Directed Replacement

Lauri Karttunen
Rank Xerox Research Centre Grenoble
6, chemin de Maupertuis
F-38240 MEYLAN, FRANCE
lauri.karttunen@xerox.fr

Abstract

This paper introduces to the finite-state calculus a family of directed replace operators. In contrast to the simple replace expression, `UPPER -> LOWER`, defined in Karttunen (1995), the new directed version, `UPPER @-> LOWER`, yields an unambiguous transducer if the lower language consists of a single string. It transduces the input string from left to right, making only the longest possible replacement at each point.

A new type of replacement expression, `UPPER @-> PREFIX ... SUFFIX`, yields a transducer that inserts text around strings that are instances of `UPPER`. The symbol ... denotes the matching part of the input which itself remains unchanged. `PREFIX` and `SUFFIX` are regular expressions describing the insertions.

Expressions of the type `UPPER @-> PREFIX ... SUFFIX` may be used to compose a deterministic parser for a "local grammar" in the sense of Gross (1989). Other useful applications of directed replacement include tokenization and filtering of text streams.

1 Introduction

Transducers compiled from simple replace expressions `UPPER -> LOWER` (Karttunen 1995, Kempe and Karttunen 1996) are generally nondeterministic in the sense that they may yield multiple results even if the lower language consists of a single string. For example, let us consider the transducer in Figure ??, representing `a b | b | b a | a b a -> x`.[1]

[1]The regular expression formalism and other notational conventions used in the paper are explained in the Appendix at the end.

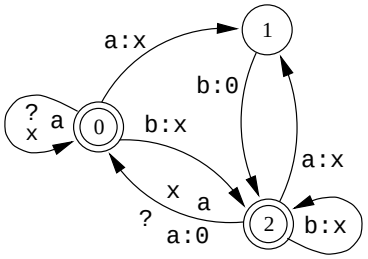

Figure 1: `a b | b | b a | a b a -> x` . The four paths with "aba" on the upper side are: <0 `a` 0 `b:x` 2 `a` 0>, <0 `a` 0 `b:x` 2 `a:0` 0>, <0 `a:x` 1 `b:0` 2 `a` 0>, and <0 `a:x` 1 `b:0` 2 `a:0` 0>.

The application of this transducer to the input "aba" produces four alternate results, "axa", "ax", "xa", and "x", as shown in Figure ??, since there are four paths in the network that contain "aba" on the upper side with different strings on the lower side.

This nondeterminism arises in two ways. First of all, a replacement can start at any point. Thus we get different results for "aba" depending on whether we start at the beginning of the string or in the middle at the "b". Secondly, there may be alternative replacements with the same starting point. In the beginning of "aba", we can replace either "ab" or "aba". Starting in the middle, we can replace either "b" or "ba". The underlining in Figure ?? shows

```
a b a    a b a    a b a    a b a
  -        ---      ---      -----
a x a      a  x       x  a       x
```

Figure 2: Four factorizations of "aba".

the four alternate factorizations of the input string, that is, the four alternate ways to partition the string "aba" with respect to the upper language of the replacement expression. The corresponding paths in the transducer are listed in Figure ??.

For many applications, it is useful to define an-

other version of replacement that produces a unique outcome whenever the lower language of the relation consists of a single string. To limit the number of alternative results to one in such cases, we must impose a unique factorization on every input.

The desired effect can be obtained by constraining the directionality and the length of the replacement. Directionality means that the replacement sites in the input string are selected starting from the left or from the right, not allowing any overlaps. The length constraint forces us always to choose the longest or the shortest replacement whenever there are multiple candidate strings starting at a given location. We use the term **directed replacement** to describe a replacement relation that is constrained by directionality and length of match. (See the end of Section 2 for a discussion about the choice of the term.)

With these two kinds of constraints we can define four types of directed replacement, listed in Figure ??.

	longest match	shortest match
left-to-right	`@->`	`@>`
right-to-left	`->@`	`>@`

Figure 3: Directed replacement operators

For reasons of space, we discuss here only the left-to-right, longest-match version. The other cases are similar.

The effect of the directionality and length constraints is that some possible replacements are ignored. For example, `a b | b | b a | a b a @-> x` maps "aba" uniquely into "x", Figure ??.

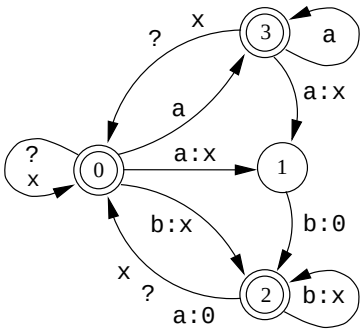

Figure 4: `a b | b | b a | a b a @-> x`. The single path with "aba" on the upper side is: <0 `a:x` 1 `b:0` 2 `a:0` 0>.

Because we must start from the left and have to choose the longest match, "aba" must be replaced, ignoring the possible replacements for "b", "ba", and "ab". The `@->` operator allows only the last factorization of "aba" in Figure ??.

Left-to-right, longest-match replacement can be thought of as a procedure that rewrites an input string sequentially from left to right. It copies the input until it finds an instance of `UPPER`. At that point it selects the longest matching substring, which is rewritten as `LOWER`, and proceeds from the end of that substring without considering any other alternatives. Figure ?? illustrates the idea.

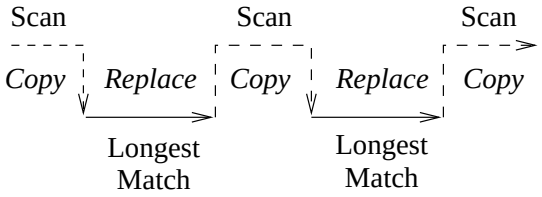

Figure 5: Left-to-right, longest-match replacement

It is not obvious at the outset that the operation can in fact be encoded as a finite-state transducer for arbitrary regular patterns. Although a unique substring is selected for replacement at each point, in general the transduction is not unambiguous because `LOWER` is not required to be a single string; it can be any regular language.

The idea of treating phonological rewrite rules in this way was the starting point of Kaplan and Kay (1994). Their notion of obligatory rewrite rule incorporates a directionality constraint. They observe (p. 358), however, that this constraint does not by itself guarantee a single output. Kaplan and Kay suggest that additional restrictions, such as longest-match, could be imposed to further constrain rule application.[2] We consider this issue in more detail.

The crucial observation is that the two constraints, left-to-right and longest-match, force a unique factorization on the input string thus making the transduction unambiguous if the `LOWER` language consists of a single string. In effect, the input string is unambiguously **parsed** with respect to the `UPPER` language. This property turns out to be important for a number of applications. Thus it is useful to provide a replacement operator that implements these constraints directly.

The definition of the `UPPER @-> LOWER` relation is presented in the next section. Section 3 introduces a novel type of replace expression for constructing transducers that unambiguously recognize and mark

[2]The tentative formulation of the longest-match constraint in (?, p. 358) is too weak. It does not cover all the cases.

instances of a regular language without actually replacing them. Section 4 identifies some useful applications of the new replacement expressions.

2 Directed Replacement

We define directed replacement by means of a composition of regular relations. As in Kaplan and Kay (1994), Karttunen (1995), and other previous works on related topics, the intermediate levels of the composition introduce auxiliary symbols to express and enforce constraints on the replacement relation. Figure ?? shows the component relations and how they are composed with the input.

```
Input string
    .o.
Initial match
    .o.
Left-to-right constraint
    .o.
Longest-match constraint
    .o.
Replacement
```

Figure 6: Composition of directed replacement

If the four relations on the bottom of Figure ?? are composed in advance, as our compiler does, the application of the replacement to an input string takes place in one step without any intervening levels and with no auxiliary symbols. But it helps to understand the logic to see where the auxiliary marks would be in the hypothetical intermediate results.

Let us consider the case of `a b | b | b a | a b a @-> x` applying to the string "aba" and see in detail how the mapping implemented by the transducer in Figure ?? is composed from the four component relations. We use three auxiliary symbols, caret (^), left bracket (<) and right bracket (>), assuming here that they do not occur in any input. The first step, shown in Figure ??, composes the input string with a transducer that inserts a caret, in the beginning of every substring that belongs to the upper language.

```
  a   b  a
^ a ^ b  a
```

Figure 7: Initial match. Each caret marks the beginning of a substring that matches "ab", "b", "ba", or "aba".

Note that only one ^ is inserted even if there are several candidate strings starting at the same location.

In the left-to-right step, we enclose in angle brackets all the substrings starting at a location marked by a caret that are instances of the upper language. The initial caret is replaced by a <, and a closing > is inserted to mark the end of the match. We permit carets to appear freely while matching. No carets are permitted outside the matched substrings and the ignored internal carets are eliminated. In this case, there are four possible outcomes, shown in Figure ??, but only two of them are allowed under the constraint that there can be no carets outside the brackets.

```
                ALLOWED

^ a ^ b   a     ^ a ^ b   a
< a   b > a     < a   b   a >

              NOT ALLOWED

^ a  ^ b   a    ^ a ^ b   a
^ a  < b > a    ^ a < b   a >
```

Figure 8: Left-to-right constraint. *No caret outside a bracketed region.*

In effect, no starting location for a replacement can be skipped over except in the context of another replacement starting further left in the input string. (Roche and Schabes (1995) introduce a similar technique for imposing the left-to-right order on the transduction.) Note that the four alternatives in Figure ?? represent the four factorizations in Figure ??.

The longest-match constraint is the identity relation on a certain set of strings. It forbids any replacement that starts at the same location as another, longer replacement. In the case at hand, it means that the internal > is disallowed in the context `< a b > a`. Because "aba" is in the upper language, there is a longer, and therefore preferred, `< a b a >` alternative at the same starting location, Figure ??.

```
    ALLOWED          NOT ALLOWED

< a   b   a >      < a   b > a
```

Figure 9: Longest match constraint. *No upper language string with an initial* < *and a nonfinal* > *in the middle.*

In the final replacement step, the bracketed regions of the input string, in the case at hand, just `< a b a >`, are replaced by the strings of the lower language, yielding "x" as the result for our example.

Note that longest match constraint ignores any internal brackets. For example, the bracketing `< a`

`> < a >` is not allowed if the upper language contains "aa" as well as "a". Similarly, the left-to-right constraint ignores any internal carets.

As the first step towards a formal definition of `UPPER @-> LOWER` it is useful to make the notion of "ignoring internal brackets" more precise. Figure ?? contains the auxiliary definitions. For the details of the formalism (briefly explained in the Appendix), please consult Karttunen (1995), Kempe and Karttunen (1996).[3]

```
UPPER'  = UPPER/[%^] - [?* %^]
UPPER'' = UPPER/[%<|%>] - [?* [%<|%>]]
```

Figure 10: Versions of `UPPER` that freely allow nonfinal diacritics.

The precise definition of the `UPPER @-> LOWER` relation is given in Figure ??. It is a composition of many auxiliary relations. We label the major components in accordance with the outline in Figure ??. The formulation of the longest-match constraint is based on a suggestion by Ronald M. Kaplan (p.c.).

Initial match

```
    ~$[ %^ | %< | %> ]
           .o.
  [. .] -> %^ || _ UPPER
           .o.
```

Left to right

```
[~$[%^] [%^:%< UPPER' 0:%>]]* ~$[%^]
           .o.
         %^ -> []
           .o.
```

Longest match

```
   ~$[%< [UPPER'' & $[%>]]]
           .o.
```

Replacement

```
    %< ~$[%>] %> -> LOWER  ;
```

Figure 11: Definition of `UPPER @-> LOWER`

The logic of `@->` replacement could be encoded in many other ways, for example, by using the three pairs of auxiliary brackets, `<i`, `>i`, `<c`, `>c`, and `<a`, `>a`, introduced in Kaplan and Kay (1994). We take here a more minimalist approach. One reason is that we prefer to think of the simple unconditional (uncontexted) replacement as the basic case, as in Karttunen (1995). Without the additional complexities introduced by contexts, the directionality and length-of-match constraints can be encoded with fewer diacritics. (We believe that the conditional case can also be handled in a simpler way than in Kaplan and Kay (1994).) The number of auxiliary markers is an important consideration for some of the applications discussed below.

In a phonological or morphological rewrite rule, the center part of the rule is typically very small: a modification, deletion or insertion of a single segment. On the other hand, in our text processing applications, the upper language may involve a large network representing, for example, a lexicon of multiword tokens. Practical experience shows that the presence of many auxiliary diacritics makes it difficult or impossible to compute the left-to-right and longest-match constraints in such cases. The size of intermediate states of the computation becomes a critical issue, while it is irrelevant for simple phonological rules. We will return to this issue in the discussion of tokenizing transducers in Section 4.

The transducers derived from the definition in Figure ?? have the property that they unambiguously parse the input string into a sequence of substrings that are either copied to the output unchanged or replaced by some other strings. However they do not fall neatly into any standard class of transducers discussed in the literature (Eilenberg 1974, Schützenberger 1977, Berstel 1979). If the `LOWER` language consists of a single string, then the relation encoded by the transducer is in Berstel's terms a **rational function**, and the network is an **unambigous** transducer, even though it may contain states with outgoing transitions to two or more destinations for the same input symbol. An unambiguous transducer may also be **sequentiable**, in which case it can be turned into an equivalent **sequential** transducer (?), which can in turn be minimized. A transducer is sequential just in case there are no states with more than one transition for the same input symbol. Roche and Schabes (1995) call such transducers **deterministic**.

Our replacement transducers in general are not unambiguous because we allow `LOWER` to be any regular language. It may well turn out that, in all cases that are of practical interest, the lower language is in fact a singleton, or at least some finite set, but it is not so by definition. Even if the replacement transducer is unambiguous, it may well be unsequentiable if `UPPER` is an infinite language. For example, the simple transducer for `a+ b @-> x` in Figure ?? cannot be sequentialized. It has to replace any string of "a"s by "x" or copy it to the output unchanged depending on whether the string eventually terminates at "b". It is obviously impossible for any finite-state

[3] `UPPER'` is the same language as `UPPER` except that carets may appear freely in all nonfinal positions. Similarly, `UPPER''` accepts any nonfinal brackets.

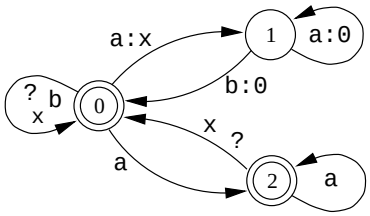

Figure 12: `a+ b @-> x`. This transducer is unambiguous but cannot be sequentialized.

device to accumulate an unbounded amount of delayed output. On the other hand, the transducer in Figure ?? is sequentiable because there the choice between `a` and `a:x` just depends on the next input symbol.

Because none of the classical terms fits exactly, we have chosen a novel term, **directed transduction**, to describe a relation induced by the definition in Figure ??. It is meant to suggest that the mapping from the input into the output strings is guided by the directionality and length-of-match constraints. Depending on the characteristics of the `UPPER` and `LOWER` languages, the resulting transducers may be unambiguous and even sequential, but that is not guaranteed in the general case.

3 Insertion

The effect of the left-to-right and longest-match constraint is to factor any input string uniquely with respect to the upper language of the replace expression, to parse it into a sequence of substrings that either belong or do not belong to the language. Instead of replacing the instances of the upper language in the input by other strings, we can also take advantage of the unique factorization in other ways. For example, we may insert a string before and after each substring that is an instance of the language in question simply to mark it as such.

To implement this idea, we introduce the special symbol ... on the right-hand side of the replacement expression to mark the place around which the insertions are to be made. Thus we allow replacement expressions of the form `UPPER @-> PREFIX ... SUFFIX`. The corresponding transducer locates the instances of `UPPER` in the input string under the left-to-right, longest-match regimen just described. But instead of replacing the matched strings, the transducer just copies them, inserting the specified prefix and suffix. For the sake of generality, we allow `PREFIX` and `SUFFIX` to denote any regular language.

The definition of `UPPER @-> PREFIX ... SUFFIX` is just as in Figure ?? except that the Replacement expression is replaced by the Insertion formula in Figure ??, a simple parallel replacement of the two auxiliary brackets that mark the selected regions. Because the placement of `<` and `>` is strictly controlled, they do not occur anywhere else.

Insertion

```
%< -> PREFIX, %> -> SUFFIX ;
```

Figure 13: Insertion expression in the definition of `UPPER @-> PREFIX ... SUFFIX`.

With the ... expressions we can construct transducers that mark maximal instances of a regular language. For example, let us assume that noun phrases consist of an optional determiner, `(d)`, any number of adjectives, `a*`, and one or more nouns, `n+`. The expression `(d) a* n+ @-> %[ ... %]` compiles into a transducer that inserts brackets around maximal instances of the noun phrase pattern. For example, it maps `"dannvaan"` into `"[dann]v[aan]"`, as shown in Figure ??.

```
  d a n n   v   a a n
  -------       -----
[ d a n n ] v [ a a n ]
```

Figure 14: Application of `(d) a* n+ @-> %[...%]` to `"dannvaan"`

Although the input string `"dannvaan"` contains many other instances of the noun phrase pattern, `"n"`, `"an"`, `"nn"`, etc., the left-to-right and longest-match constraints pick out just the two maximal ones. The transducer is displayed in Figure ??. Note that `?` here matches symbols, such as `v`, that are not included in the alphabet of the network.

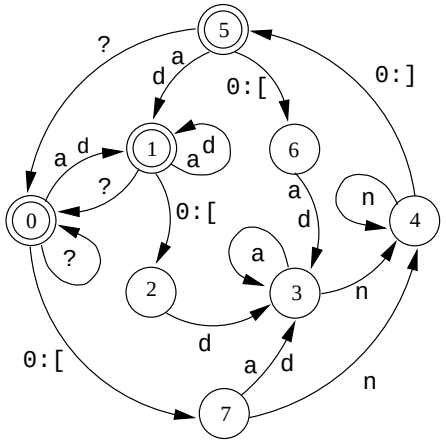

Figure 15: `(d) a* n+ @-> %[...%]`. The one path with "dannvaan" on the upper side is: <0 0:[7 d 3 a 3 n 4 n 4 0:] 5 v 0 0:[7 a 3 a 3 n 4 0:] 5>.

4 Applications

The directed replacement operators have many useful applications. We describe some of them. Although the same results could often be achieved by using lex and yacc, sed, awk, perl, and other Unix utilities, there is an advantage in using finite-state transducers for these tasks because they can then be smoothly integrated with other finite-state processes, such as morphological analysis by lexical transducers (Karttunen 1994) and rule-based part-of-speech disambiguation (Chanod and Tapanainen 1995, Roche and Schabes 1995).

4.1 Tokenization

A tokenizer is a device that segments an input string into a sequence of tokens. The insertion of end-of-token marks can be accomplished by a finite-state transducer that is compiled from tokenization rules. The tokenization rules may be of several types. For example, `[WHITE%_SPACE+ @-> SPACE]` is a normalizing transducer that reduces any sequence of tabs, spaces, and newlines to a single space. `[LETTER+ @-> ... END%_OF%_TOKEN]` inserts a special mark, e.g. a newline, at the end of a letter sequence.

Although a space generally counts as a token boundary, it can also be part of a multiword token, as in expressions like "at least", "head over heels", "in spite of", etc. Thus the rule that introduces the `END_OF_TOKEN` symbol needs to combine the `LETTER+` pattern with a list of multiword tokens which may include spaces, periods and other delimiters.

Figure ?? outlines the construction of a simple tokenizing transducer for English.

```
WHITE%_SPACE+ @-> SPACE
        .o.
[ LETTER+ |
  a t %  l e a s t |
  h e a d %  o v e r %  h e e l s |
  i n %  s p i t e %  o f ]
 @-> ... END%_OF%_TOKEN
        .o.
SPACE -> [] || .#. | END%_OF%_TOKEN _ ;
```

Figure 16: A simple tokenizer

The tokenizer in Figure ?? is composed of three transducers. The first reduces strings of whitespace characters to a single space. The second transducer inserts an `END_OF_TOKEN` mark after simple words and the listed multiword expressions. The third removes the spaces that are not part of some multiword token. The percent sign here means that the following blank is to be taken literally, that is, parsed as a symbol.

Without the left-to-right, longest-match constraints, the tokenizing transducer would not produce deterministic output. Note that it must introduce an `END_OF_TOKEN` mark after a sequence of letters just in case the word is not part of some longer multiword token. This problem is complicated by the fact that the list of multiword tokens may contain overlapping expressions. A tokenizer for French, for example, needs to recognize "de plus" (moreover), "en plus" (more), "en plus de" (in addition to), and "de plus en plus" (more and more) as single tokens. Thus there is a token boundary after "de plus" in *de plus on ne le fait plus* (moreover one doesn't do it anymore) but not in *on le fait de plus en plus* (one does it more and more) where "de plus en plus" is a single token.

If the list of multiword tokens contains hundreds of expressions, it may require a lot of time and space to compile the tokenizer even if the final result is not too large. The number of auxiliary symbols used to encode the constraints has a critical effect on the efficiency of that computation. We first observed this phenomenon in the course of building a tokenizer for the British National Corpus according to the specifications of the BNC Users Guide (?), which lists around 300 multiword tokens and 260 foreign phrases. With the current definition of the directed replacement we have now been able to compute similar tokenizers for several other languages (French, Spanish, Italian, Portuguese, Dutch, German).

4.2 Filtering

Some text processing applications involve a preliminary stage in which the input stream is divided into regions that are passed on to the calling process and regions that are ignored. For example, in processing an SGML-coded document, we may wish to delete all the material that appears or does not appear in a region bounded by certain SGML tags, say `<A>` and `</A>`.

Both types of filters can easily be constructed using the directed replace operator. A negative filter that deletes all the material between the two SGML codes, including the codes themselves, is expressed as in Figure ??.

```
"<A>" ~$["<A>"|"</A>"] "</A>" @-> [] ;
```

Figure 17: A negative filter

A positive filter that excludes everything else can be expressed as in Figure ??.

```
~$"</A>" "<A>" @-> "<A>"
          .o.
"</A>" ~$"<A>" @-> "</A>" ;
```

Figure 18: A positive filter

The positive filter is composed of two transducers. The first reduces to <A> any string that ends with it and does not contain the </A> tag. The second transducer does a similar transduction on strings that begin with </A>. Figure 12 illustrates the effect of the positive filter.

```
<B>one</B><A>two</A><C>three</C><A>four</A>
-------------   ----------------
     <A>      two       </A>       <A>four</A>
```

Figure 19: Application of a positive filter

The idea of filtering by finite-state transduction of course does not depend on SGML codes. It can be applied to texts where the interesting and uninteresting regions are defined by any kind of regular pattern.

4.3 Marking

As we observed in section 3, by using the ... symbol on the lower side of the replacement expression, we can construct transducers that mark instances of a regular language without changing the text in any other way. Such transducers have a wide range of applications. They can be used to locate all kinds of expressions that can be described by a regular pattern, such as proper names, dates, addresses, social security and phone numbers, and the like. Such a marking transducer can be viewed as a deterministic parser for a "local grammar" in the sense of Gross (1989), Roche (1993), Silberztein (1993) and others.

By composing two or more marking transducers, we can also construct a single transducer that builds nested syntactic structures, up to any desired depth. To make the construction simpler, we can start by defining auxiliary symbols for the basic regular patterns. For example, we may define NP as [(d) a* n+]. With that abbreviatory convention, a composition of a simple NP and VP spotter can be defined as in Figure ??.

```
NP @-> %[NP ... %]
      .o.
v %[NP NP %]  @-> %[VP ... %] ;
```

Figure 20: Composition of an NP and a VP spotter

Figure ?? shows the effect of applying this composite transducer to the string "dannvaan".

```
d a n n       v     a a n
-------       -     -----
[NP d a n n ] [VP v [NP a a n ] ]
```

Figure 21: Application of an NP-VP parser

By means of this simple "bottom-up" technique, it is possible to compile finite-state transducers that approximate a context-free parser up to a chosen depth of embedding. Of course, the left-to-right, longest-match regimen implies that some possible analyses are ignored. To produce all possible parses, we may introduce the ... notation to the simple replace expressions in Karttunen (1995).

5 Extensions

The definition of the left-to-right, longest-match replacement can easily be modified for the three other directed replace operators mentioned in Figure ??. Another extension, already implemented, is a directed version of parallel replacement (Kempe and Karttunen 1996), which allows any number of replacements to be done simultaneously without interfering with each other. Figure ?? is an example of a directed parallel replacement. It yields a transducer that maps a string of "a"s into a single "b" and a string of "b"s into a single "a".

```
a+ @-> b, b+ @-> a ;
```

Figure 22: Directed, parallel replacement

The definition of directed parallel replacement requires no additions to the techniques already presented. In the near future we also plan to allow directional and length-of-match constraints in the more complicated case of conditional context-constrained replacement.

6 Acknowledgements

I would like to thank Ronald M. Kaplan, Martin Kay, André Kempe, John Maxwell, and Annie Zaenen for helpful discussions at the beginning of the project, as well as Paula Newman and Kenneth R. Beesley for editorial advice on the first draft of the paper. The work on tokenizers and phrasal analyzers by Anne Schiller and Gregory Grefenstette revealed the need for a more efficient implementation of the idea. The final version of the paper has benefited from detailed comments by Ronald M. Kaplan and two anonymous reviewers, who convinced me to discard the ill-chosen original title ("Deterministic Replacement") in favor of the present one.

7 Appendix: Notational conventions

The regular expression formalism used in this paper is essentially the same as in Kaplan and Kay (1994), in Karttunen (1995), and in Kempe and Karttunen (1996). Upper-case strings, such as `UPPER`, represent regular languages, and lower-case letters, such as `x`, represent symbols. We recognize two types of symbols: unary symbols (`a`, `b`, `c`, etc) and symbol pairs (`a:x`, `b:0`, etc.).

A symbol pair `a:x` may be thought of as the crossproduct of `a` and `x`, the minimal relation consisting of `a` (the upper symbol) and `x` (the lower symbol). 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` into itself. Consequently, we also write `a:a` as just `a`.

Three special symbols are used in regular expressions: `0` (zero) represents the empty string (often denoted by ϵ); `?` stands for any symbol in the known alphabet and its extensions; in replacement expressions, `.#.` marks the start (left context) or the end (right context) of a string. The percent sign, `%`, is used as an escape character. It allows letters that have a special meaning in the calculus to be used as ordinary symbols. Thus `%[` denotes the literal square bracket as opposed to `[`, which has a special meaning as a grouping symbol; %0 is the ordinary zero symbol. Double quotes around a symbol have the same effect as the percent sign.

The following simple expressions appear freqently in the formulas: `[]` the empty string language, `?*` the universal ("sigma star") language.

The regular expression operators used in the paper are: `*` zero or more (Kleene star), `+` one or more (Kleene plus), `~` not (complement), `$` contains, `/` ignore, `|` or (union), `&` and (intersection), `-` minus (relative complement), `.x.` crossproduct, `.o.` composition, `->` simple replace.

In the transducer diagrams (Figures ??, ??, etc.), the nonfinal states are represented by single circles, final states by double circles. State 0 is the initial state. The symbol `?` represents any symbols that are not explicitly present in the network. Transitions that differ only with respect to the label are collapsed into a single multiply labelled arc.

References

Jean Berstel. 1979. *Transductions and Context-Free Languages.* B.G. Teubner, Stuttgart, Germany.

Jean-Pierre Chanod and Pasi Tapanainen. 1995. Tagging French—comparing a statistical and a constraint-based model. In *The Proceedings of the Seventh Conference of the European Chapter of the Association for Computational Linguistics*, Dublin, Ireland.

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

Maurice Gross. 1989. The Use of Finite Automata in the Lexical Representation of Natural Language. In *Lecture Notes in Computer Science*, pages 34–50, Springer-Verlag, Berlin, Germany.

Ronald M. Kaplan and Martin Kay. 1994. Regular Models of Phonological Rule Systems. *Computational Linguistics*, 20:3, pages 331–378.

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

Lauri Karttunen. 1994. Constructing Lexical Transducers. In *The Proceedings of the Fifteenth International Conference on Computational Linguistics. Coling 94*, I, pages 406–411, Kyoto, Japan.

Lauri Karttunen. 1995. The Replace Operator. In *The Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics. ACL-95*, pages 16–23, Boston, Massachusetts.

André Kempe and Lauri Karttunen. 1996. Parallel Replacement in the Finite-State Calculus. In *The Proceedings of the Sixteenth International Conference on Computational Linguistics. Coling 96.* Copenhagen, Denmark.

Geoffrey Leech. 1995. *User's Guide to the British National Corpus.* Lancaster University.

Mehryar Mohri. 1994. *On Some Applications of Finite-State Automata Theory to Natural Language Processing.* Technical Report 94-22. L'Institute Gaspard Monge. Université de Marne-la-Vallée. Noisy Le Grand.

Emmanuel Roche. 1993. *Analyse syntaxique transformationelle du français par transducteurs et lexique-grammaire.* Doctoral dissertation, Université Paris 7.

Emmanuel Roche and Yves Schabes. 1995. Deterministic Part-of-Speech Tagging. *Computational Linguistics*, 21:2, pages 227–53.

Marcel Paul Schützenberger. 1977. Sur une variante des fonctions sequentielles. *Theoretical Computer Science*, 4, pages 47–57.

Max Silberztein. 1993. *Dictionnaires Electroniques et Analyse Lexicale du Français—Le Système INTEX*, Masson, Paris, France.