{NO\_UPPER} ;$$

where NO\_UPPER abbreviates $\emptyset[\text{UPPER} - [\varepsilon]]$. The definition describes a regular relation whose members contain any number (including zero) of iterations of [UPPER .x. LOWER], possibly alternating with strings not containing UPPER that are mapped to themselves. Note that if UPPER does not contain the empty string, $\emptyset[\text{UPPER}]$ and $\emptyset[\text{UPPER} - [\varepsilon]]$ are equivalent. But if UPPER contains the empty string, $\emptyset[\text{UPPER}]$ is null whereas $\emptyset[\text{UPPER} - [\varepsilon]]$ at least contains the empty string. We need the latter language in our definition to get the intended meaning for UPPER $\rightarrow$ LOWER.

1.1. Examples

We illustrate the meaning of the replacement operator with a few simple examples. The regular expression

$$a \ b \ | \ c \rightarrow x ;$$

(same as [[a b] | c] $\rightarrow x$)

describes a relation consisting of an infinite set of pairs such as

$$a \ b \ a \ c \ a \ x \ a \ x \ a$$

where all occurrences of ab and c are mapped to x interspersed with unchanging pairings. It also includes all possible pairs like

$$x \ a \ x \ a \ x \ a \ x \ a$$

that do not contain either ab or c anywhere.

Figure 1 shows the state diagram of a transducer that encodes this relation. The transducer consists of states and arcs that indicate a transition from state to state over a given pair of symbols. For convenience we represent identity pairs by a single symbol; for example, we write a:a as a. The symbol ? represents the identity pairs of symbols that are not explicitly present in the network. In this case, ? stands for any identity pair other than a:a, b:b, c:c, and x:x. Transitions that differ only with respect to the label are collapsed into a single multiply labelled arc. The state labeled 0 is the start state. Final states are distinguished by a double circle.
rection to an $x$, the transducer produces three results, namely $ab$, $c$, and $x$, where the last alternative represents an unchanged $x$ in the upper side language. The corresponding paths through the network are $0\rightarrow 1\rightarrow 0$, $0\rightarrow 0$ (over $c:x$), and $0\rightarrow 0$ (over $x$).

The replace operator $\rightarrow$ is like the crossproduct operator $\times$, in that it constructs a relation out of two regular languages. Consequently, it can be combined with other relations in composition, union, concatenation and other operations that are defined for regular relations.

Replacement expressions such as $[ab | c \rightarrow x]$ may be thought of as denoting string replacement rules. The application of such a rule to a string can be modeled quite directly in the following way:

(i) construe the input string as an identity relation;
(ii) compose the input with the replacement relation;
(iii) extract the lower-side language from the composite relation.

More generally, we may apply replacement rules to any regular set, such as a lexicon, thus rewriting in a single operation a possibly infinite set of strings. The development of lexical transducers (Karttunen, Kaplan, and Zaenen 1992, Karttunen 1994) is based on this insight.

In (ii), the upper side of the replacement relation is matched against the input. We generally refer to the upper side of $\rightarrow$ relations as the input side because they are generally intended to be applied in that direction. However, all relations are inherently bidirectional; from a formal point of view, there is no privileged input side. We discuss the effect of composing a string with the lower side of a replacement relation in section 1.3.

It is important to keep in mind that the result of (ii) is not an output string but a relation that links the input to the output. To get the output, we need another auxiliary operation that extracts the lower-side language, which may contain any number of output strings. We have such an operator in our regular-expression language, namely $\cdot 2$. Thus we can easily write single regular expressions that capture the conventional notion of "applying a rule." For example, consider the application of the relation $[ab \rightarrow x]$ to the string $baab$. In this case, the expression is $[baab .o. ab \rightarrow x] .2$, more perspicuously written as

\[
[b \ a \ a \ b.
\cdot o.
\ a \ b \rightarrow x] .2 ;
\]

which simultaneously (i) constructs the input language, (ii) composes it with replacement relation, and (iii) extracts the lower-side language of relation. Here the result of (ii) is the simple input/output relation:

\[
[baab
ba \ x]
\]

We use this graphical way of illustrating the effect of replacement relations throughout the paper, but we avoid the term "rule" in order not to be confused by other possible interpretations of this term. On the other hand, we often fail to distinguish carefully between regular expressions, the corresponding automata, and the relations they express, treating them as interchangeable for the sake of convenience.

In case a given input string matches the replacement relation in two ways, two outputs are produced. For example,

\[
a \ b \ | \ b \ c \rightarrow x ;
\]

maps $abc$ to both $ax$ and $xc$:

\[
a \ b \ c ,
a \ b \ c
a \ x ,
a \ x
x \ c ,
x \ c
\]

Because the two targets of the replacement operation overlap here, only one of them can be replaced at one time. Thus we get two outputs in such cases. The corresponding transducer
paths in Figure 2 are 0-1-3-0 and 0-2-0-0, where the last 0-0 transition is over a c arc.

If this ambiguity is not desirable for some purpose, we may write two replacement expressions and combine them in a way that indicates which replacement should be preferred if a choice has to be made. For example, if the ab match should have precedence, we can write

\[
\begin{align*}
0 \rightarrow x \\
0 \rightarrow x ;
\end{align*}
\]

In the latter case NO_UPPER would denote a null relation here because $\neg S[\{\}]$ is null. Consequently, [17] as a whole would denote something quite different from what we intend, namely, the null relation.

If UPPER describes the null set, as in

\[
\neg S[\{\}] \rightarrow a \mid b ;
\]

the LOWER part is irrelevant because there is no replacement. This expression is a description of the sigma-star language.

If LOWER describes the empty set, replacement becomes deletion. For example,

\[
a \mid b \rightarrow \[] ;
\]

removes all as and bs from the input.

All the replacement expressions discussed so far have one property in common: the language on the upper side of the relation is the universal language. That is an important property since it means that every string, including all the ones where no replacement takes place, have a counterpart on the lower side. The corresponding transducer never fails on any input in this direction. However, in one situation that is not the case. If LOWER describes the null set, as in

\[
\neg S[a \mid b] ;
\]

all strings containing UPPER, here a or b, are excluded from the upper side language. Everything else is mapped to itself. An equivalent expression is $\neg S[a \mid b]$.

1.3. Inverse and bidirectional replacement

When two relations are composed with one another, the result generally depends on which relation is ordered first in the composition. The same is true for the application of a relation to a string. So far we have only looked at cases where the input is ordered first and composed with the upper side of a replacement relation. In that direction, the $\rightarrow$ relation is privative: it replaces every instance of UPPER by LOWER. For example,

\[
\begin{align*}
0 \rightarrow x \\
0 \rightarrow x ;
\end{align*}
\]

where no ab on the upper side survives to the lower side. But the $\rightarrow$ relation is not privative
in the inverse direction. If the relation and the output string of the above application are composed in the opposite order, we get two results:

\[
\begin{align*}
\text{a b} \rightarrow \text{x} & \\
\text{.o.} & \quad \text{yields} \\
\text{x} & \\
\end{align*}
\]

The latter case represents an x in the upper language of \([a \ b \rightarrow x]\) that is mapped to itself.

These examples are useful here because they help us to illustrate the meaning of \(<-\), the inverse replacement operator.

\[
\text{UPPER} \leftarrow \text{LOWER}
\]

is defined as the inverse of the relation \(\text{LOWER} \rightarrow \text{UPPER}\). The difference between \(\rightarrow\) and \(\leftarrow\) can be seen by comparing the above two examples with the corresponding applications of \([a \ b \leftarrow x]\).

\[
\begin{align*}
\text{a b} & \\
\text{.o.} & \quad \text{yields} \\
\text{x} & \\
\end{align*}
\]

Two results are produced because \(ab\) in the upper language may correspond to either \(ab\) or \(x\) in the lower language of \([a \ b \leftarrow x]\).

\[
\begin{align*}
\text{a b} \leftarrow \text{x} & \\
\text{.o.} & \quad \text{yields} \\
\text{x} & \\
\end{align*}
\]

The inverse replacement, \(a \ b \leftarrow x\), is privative in upward direction: no \(x\)s on the lower side of \([a \ b \leftarrow x]\) survive to the upper side.

The bidirectional replacement relation,

\[
\text{UPPER} \leftrightarrow \text{LOWER}
\]

can be defined as a composition of the corresponding \(\leftarrow\) and \(\rightarrow\) expressions. The most straightforward definition would be

\[
\begin{align*}
\text{UPPER} \rightarrow \text{LOWER} \\
\text{.o.} \\
\text{UPPER} \leftarrow \text{LOWER} ;
\end{align*}
\]

However, this simple composition of \(\rightarrow\) and \(\leftarrow\) expressions does not have the intended meaning. It fails to connect \(\text{UPPER}\) and \(\text{LOWER}\) because of a mismatch in the middle layer of the composition.

To get the desired result, we introduce in the definition of \(\leftrightarrow\) an arbitrary auxiliary symbol, \(\@\). The purpose of \(\@\) is to serve as a pivot that links the upper \(\text{UPPER}\) and the lower \(\text{LOWER}\) in the intended way:

\[
\begin{align*}
\text{UPPER} & \rightarrow \%@ \\
\text{.o.} \\
\%@ & \leftarrow \text{LOWER} ;
\end{align*}
\]

Because the linking \(\@\) is eliminated in the composition, in the resulting relation \(\text{UPPER}\) in the upper language always corresponds to \(\text{LOWER}\) in the lower language, and vice versa.

To complete the definition, we only need to eliminate any trace of \(\@\)s from the upper and the lower side of the relation. A simple way to accomplish the task is to compose both sides with the relation \(-S[\%@]\) that eliminates all strings containing \(\@\) and maps everything else to itself.

Thus

\[
\begin{align*}
\text{UPPER} \leftrightarrow \text{LOWER} \\
\text{.o.} \\
\end{align*}
\]

is defined as

\[
\begin{align*}
-\$[\%@] \\
\text{UPPER} & \rightarrow \%@ \\
\text{.o.} \\
\%@ & \leftarrow \text{LOWER} \\
\text{.o.} \\
-\$[\%@] ;
\end{align*}
\]

The bidirectional relation is privative in both directions. For example, \([a \ b \leftrightarrow x]\) maps all instances of \(ab\) to \(x\), and vice versa. Because the replacement is obligatory, in this case there are no \(x\)s in the upper side language of the relation and no \(ab\) sequences on the lower side. Nevertheless, there are cases in which the lower side of an \(\text{UPPER} \leftrightarrow \text{LOWER}\) relation contains instances of \(\text{UPPER}\), even though the relation is privative. They may arise as a result of the replacement. For example, consider the relation

\[
\begin{align*}
\text{a b} & \leftrightarrow \text{b} ;
\end{align*}
\]

applied to the string \(aab\):

\[
\begin{align*}
\text{a a b} & \\
\text{.o.} & \quad \text{yields} \\
\text{a b} & \\
\end{align*}
\]

In this case we have an ab on the lower side but it does not correspond to an ab on the upper side.

The auxiliary symbol in the definition of <-> is a preview of the method we use extensively in section 2. to define conditional replacement.

1.4. Optional replacement

The optional versions of unconditional replacement are derived simply by augmenting LOWER with UPPER in the replacement relation.

[33]

UPPER (->) LOWER

is defined as

[34]

UPPER -> [LOWER | UPPER] ;

The optional replacement relation maps UPPER to both LOWER and UPPER. The optional versions of <-> are defined in the same way.

2. Conditional replacement

We now extend the notion of simple replacement by allowing the operation to be constrained by a left and a right context. A conditional replacement expression has four components: UPPER, LOWER, LEFT, and RIGHT. They must all be regular expressions that describe a simple language, not a relation. In other words, they may not contain any fst pairs, the crossproduct (.x.) or the composition (.o.) operators, and, of course, not the replacement operator -> itself, unless these expressions are wrapped inside some operator, such as .1 and .2, that extracts a simple language from a relation. We write the replacement part UPPER -> LOWER, as before, and the context part as LEFT _ RIGHT, where the underscore indicates where the replacement takes place.

In addition, we need a separator between the replacement and the context part. Traditionally, replacement rules are written

[35]

UPPER -> LOWER / LEFT _ RIGHT.

However, that notation is not suitable for us. One trivial reason is that / in our regular expression calculus is the "ignore" operator. A more serious issue is that there are several ways in which a context may be used to constrain the replacement. We recognize four modes of applying the constraints; thus we have to distinguish them in our notation. For this reason we use four alternate separators, ||, //, \ and \\/, which gives rise to four types of conditional replacement expressions:

(1) Upward-oriented:

UPPER -> LOWER || LEFT _ RIGHT ;

(2) Right-oriented:

UPPER -> LOWER // LEFT _ RIGHT ;

(3) Left-oriented:

UPPER -> LOWER \ \ LEFT _ RIGHT ;

(4) Downward-oriented:

UPPER -> LOWER \ \ LEFT _ RIGHT ;

All four kinds of replacement expressions describe a relation that maps UPPER to LOWER between LEFT and RIGHT leaving everything else unchanged. The difference is in the interpretation of "between LEFT and RIGHT." We must distinguish whether we are talking about the upper context, the lower context, or some combination of the two. We provide four interpretations, soon to be made precise, but one can imagine others. We start with the upward-oriented (||) version because it is the most straightforward way of using a context to constrain a replacement, therefore, the easiest to understand.

Except for the context separator, the syntax of all four versions of replacement expressions is the same. Later on we will also allow for multiple contexts, with a separating comma: LEFT1 _ RIGHT1, LEFT2 _ RIGHT2, ... , LEFTn _ RIGHTn. This extension of the formalism has not yet been implemented.

The LEFT and RIGHT parts are optional. Thus

[37]

UPPER -> LOWER || LEFT _ ;
UPPER -> LOWER || _ RIGHT ;
UPPER -> LOWER || _ ;

are also well-formed regular expressions. The compiler supplies the empty string language for a missing context. Thus the last of the three expressions above is equivalent to

[38]

UPPER -> LOWER || [ ] _ [ ] ;

The conditional versions of inverse, <->, and bi-conditional, <->, replacement are just as their unconditional counterparts except for the constraints imposed by the context.
2.1. Definition of conditional replacement

2.1.1. Overview: divide and conquer

We define \( \textup{UPPER} \rightarrow \textup{LOWER} \mid \mid \textup{LEFT} _\ \textup{RIGHT} \) and the other versions of conditional replacement in terms of expressions that are already in our regular-expression language, including the unconditional version just defined. Our general intention is to make the conditional replacement behave exactly like unconditional replacement except that the operation does not take place unless the specified context is present.

This may seem a simple matter but it is not, as Kaplan and Kay 1994 show. There are several sources of complexity. One is that the part that is being replaced may at the same time serve as the context of another adjacent replacement. Another complication is the fact we just stated: there are several ways to constrain a replacement by a context. In many cases the effect is the same regardless of how the contextual constraints are applied, in other cases the outcomes are strikingly different.

We solve the problem by using a technique that was originally invented for the implementation of phonological rewrite rules (Kaplan and Kay 1981, 1994) and later adapted for two-level rules (Kaplan, Karttunen, Koskenniemi 1987, Karttunen and Beesley 1992). The strategy is first to decompose the complex relation into a set of relatively simple components, define the components independently of one another, and then define the whole operation as a composition of these auxiliary relations.

We define the conditional replacement relation as the composition of several intermediate relations. There are six of them, to be defined shortly:

1. InsertBrackets
2. ConstrainBrackets
3. LeftContext
4. RightContext
5. Replace
6. RemoveBrackets

Relations (1), (5), and (6) involve the unconditional replacement operator defined in the previous section.

The composition of these relations in the order given defines the upward-oriented (\( \mid \mid \) ) replacement. When an input string is composed with the upper side of the relation, it replaces \( \textup{UPPER} \) by \( \textup{LOWER} \) when \( \textup{UPPER} \) is between \( \textup{LEFT} \) and \( \textup{RIGHT} \) in the input context, leaving the string otherwise unchanged. We will define the other variants of the replacement expression shortly.

Two auxiliary symbols, < and >, are introduced in (1) and (6). The distribution of the auxiliary brackets is controlled by (2), (3), and (4). (2) constrains them with respect to each other, (3) and (4) with respect to left and right contexts. The left bracket, <, indicates the end of a complete left context. The right bracket, >, marks the beginning of a complete right context. The replacement expression (5) includes the auxiliary brackets on both sides of the relation. The final result of the composition does not contain any brackets. (1) removes them from the upper side, (6) from the lower side.

2.1.2. Auxiliary definition: the restrict operator

The constraints (3) and (4) on contexts and auxiliary brackets can be defined most concisely using the restrict-operator, =>, originally introduced by Koskenniemi (1983) for the two-level rule formalism. Because => is less known than the other regular expression operators that appear in our definitions, we define the restrict operator here explicitly. In the Xerox regular expression calculus

\[
\texttt{CENTER} \Rightarrow \texttt{LEFT} \_ \texttt{RIGHT} ;
\]

is defined as

\[
\neg\{\neg\{\?* \texttt{LEFT} \}\, \texttt{CENTER} \, \?*\} \& \neg\{\?* \texttt{CENTER} \, \neg\{\texttt{RIGHT} \, \?*\}\} ;
\]

Here \( \texttt{CENTER} \), \( \texttt{LEFT} \), and \( \texttt{RIGHT} \) may be any expressions that describe a simple regular language. The first of the two conjoined expressions in the definition excludes strings which contain an instance of \( \texttt{CENTER} \) that is not immediately preceded by \( \texttt{LEFT} \). The second conjunct excludes strings in which \( \texttt{CENTER} \) is not immediately followed by \( \texttt{RIGHT} \). Taken together, the two conditions define a language in which strings described by \( \texttt{CENTER} \) may only appear between \( \texttt{LEFT} \) and \( \texttt{RIGHT} \) strings. A more concise, equivalent definition in terms of Kaplan’s If-P-then-S and If-S-then-P operators is given in section 7.3 of Kaplan and Kay (1994).
Without the restrict => operator, the definitions of the two context conditions, (3) and (4), in the next section would be more cumbersome.

2.1.3. Basic definition

The full specification of the six component relations is given below. Note that in our regular expression language, we have to prefix our auxiliary context markers with the escape symbol % to distinguish them from other uses of < and > in the Xerox finite-state calculus.

We define the component relations in the following way. Note that UPPER, LOWER, LEFT, and RIGHT are placeholders for regular expressions of any complexity. In each case we give a regular expression that precisely defines the component, followed by an English sentence describing the same language or relation.

(1) InsertBrackets

\[
[] <- %< | %> ;
\]

*The relation that eliminates from the upper side language all context markers that appear on the lower side.*

Alternatively one may think of the relation as a transducer that freely inserts brackets mapping in the other direction. It is the inverse of the relation in clause (6) of the definition. For the technical reason just explained we have to write our brackets as %< and %> in the regular expression.

(2) ConstrainBrackets

\[
~$[%< %>] ;
\]

*The language consisting of strings that do not contain <> anywhere.*

Note that the strings on the lower side of (1) are interspersed with unconstrained bursts of < and >. In particular, if two brackets occur next to each other, they may come in either order, < > or > <. For reasons that will soon become evident, we only want to allow the second possibility. The composition of (1) and (2) invokes this constraint.

(3) LeftContext

\[
[ %< => LEFT' ] &
[ LEFT' => _ %</%> ] ;
\]

*The language in which any instance of < is immediately preceded by LEFT', and every LEFT' is immediately followed by <.*

Here LEFT' abbreviates [LEFT/[%<|%>] - [%?* %<]]. In other words, the constraint ignores all brackets except for a final < following LEFT. The first conjunct lets a single left bracket follow a left context; the second conjunct forces a < after a left context. Because right brackets are ignored everywhere, they may intervene between the end of the left context and its marker: LEFT>> is legal.

(4) RightContext

\[
[ %> => _ RIGHT' ] &
[ RIGHT' => %>/%< _ ] ;
\]

*The language in which any instance of > is immediately followed by RIGHT', and any RIGHT' is immediately preceded by >.*

Here RIGHT' is an abbreviation for [RIGHT/[%<|%>] - [%?* %>]]. We ignore all brackets except for an initial right bracket. The right context component is the mirror image of (3). It can be derived from the corresponding left context expression by reversing the language and the brackets.

(5) Replace

\[
%< UPPER' %> -> %< LOWER %> ;
\]

*The unconditional replacement of UPPER (ignoring any internal brackets) by LOWER between the appropriate context brackets.*

Here UPPER' is an abbreviation for UPPER /[%<|%>]. We could eliminate the brackets because here they serve no further purpose. But for other versions of the rule, one or both of the brackets must be kept, so we may just as well always preserve them in the replacement.

This replacement clause is intended only for nonempty UPPERS. It is useful to have another replacement formula for epenthesis rules. We will discuss this case later in sections 2.4.3 and 2.4.4.

The inverse and bidirectional versions of conditional replacement are defined with the corresponding operator, <-> or <->, respectively. See section 1.3 for discussion.
The relation that maps the strings of the upper language to the same strings without any context markers.

The upper side brackets are eliminated by the inverse replacement defined in (1).

The complete definition of our first version of conditional replacement is the composition of these six relations:

\[
\text{UPPER} \rightarrow \text{LOWER} \mid | \text{LEFT} \_ \text{RIGHT} ;
\]

\[\text{InsertBrackets} .o.\]
\[\text{ConstrainBrackets} .o.\]
\[\text{LeftContext} .o.\]
\[\text{RightContext} .o.\]
\[\text{Replace} .o.\]
\[\text{RemoveBrackets} ;\]

The composition with the left and right context constraints prior to the replacement means that any instance of \text{UPPER} that is subject to replacement is surrounded by the proper context on the upper side. Within this region, replacement operates just as it does in the unconditional case.

2.1.4. Left-, right-, and downward-oriented replacement

We may choose to delay the application of one or the other context constraint so that the constraint is checked on the lower side of the replacement relation. So far we have used only one out of four logical possibilities. Three other versions of conditional replacement can be defined by varying the order of the three middle relations in the composition. In the right-oriented version (\//), the left context is checked on the lower side of replacement:

\[
\text{UPPER} \rightarrow \text{LOWER} \// \text{LEFT} \_ \text{RIGHT} ;
\]

\[\text{Replace} .o.\]
\[\text{LeftContext} .o.\]
\[\text{RightContext} .o.\]

The first three versions roughly correspond to the three alternative interpretations of phonological rewrite rules discussed in Kaplan and Kay 1994. The upward-oriented version corresponds to simultaneous rule application; the right- and left-oriented versions can model rightward or leftward iterating processes, such as vowel harmony and assimilation.

The fourth logical possibility is that the replacement operation is constrained by the lower context.

\[
\text{UPPER} \rightarrow \text{LOWER} \\/ \text{LEFT} \_ \text{RIGHT} ;
\]

\[\text{Replace} .o.\]
\[\text{LeftContext} .o.\]
\[\text{RightContext} .o.\]

When the component relations are composed together in this manner, \text{UPPER} gets mapped to \text{LOWER} just in case it ends up between \text{LEFT} and \text{RIGHT} in the output string. This fourth version seems well-suited for the modeling of the currently popular optimality theory in phonology (Prince and Smolensky 1993).

2.2. Examples

Let us illustrate the consequences of these definitions with a few examples. Consider the upward oriented replacement relation:

\[
a \ b \rightarrow x \mid | \ c \ (d) \_ \ e \mid f ;
\]
The infinite set of strings in this relation includes pairs like:

\[ \text{cdabe, cabf, cdxe} \]

Consequently, the transducer compiled from the rule maps \text{cdabe} to \text{cdxe}. In the reverse direction, it maps \text{cdxe} to both \text{cdxe} and \text{cdabe} because in this case the \text{x} on the lower side could correspond to an \text{x} or to an \text{ab} on the upper side. As we mentioned already in connection with the unconditional version of the operation, we can write expressions such as:

\[ \text{[ cdabe } \]
\[ \text{a b -> x || c (d) _ e | f } \]

which simultaneously defines the replacement, applies it to \text{cdabe} and extracts the result. Here as before, \text{.2} denotes the operation that yields the lower-side language of a relation.

In order to understand the logic of the replacement constraints, it is useful to consider a concrete example in two stages. Let us first compose our sample word, not with the replacement expression itself but, with the initial four components in its definition:

\[ \text{[ c d a b e } \]
\[ \text{.o.} \]
\[ \text{InsertBrackets} \]
\[ \text{.o.} \]
\[ \text{ConstrainBrackets} \]
\[ \text{.o.} \]
\[ \text{LeftContext} \]
\[ \text{.o.} \]
\[ \text{RightContext} ; \]

This composition defines a very simple relation consisting of the pair:

\[ \text{c d a b e} \]
\[ \text{c < d < a b > e} \]

Two left brackets on the lower side mark the two places in the word that satisfy the left context of the rule. The single > indicates the beginning of a right context. Because the replacement target is between two context brackets, \text{ab} is replaced by \text{x} when we continue the composition with the remaining two component expressions:

\[ \text{c < d < a b > e} \]
\[ \text{Replace} \]
\[ \text{.o.} \]
\[ \text{RemoveBrackets ;} \]

Because the context markers are removed by the final composition, the result is the simple relation:

\[ \text{c d a b e} \]
\[ \text{c d x e} . \]

The transducer compiled from \{ab \rightarrow x || c (d) _ e | f\} of course does not contain any context markers. It transduces \text{cdabe} to \text{cdxe} in one single step without any intermediate levels. The auxiliary context markers are only used internally by the regular-expression compiler to hardwire the context constraints into the labels and state transitions of the network.

As already mentioned, one of the complications of conditional replacement is that the target of the replacement may simultaneously serve as a part of the context of an adjacent replacement. In such cases, it may happen that the four ways of invoking the context constraints lead to different results. Let us first consider four versions of the same replacement expression, starting with the upward-oriented version:

\[ \text{c d a b e} \]
\[ \text{a b -> x || a b _ a} ; \]

applied to the string \text{abababa}. The resulting relation is:

\[ \text{a b a b a b a} \]
\[ \text{a b x x a} \]

It is easy to see why this is so. The fictional intermediate bracketed result in this case is \text{>ab<ab<ab<<a} with two marked application sites. Because the replacement is obligatory, both occurrences of \text{ab} are replaced by \text{x}.

This example gives us an opportunity to point out the effect of the \text{ConstrainBrackets}, (2), that prohibits \text{<>} sequences. In the case at hand, the end of the left context meets the beginning of the right context in three places where \text{ab} is followed by an \text{a}. Without this constraint on the context markers, \text{>ab<ab<ab><a} would also be a valid intermediate result. Here the middle part \text{<ab>} matches the upper side of the replacement relation in two ways: (i) \text{<ab><}
(ignoring the intervening \(<\) and (ii) \(<ab>\).

In the first case, when \(<ab<--> is replaced by \(x\), the following \(ab\) remains unchanged because of the missing context bracket; in the second case the preceding \(<ab>\) survives for the same reason. In other words, without ConstrainBrackets the replacement would sometimes be optional.

A transducer for the \(ab -> x \mid \mid a b \_ a\) relation is shown in Figure 4.

![Figure 4: ab -> x \mid \mid a b \_ a](image)

The path through the network that maps \(abababa\) to \(abxxa\) is \(0-1-2-5-7-5-6-3\).

2.3. Right-, left- and downward-oriented replacement

Let us now consider the remaining three variants of our replacement expression, first the right-oriented version.

\[ab -> x // a b \_ a;\]

![Figure 5: ab -> x // a b \_ a](image)

Applying this to \(abababa\) we now get a different result:

\[abababab\]
\[baba\]
following the path \(0-1-2-5-6-1-2-3\).

Why? It is because the right-oriented version is defined so that the LeftContext constraint pertains to the lower side of the replace relation. If we postulate the intermediate bracketing for \(abababa\) we see that it meets the constraint: \(<ab><x><ab><a\). But the corresponding representation for \(abxxa\), \(<ab><x><x><a\), is not in the LeftContext language because the left bracket in \(x><x\) is not preceded by \(ab\). Consequently, this replacement result is eliminated in the composition with LeftContext.

The left-oriented version of the rule shows the opposite behavior because it constrains the left context on the upper side of the replacement relation and the RightContext constraint applies to the lower side.

\[ab -> x \_ a b \_ a;\]

![Figure 6: ab -> x \_ a b \_ a](image)

With \(abababa\) composed on the upper side, it yields

\[abababab\]
\[baba\]
by the path \(0-1-2-3-4-5-6-3\).

The first two occurrences of \(ab\) remain unchanged because neither one has the right context on the lower side to be replaced by \(x\).

Finally, the downward-oriented fourth version:

\[ab -> x \_ a b \_ a;\]

![Figure 7: ab -> x \_ a b \_ a](image)
This time, surprisingly, we get two outputs from the same input:

\[ \text{a b a b a b a , a b a b a b a} \]
\[ \text{a b x a b a a b a x a} \]

Path 0-1-2-5-6-1-2-3 yields abxaba, path 0-1-2-3-4-5-6-1 gives us ababxa.

It is easy to see that if the constraint for the replacement pertains to the lower side, then in this case it can be satisfied in two ways.

If LOWER overlaps with the left or the right context, different versions of the replacement rule often have interestingly different effects. Consider the upward- and right-oriented replacements when LOWER and LEFT are the same:

\[ a \rightarrow b \ | \ | b \_ \ vs. \ a \rightarrow b \ // \ b \_ \]

They make different replacements for inputs like baa:

\[ \text{b a a , b a a} \]

\[ a \rightarrow b \ | \ | b \_ ; \ b b a \]
\[ b a a \]
\[ a \rightarrow b \ // \ b \_ ; \ b b b \]

The upward-oriented version does not replace the second a because that a does not have a b on its right on the upper side. The right-oriented version replaces both as they both have a b on the right side in the output. Once the left-most a in a string of as is replaced by a b, all the adjacent as to the right of it follow suite. For example, baaaaa is replaced by bbbbb. This is why we call this version "right-oriented." The \| and // versions of the rule give the same result here as \| and //, respectively.

Now consider the opposite case where LOWER and RIGHT are the same:

\[ a \rightarrow b \ | \ | \_ b \ vs. \ a \rightarrow b \ \| \ \_ b \]

Here the left-oriented version iterates to the left for inputs like aab:

\[ \text{a a b , a a b} \]

\[ a \rightarrow b \ | \ | \_ b ; \ a b b \]

In the upward-oriented case, the first a remains unchanged because it is not next to b. The left-oriented makes the change because there is an adjacent b on the lower side. In this case, the right-oriented and upward-oriented replacements are equivalent; the downward-oriented version gives the same result as the left-oriented replacement.

If there is no overlap between UPPER, LOWER and the adjacent contexts, then all three versions of the replacement rule give exactly the same result.

2.4. Special cases

Let us consider some special cases that help us to clarify the semantics of the \( \rightarrow \) operator.

If LOWER contains at least the empty string, the language on the upper side of the replacement relation is always the sigma-star language, which contains all strings of any length, including the empty string. This holds for all replacement expressions, conditional or unconditional. The language of the lower side of the relation, on the other hand, is usually more constrained when the replacement is obligatory because it excludes the strings where UPPER has been replaced by LOWER.

2.4.1. Null components

As we noted before, we can also write replacement rules where LOWER describes the null set, as in

\[ a b \rightarrow \sim[] \ | \ | x \ b y ; \]

This yields a simple relation that maps everything to itself but it excludes all strings that contain xaby. Thus replacement by the null set amounts to a prohibition. A simpler equivalent regular expression is \( \sim[x a b y] \).

In any rule where UPPER describes a null set, such as

\[ \sim[] \rightarrow a \ | \ | x \ y ; \]

the other components are irrelevant because the expression as a whole just compiles to the sigma-star language, including the case where
LOWER describes the null set. The same goes for expressions where one or both contexts are null, as in

\[
\text{a} \rightarrow \neg \{ \} \_ \text{b} ;
\]

2.4.2. Boundary symbol

The context part of a replacement expression is implicitly extended to indefinitely far to the left and to the right beyond what is explicitly specified. Thus

\[
\text{LEFT} \_ \text{RIGHT}
\]

actually means

\[
?^* \text{LEFT} \_ \text{RIGHT} ?^*
\]

Although this is a useful convention, sometimes it is desirable to refer to the beginning or to the end of a string to constrain a replacement.

A special boundary symbol, #, indicates the beginning of a string when in the left context and the end of the string in the right context. For example,

\[
\text{a} \rightarrow \text{A} \mid \# \mid \text{b} _ ;
\]

replaces a by A in the beginning of a string and after a b. As the example shows, # is syntactically like any other regular expression symbol but the compiler gives the boundary symbol its intended interpretation and removes it from the alphabet of the network.

The special interpretation of # can be canceled by prefixing it with the escape character %.

2.4.3. Epsilon replacements

As we mentioned earlier, in connection with unconditional replacement expressions, if \text{UPPER} designates the empty string language, the result is also the identity relation regardless of what \text{LOWER} is. The definition of the Replace relation given above (5), would make this true for the conditional replacement as well.

But we choose to have a more useful result for the conditional case by using another Replace relation when \text{UPPER} is empty. If we in that case substitute

\[
\% > \% < \rightarrow \% < \text{LOWER} \%
\]

rather than

\[
\% < \text{UPPER/} \% < \% > \% > \rightarrow \\
\% < \text{LOWER} \%
\]

then the final relation that emerges from the composition is more interesting. It inserts one occurrence of \text{LOWER} in all places that meet the context constraints. Thus

\[
[ \] \rightarrow \text{LOWER} \mid \mid _ ;
\]

does not have the same meaning anymore as

\[
[ \] \rightarrow \text{LOWER} ;
\]

For example,

\[
an \ b \ a \ b \\
. o. \text{ yields}
\]

\[
[ \] \rightarrow x \ x^* a x^* b x^*
\]

because the \([ \] \rightarrow x\) relation freely introduces xs everywhere in the output string. The lower-side language here is \{a b\}/x. In contrast,

\[
a \ b \\
. o. \text{ yields}
\]

\[
[ \] \rightarrow x \mid \mid _ \ x a x b x
\]

because an empty context condition is always satisfied but only one copy of x is inserted at each site. Similarly,

\[
[ \] \rightarrow x \mid y \mid a \_ b ;
\]

yields a relation that inserts an x or a y between every a and b. Thus

\[
a b a b a \\
. o. \text{ yields}
\]

\[
[ \] \rightarrow x \mid y \mid a \_ b
\]

yields an output relation with four pairs:

\[
a b a b a , a b a b a \ a x b a x b a \ a x b a y b a \\
a b a b a , a b a b a \ a y b a x b a \ a y b a y b a
\]

This treatment of epsilon rules brings up an important issue of interpretation. Our definition of conditional replacement associates each application with a pair of context brackets. Because there is only one pair of >x brackets available at any application site, our rule allows only single replacements. This is why in the
previous example we get just one \( x \) or \( y \) at each insertion site rather than an infinite sequence of \( x \)s and \( y \)s.

It can be argued that the latter result would be more consistent from a general point of view, but we feel that it would be much less practical. If the infinite regression of insertions is needed, it can easily be obtained by using a Kleene star in \( \text{LOWER} \):

\[
[ ] \rightarrow [x \mid y]^+ || a \_ b ;
\]

2.4.4. Generalized definition of Replace

It is actually possible to give a single regular expression that has the same effect in the replacement step as the original definition in [46], repeated below

(5) Replace

\[
%< [\text{UPPER}]/[%<|>%] %>
\]

\[
%> %< \rightarrow %< \text{LOWER} %>
\]

in cases where \( \text{UPPER} \) is not an epsilon fsm and the same effect as

\[
%> %< \rightarrow %< \text{LOWER} %>
\]

when it is. This general replacement formula is

\[
[ %< [\text{UPPER}]/[%<|>%] %>] | [ [\text{UPPER} - ?+] [\%> %<] ] \rightarrow %< \text{LOWER} %>
\]

This expression describes the unconditional replacement of \( <\text{UPPER}> \) (ignoring any internal brackets) by \( <\text{LOWER}> \), unioned with the unconditional replacement of \( > < \) by \( <\text{LOWER}> \) whenever \( \text{UPPER} \) includes the empty string.

Note that formula to the left of the arrow is a disjunction whose right side is the concatenation of \([\text{UPPER} - ?+]\) with \([\%> %<]\). This concatenation produces one of two results. If \( \text{UPPER} \) includes the empty string, the result is the sequence \( >> \) because \([\text{UPPER} - ?+]\) is the empty string language. If \( \text{UPPER} \) does not include the empty string, the result is the null set. Consequently the effect of the general formula is to add \( >> \) to the replacement target if and only if \( \text{UPPER} \) includes the empty string. This logic can of course be implemented in a more efficient way by the regular expression compiler.

3. Comparisons

3.1. Phonological rewrite rules

Our definition of replacement is in its technical aspects very closely related to the way phonological rewrite-rules are defined in Kaplan and Kay 1994 but there are important differences. The initial motivation in their original 1981 presentation was to model a left-to-right deterministic process of rule application. In the course of exploring the issues, Kaplan and Kay developed a more abstract notion of rewrite rules that allowed the constraints on the operation to refer to both input and output strings. They invented the technique of encoding the context conditions with the help of auxiliary markers. We used the same method in the two-level rule compiler (Karttunen, Kaplan, and Koskenniemi 1987). Although the final 1994 version of the Kaplan and Kay paper lays out a sophisticated general framework, among other things, re-defining the compilation algorithm for two-level rules, it remains locked on the initial target of modeling a deterministic procedural application of rewrite rules.

The definition of replacement that we give in this paper has a very different starting point. The basic case for us is unconditional obligatory replacement, which we define in a purely relational way without any consideration of how replacement might be applied. The \( \rightarrow \) operator is like any other operator in the calculus of regular expressions in that it is defined in terms of other expressions. By starting with obligatory replacement, we can easily also define an optional version of the operator. For Kaplan and Kay, the primary notion is optional rewriting. It is more cumbersome for them to provide an obligatory version. More auxiliary brackets are needed. Another, less important, difference is in the treatment of epsilon replacements.

In spite of the very different motivation, both systems give the same result in a great number of cases. But sometimes they diverge, in particular with respect to right- and left-oriented replacement, which Kaplan and Kay especially focused on. It appears that in the upward-oriented case there is no difference; the downward-oriented version is not mentioned in their paper. We have not yet explored the technical differences between their and our definition in sufficient detail. At this point we assume that
the differences that may emerge arise from procedural considerations.

3.2. Two-level rules

Our definition of replacement also has a close connection to two-level rules. A two-level rule always specifies whether a context element belongs to the input (= lexical) or the output (= surface) context of the rule. The two-level model also shares our pure relational view of replacement as it is not concerned about the application procedure.

In one respect the two-level model is more expressive than our extended regular expression calculus. In the two-level formalism it is possible to refer to both the lexical and the surface context on the same side, as in

\[ a:x \leftrightarrow :a \ b: \_ \ ; \]

This rule says that the \( a:x \) correspondence is obligatory and allowed if and only if \( a:x \) is next to a surface \( a \) (regardless of what is on the lexical side) followed by a lexical \( b \) (not considering what became of it on the surface side). Our definition does not allow this case because each context constraint applies either before or after the replacement but not both.

This feature of the two-level formalism comes from the fact that all symbols are treated as atomic in the two-level regular expression calculus. The move from “feasible pairs” to real fst pairs happens only after the rules have been compiled to simple automata. Because 0s in the two-level calculus are not epsilons but ordinary symbols, the rule writer must carefully keep in mind all possible deletion and insertion sites so that pairs with one-sided zeros, such as \( x:0 \) and \( 0:y \), are included in all relevant contexts. Because two-level rules (in the \( \leftrightarrow \) direction) are limited to single-symbol correspondences, it is very cumbersome to express multisegment replacements.

The present formalism can be extended to constrain replacements with respect upper and lower contexts on the same side of the replacement site. One solution is to allow for several context specifications with alternate orientation. For example,

\[ a \rightarrow x \ // \ a \ (?) \_ \ || \ b \_ \ ; \]

would have a similar effect as the two-level rule above. It constrains the replacement with a double condition on the left of the replacement, requiring an \( a \) on the lower side, possibly followed by some unspecified symbol, and \( a \ b \) on the upper side.

4. Conclusion

The goal of this paper has been to introduce to the calculus of regular expressions a replace operator, \( \rightarrow \), with a set of associated replacement expressions that concisely encode alternate variations of the operation.

We defined unconditional and conditional replacement, taking the unconditional obligatory replacement as the basic case. We provide a simple declarative definition for it, easily expressed in terms of the other regular expression operators, and extend it to the conditional case providing four ways to constrain replacement by a context.

These definitions have already been implemented. The figures in this paper correspond exactly to the output of the regular expression compiler in the Xerox finite-state calculus.

Acknowledgments

This work is based on many years of productive collaboration with Ronald M. Kaplan and Martin Kay, my colleagues at PARC. I am particularly indebted to Ronald Kaplan for a constructive critique of the initial draft of the paper. Special thanks are also due to Kenneth R. Beesley for technical help on the definitions of the replace operators and for expert editorial advice. I am grateful to Pasi Tapanainen, Jean-Pierre Chanod and Annie Zaenen for helping to correct many terminological and rhetorical weaknesses in earlier versions of the paper.

References

Kaplan, Ronald M., and Kay, Martin (1981). Phonological Rules and Finite-State Transducers. Paper presented at the Annual Meeting of the Linguistic Society of America. New York.

Kaplan, Ronald M. and Kay, Martin (1994). Regular Models of Phonological Rule Systems. Computational Linguistics. 20:3 331-378. 1994.

Karlsson, Fred, Voutilainen, Atro, Heikkilä, Juha, and Anttila, Arto (1994) Constraint Grammar: a Language-Independent System for
*Parsing Unrestricted Text.* Mouton de Gruyter, Berlin, 1994.

Karttunen, Lauri, Kaplan, Ronald M., Zaenen, Annie (1992). Two-level Morphology with Composition. In the *Proceedings of Coling-92.* Nantes, France.

Karttunen, Lauri, Koskenniemi, Kimmo, and Kaplan, Ronald M. (1987) A Compiler for Two-level Phonological Rules. In Report No. CSLI-87-108. Center for the Study of Language and Information. Stanford University.

Karttunen, Lauri and Beesley, Kenneth R. (1992). *Two-level Rule Compiler.* Technical Report. ISTL-92-2. Xerox Palo Alto Research Center.

Karttunen, Lauri (1994). Constructing Lexical Transducers. In the *Proceedings of Coling-94.* Kyoto, Japan.

Koskenniemi, Kimmo (1983). *Two-level Morphology: A General Computational Model for Word-Form Recognition and Production.* Department of General Linguistics. University of Helsinki.

Koskenniemi, Kimmo (1990). Finite-State Parsing and Disambiguation. In the Proceedings of Coling-90. Helsinki, Finland.

Koskenniemi, Kimmo, Tapanainen, Pasi, and Voutilainen, Atro (1992). Compiling and using finite-state syntactic rules. In the *Proceedings of Coling-92.* Nantes, France.

Prince, Alan and Smolensky, Paul (1993). *Optimality Theory: constraint interaction in generative grammar.* To appear, MIT Press. TR-2, Rutgers University Cognitive Science Center.

Voutilainen, Atro (1994) *Three Studies of Grammar-Based Surface Parsing of Unrestricted English Text.* The University of Helsinki. 1994.