Transducers from Rewrite Rules with Backreferences

Dale Gerdemann
University of Tuebingen
Kl. Wilhelmstr. 113
D-72074 Tuebingen
dg@sfs.nphil.uni-tuebingen.de

Gertjan van Noord
Groningen University
PO Box 716
NL 9700 AS Groningen
vannoord@let.rug.nl

Abstract

Context sensitive rewrite rules have been widely used in several areas of natural language processing, including syntax, morphology, phonology and speech processing. Kaplan and Kay, Karttunen, and Mohri & Sproat have given various algorithms to compile such rewrite rules into finite-state transducers. The present paper extends this work by allowing a limited form of backreferencing in such rules. The explicit use of backreferencing leads to more elegant and general solutions.

1 Introduction

Context sensitive rewrite rules have been widely used in several areas of natural language processing. Johnson (1972) has shown that such rewrite rules are equivalent to finite state transducers in the special case that they are not allowed to rewrite their own output. An algorithm for compilation into transducers was provided by Kaplan and Kay (1994). Improvements and extensions to this algorithm have been provided by Karttunen (1995), Karttunen (1997), Karttunen (1996) and Mohri and Sproat (1996). In this paper, the algorithm will be extended to provide a limited form of backreferencing. Backreferencing has been implicit in previous research, such as in the "batch rules" of Kaplan and Kay (1994), bracketing transducers for finite-state parsing (Karttunen, 1996), and the "LocalExtension" operation of Roche and Schabes (1995). The explicit use of backreferencing leads to more elegant and general solutions.

Backreferencing is widely used in editors, scripting languages and other tools employing regular expressions (Friedl, 1997). For example, Emacs uses the special brackets `\(` and `\)` to capture strings along with the notation $\backslash n$ to recall the nth such string. The expression **\(a*\)b\1** matches strings of the form $a^n b a^n$. Unrestricted use of backreferencing thus can introduce non-regular languages. For NLP finite state calculi (Karttunen et al., 1996; van Noord, 1997) this is unacceptable. The form of backreferences introduced in this paper will therefore be restricted.

The central case of an allowable backreference is:

$$x \Rightarrow T(x)/\lambda_\rho \tag{1}$$

This says that each string x preceded by λ and followed by ρ is replaced by $T(x)$, where λ and ρ are arbitrary regular expressions, and T is a transducer.[1] This contrasts sharply with the rewriting rules that follow the tradition of Kaplan & Kay:

$$\phi \Rightarrow \psi/\lambda__\rho \tag{2}$$

In this case, any string from the language ϕ is replaced by any string independently chosen from the language ψ.

We also allow multiple (non-permuting) backreferences of the form:

[1] The syntax at this point is merely suggestive. As an example, suppose that T_{acr} transduces phrases into acronyms. Then

$$x \Rightarrow T_{acr}(x)/\langle\text{abbr}\rangle_\langle/\text{abbr}\rangle$$

would transduce `<abbr>non-deterministic finite automaton</abbr>` into `<abbr>NDFA</abbr>`.

To compare this with a backreference in Perl, suppose that T_{acr} is a subroutine that converts phrases into acronyms and that R_{acr} is a regular expression matching phrases that can be converted into acronyms. Then (ignoring the left context) one can write something like: s/(R_{acr})(?=⟨/ABBR⟩)/T_{acr}($1)/ge;. The backreference variable, $1, will be set to whatever string R_{acr} matches.

$$x_1 x_2 \ldots x_n \Rightarrow T_1(x_1) T_2(x_2) \ldots T_n(x_n) / \lambda_\rho \quad (3)$$

Since transducers are closed under concatenation, handling multiple backreferences reduces to the problem of handling a single backreference:

$$x \Rightarrow (T_1 \cdot T_2 \cdot \ldots \cdot T_n)(x) / \lambda_\rho \quad (4)$$

A problem arises if we want capturing to follow the POSIX standard requiring a longest-capture strategy. Friedl (1997) (p. 117), for example, discusses matching the regular expression (to|top)(o|polo)?(gical|o?logical) against the word: **topological**. The desired result is that (once an overall match is established) the first set of parentheses should capture the longest string possible (**top**); the second set should then match the longest string possible from what's left (**o**), and so on. Such a left-most longest match concatenation operation is described in §3.

In the following section, we initially concentrate on the simple case in (1) and show how (1) may be compiled assuming left-to-right processing along with the overall longest match strategy described by Karttunen (1996).

The major components of the algorithm are not new, but straightforward modifications of components presented in Karttunen (1996) and Mohri and Sproat (1996). We improve upon existing approaches because we solve a problem concerning the use of special marker symbols (§2.1.2). A further contribution is that all steps are implemented in a freely available system, the FSA Utilities of van Noord (1997) (§2.1.1).

2 The Algorithm

2.1 Preliminary Considerations

Before presenting the algorithm proper, we will deal with a couple of meta issues. First, we introduce our version of the finite state calculus in §2.1.1. The treatment of special marker symbols is discussed in §2.1.2. Then in §2.1.3, we discuss various utilities that will be essential for the algorithm.

2.1.1 FSA Utilities

The algorithm is implemented in the FSA Utilities (van Noord, 1997). We use the notation provided by the toolbox throughout this paper. Table 1 lists the relevant regular expression operators. FSA Utilities offers the possibility to define new regular expression operators. For example, consider the definition of the nullary operator `vowel` as the union of the five vowels:

`[]`	empty string
`[E1,...En]`	concatenation of `E1 ...En`
`{}`	empty language
`{E1,...En}`	union of `E1,...En`
`E*`	Kleene closure
`E^`	optionality
`~E`	complement
`E1-E2`	difference
`$ E`	containment
`E1 & E2`	intersection
`?`	any symbol
`A:B`	pair
`E1 x E2`	cross-product
`A o B`	composition
`domain(E)`	domain of a transduction
`range(E)`	range of a transduction
`identity(E)`	identity transduction
`inverse(E)`	inverse transduction

Table 1: Regular expression operators.

```
macro(vowel,{a,e,i,o,u}).
```

In such macro definitions, Prolog variables can be used in order to define new n-ary regular expression operators in terms of existing operators. For instance, the *lenient_composition* operator (Karttunen, 1998) is defined by:

```
macro(priority_union(Q,R),
      {Q, ~domain(Q) o R}).
macro(lenient_composition(R,C),
      priority_union(R o C,R)).
```

Here, `priority_union` of two regular expressions `Q` and `R` is defined as the union of `Q` and the composition of the complement of the domain of `Q` with `R`. Lenient composition of `R` and `C` is defined as the priority union of the composition of `R` and `C` (on the one hand) and `R` (on the other hand).

Some operators, however, require something more than simple macro expansion for their definition. For example, suppose a user wanted to match n occurrences of some pattern. The FSA Utilities already has the '*' and '+' quantifiers, but any other operators like this need to be user defined. For this purpose, the FSA Utilities supplies simple Prolog hooks allowing this general quantifier to be defined as:

```
macro(match_n(N,X),Regex) :-
   match_n(N,X,Regex).

match_n(0,_X,[]).
match_n(N,X,[X|Rest]) :-
   N > 0,
   N1 is N-1,
   match_n(N1,X,Rest).
```

For example: `match_n(3,a)` is equivalent to the ordinary finite state calculus expression `[a,a,a]`.

Finally, regular expression operators can be defined in terms of operations on the underlying automaton. In such cases, Prolog hooks for manipulating states and transitions may be used. This functionality has been used in van Noord and Gerdemann (1999) to provide an implementation of the algorithm in Mohri and Sproat (1996).

2.1.2 Treatment of Markers

Previous algorithms for compiling rewrite rules into transducers have followed Kaplan and Kay (1994) by introducing special marker symbols (*markers*) into strings in order to mark off candidate regions for replacement. The assumption is that these markers are outside the resulting transducer's alphabets. But previous algorithms have not ensured that the assumption holds.

This problem was recognized by Karttunen (1996), whose algorithm starts with a filter transducer which filters out any string containing a marker. This is problematic for two reasons. First, when applied to a string that does happen to contain a marker, the algorithm will simply fail. Second, it leads to logical problems in the interpretation of complementation. Since the complement of a regular expression R is defined as $\Sigma - R$, one needs to know whether the marker symbols are in Σ or not. This has not been clearly addressed in previous literature.

We have taken a different approach by providing a contextual way of distinguishing markers from non-markers. Every symbol used in the algorithm is replaced by a pair of symbols, where the second member of the pair is either a 0 or a 1 depending on whether the first member is a marker or not.[2] As the first step in the algorithm, 0's are inserted after every symbol in the input string to indicate that initially every symbol is a non-marker. This is defined as:

```
macro(non_markers,[?,[]:0]*).
```

Similarly, the following macro can be used to insert a 0 after every symbol in an arbitrary expression E.

```
macro(non_markers(E),
      range(E o non_markers)).
```

Since `E` is a recognizer, it is first coerced to `identity(E)`. This form of implicit conversion is standard in the finite state calculus.

Note that 0 and 1 are perfectly ordinary alphabet symbols, which may also be used within a replacement. For example, the sequence [1,0] represents a non-marker use of the symbol 1.

2.1.3 Utilities

Before describing the algorithm, it will be helpful to have at our disposal a few general tools, most of which were described already in Kaplan and Kay (1994). These tools, however, have been modified so that they work with our approach of distinguishing markers from ordinary symbols. So to begin with, we provide macros to describe the alphabet and the alphabet extended with marker symbols:

```
macro(sig,[?,0]).
macro(xsig,[?,{0,1}]).
```

The macro `xsig` is useful for defining a specialized version of complementation and containment:

```
macro(not(X),xsig* - X).
macro($$(X),[xsig*,X,xsig*]).
```

The algorithm uses four kinds of brackets, so it will be convenient to define macros for each of these brackets, and for a few disjunctions.

```
macro(lb1,['<1',1]).
macro(lb2,['<2',1]).
macro(rb2,['2>',1]).
macro(rb1,['1>',1]).
macro(lb,{lb1,lb2}).
macro(rb,{rb1,rb2}).
macro(b1,{lb1,rb1}).
macro(b2,{lb2,rb2}).
macro(brack,{lb,rb}).
```

As in Kaplan & Kay, we define an Intro(S) operator that produces a transducer that freely introduces instances of S into an input string. We extend this idea to create a family of Intro operators. It is often the case that we want to freely introduce marker symbols into a string at any position *except* the beginning or the end.

```
%% Free introduction
macro(intro(S),{xsig-S,[] x S}*).

%% Introduction, except at begin
macro(xintro(S),{[],[xsig-S,intro(S)]}).

%% Introduction, except at end
macro(introx(S),{[],[intro(S),xsig-S]}).
```

[2] This approach is similar to the idea of laying down tracks as in the compilation of monadic second-order logic into automata Klarlund (1997, p. 5). In fact, this technique could possibly be used for a more efficient implementation of our algorithm: instead of adding transitions over 0 and 1, one could represent the alphabet as bit sequences and then add a final 0 bit for any ordinary symbol and a final 1 bit for a marker symbol.

```
%% Introduction, except at begin & end
macro(xintrox(S),{[],[xsig-S],
   [xsig-S,intro(S),xsig-S]}).
```

This family of Intro operators is useful for defining a family of Ignore operators:

```
macro( ign( E1,S),range(E1 o  intro( S))).
macro(xign( E1,S),range(E1 o xintro( S))).
macro( ignx(E1,S),range(E1 o  introx(S))).
macro(xignx(E1,S),range(E1 o xintrox(S))).
```

In order to create filter transducers to ensure that markers are placed in the correct positions, Kaplan & Kay introduce the operator `P-iff-S(L1,L2)`. A string is described by this expression iff each prefix in `L1` is followed by a suffix in `L2` and each suffix in `L2` is preceded by a prefix in `L1`. In our approach, this is defined as:

```
macro(if_p_then_s(L1,L2),
      not([L1,not(L2)])).
macro(if_s_then_p(L1,L2),
      not([not(L1),L2])).
macro(p_iff_s(L1,L2),
      if_p_then_s(L1,L2)
            &
      if_s_then_p(L1,L2)).
```

To make the use of `p_iff_s` more convenient, we introduce a new operator `l_iff_r(L,R)`, which describes strings where every string position is preceded by a string in `L` just in case it is followed by a string in `R`:

```
macro(l_iff_r(L,R),
   p_iff_s([xsig*,L],[R,xsig*])).
```

Finally, we introduce a new operator `if(Condition,Then,Else)` for conditionals. This operator is extremely useful, but in order for it to work within the finite state calculus, one needs a convention as to what counts as a boolean true or false for the condition argument. It is possible to define `true` as the universal language and `false` as the empty language:

```
macro(true,? *).   macro(false,{}).
```

With these definitions, we can use the complement operator as negation, the intersection operator as conjunction and the union operator as disjunction. Arbitrary expressions may be coerced to booleans using the following macro:

```
macro(coerce_to_boolean(E),
      range(E o (true x true))).
```

Here, `E` should describe a recognizer. `E` is composed with the universal transducer, which transduces from anything (`?*`) to anything (`?*`). Now with this background, we can define the conditional:

```
macro(if(Cond,Then,Else),
   {  coerce_to_boolean(Cond) o Then,
     ~coerce_to_boolean(Cond) o Else
   }).
```

2.2 Implementation

A rule of the form $x \rightarrow T(x)/\lambda__\rho$ will be written as `replace(T,Lambda,Rho)`. Rules of the more general form $x_1 \ldots x_n \Rightarrow T_1(x_1) \ldots T_n(x_n)/\lambda__\rho$ will be discussed in §3. The algorithm consists of nine steps composed as in figure 1.

The names of these steps are mostly derived from Karttunen (1995) and Mohri and Sproat (1996) even though the transductions involved are not exactly the same. In particular, the steps derived from Mohri & Sproat (`r`, `f`, `l1` and `l2`) will all be defined in terms of the finite state calculus as opposed to Mohri & Sproat's approach of using low-level manipulation of states and transitions.[3]

The first step, `non_markers`, was already defined above. For the second step, we first consider a simple special case. If the empty string is in the language described by `Right`, then `r(Right)` should insert an `rb2` in every string position. The definition of `r(Right)` is both simpler and more efficient if this is treated as a special case. To insert a bracket in every possible string position, we use:

```
[[[] x rb2,sig]*,[] x rb2]
```

If the empty string is not in `Right`, then we must use `intro(rb2)` to introduce the marker `rb2`, followed by `l_iff_r` to ensure that such markers are immediately followed by a string in `Right`, or more precisely a string in `Right` where additional instances of `rb2` are freely inserted in any position other than the beginning. This expression is written as:

```
            intro(rb2)
                o
l_iff_r(rb2,xign(non_markers(Right),rb2))
```

Putting these two pieces together with the conditional yields:

```
macro(r(R),
  if([] & R,       % If: [] is in R:
     [[[] x rb2,sig]*,[] x rb2],
      intro(rb2)   % Else:
            o
l_iff_r(rb2,xign(non_markers(R),rb2)))).
```

The third step, `f(domain(T))` is implemented as:

[3]The alternative implementation is provided in van Noord and Gerdemann (1999).

```
macro(replace(T,Left,Right),
            non_markers              % introduce 0 after every symbol
                o                    % (a b c => a 0 b 0 c 0).
             r(Right)                % introduce rb2 before any string
                o                    % in Right.
            f(domain(T))             % introduce lb2 before any string in
                o                    % domain(T) followed by rb2.
       left_to_right(domain(T))      % lb2 ... rb2 around domain(T) optionally
                o                    % replaced by lb1 ... rb1
       longest_match(domain(T))      % filter out non-longest matches marked
                o                    % in previous step.
          aux_replace(T)             % perform T's transduction on regions marked
                o                    % off by b1's.
             l1(Left)                % ensure that lb1 must be preceded
                o                    % by a string in Left.
             l2(Left)                % ensure that lb2 must not occur preceded
                o                    % by a string in Left.
         inverse(non_markers)).      % remove the auxiliary 0's.
```

Figure 1: Definition of `replace` operator.

```
macro(f(Phi), intro(lb2)
                 o
l_iff_r(lb2,[xignx(non_markers(Phi),b2),
             lb2^,rb2])).
```

The `lb2` is first introduced and then, using `l_iff_r`, it is constrained to occur immediately before every instance of (ignoring complexities) `Phi` followed by an `rb2`. `Phi` needs to be marked as normal text using `non_markers` and then `xign_x` is used to allow freely inserted `lb2` and `rb2` anywhere except at the beginning and end. The following `lb2^` allows an optional `lb2`, which occurs when the empty string is in `Phi`.

The fourth step is a guessing component which (ignoring complexities) looks for sequences of the form `lb2 Phi rb2` and converts some of these into `lb1 Phi rb1`, where the `b1` marking indicates that the sequence is a candidate for replacement. The complication is that `Phi`, as always, must be converted to `non_markers(Phi)` and instances of `b2` need to be ignored. Furthermore, between pairs of `lb1` and `rb1`, instances of `lb2` are deleted. These `lb2` markers have done their job and are no longer needed. Putting this all together, the definition is:

```
macro(left_to_right(Phi),
  [[xsig*,
    [lb2 x lb1,
     (ign(non_markers(Phi),b2)
              o
      inverse(intro(lb2))
     ),
     rb2 x rb1]
  ]*, xsig*]).
```

The fifth step filters out non-longest matches produced in the previous step. For example (and simplifying a bit), if `Phi` is `ab*`, then a string of the form ... rb1 a b lb1 b ... should be ruled out since there is an instance of `Phi` (ignoring brackets except at the end) where there is an internal `lb1`. This is implemented as:[4]

```
macro(longest_match(Phi),
  not($$([lb1,
          (ignx(non_markers(Phi),brack)
                    &
                 $$(rb1)
          ),     % longer match must be
          rb     % followed by an rb
         ]))     % so context is ok
          o
% done with rb2, throw away:
   inverse(intro(rb2))).
```

The sixth step performs the transduction described by `T`. This step is straightforwardly implemented, where the main difficulty is getting `T` to apply to our specially marked string:

```
macro(aux_replace(T),
  {{sig,lb2},
   [lb1,
    inverse(non_markers)
```

[4]The line with `$$(rb1)` can be optimized a bit: Since we know that an `rb1` must be preceded by `Phi`, we can write: `[ign_(non_markers(Phi),brack),rb1,xsig*])`. This may lead to a more constrained (hence smaller) transducer.

```
        o T o
      non_markers,
    rb1 x []
   ]
  }*).
```

The seventh step ensures that `lb1` is preceded by a string in `Left`:

```
macro(l1(L),
  ign(if_s_then_p(
       ignx([xsig*,non_markers(L)],lb1),
       [lb1,xsig*]),
      lb2)
        o
 inverse(intro(lb1))).
```

The eighth step ensures that `lb2` is not preceded by a string in `Left`. This is implemented similarly to the previous step:

```
macro(l2(L),
  if_s_then_p(
    ignx(not([xsig*,non_markers(L)]),lb2),
    [lb2,xsig*])
          o
 inverse(intro(lb2))).
```

Finally the ninth step, `inverse(non_markers)`, removes the `0`'s so that the final result in not marked up in any special way.

3 Longest Match Capturing

As discussed in §1 the POSIX standard requires that multiple captures follow a longest match strategy. For multiple captures as in (3), one establishes first a longest match for $domain(T_1) \cdot \ldots \cdot domain(T_n)$. Then we ensure that each of $domain(T_i)$ in turn is required to match as long as possible, with each one having priority over its rightward neighbors. To implement this, we define a macro `lm_concat(Ts)` and use it as:

```
replace(lm_concat(Ts),Left,Right)
```

Ensuring the longest overall match is delegated to the `replace` macro, so `lm_concat(Ts)` needs only ensure that each individual transducer within `Ts` gets its proper left-to-right longest matching priority. This problem is mostly solved by the same techniques used to ensure the longest match within the `replace` macro. The only complication here is that `Ts` can be of unbounded length. So it is not possible to have a single expression in the finite state calculus that applies to all possible lenghts. This means that we need something a little more powerful than mere macro expansion to construct the proper finite state calculus expression. The FSA Utilities provides a Prolog hook for this purpose. The resulting definition of `lm_concat` is given in figure 2.

Suppose (as in Friedl (1997)), we want to match the following list of recognizers against the string `topological` and insert a marker in each boundary position. This reduces to applying:

```
lm_concat([
   [{[t,o],[t,o,p]},[]: '#'],
   [{o,[p,o,l,o]},[]: '#'],
   {[g,i,c,a,l],[o^,l,o,g,i,c,a,l]}
         ])
```

This expression transduces the string `topological` only to the string `top#o#logical`.[5]

4 Conclusions

The algorithm presented here has extended previous algorithms for rewrite rules by adding a limited version of backreferencing. This allows the output of rewriting to be dependent on the form of the strings which are rewritten. This new feature brings techniques used in Perl-like languages into the finite state calculus. Such an integration is needed in practical applications where simple text processing needs to be combined with more sophisticated computational linguistics techniques.

One particularly interesting example where backreferences are essential is cascaded deterministic (longest match) finite state parsing as described for example in Abney (Abney, 1996) and various papers in (Roche and Schabes, 1997a). Clearly, the standard rewrite rules do not apply in this domain. If *NP* is an NP recognizer, it would not do to say $NP \Rightarrow [NP]/\lambda_\rho$. Nothing would force the string matched by the *NP* to the left of the arrow to be the same as the string matched by the *NP* to the right of the arrow.

One advantage of using our algorithm for finite state parsing is that the left and right contexts may be used to bring in top-down filtering.[6] An often cited advantage of finite state

[5] An anonymous reviewer suggested that `lm_concat` could be implemented in the framework of Karttunen (1996) as:

$$[\mathrm{t\ o}\ |\mathrm{t\ o\ p}\ |\ \mathrm{o}\ |\ \mathrm{p\ o\ l\ o}\ ] \rightarrow \ldots\ \#;$$

Indeed the resulting transducer from this expression would transduce `topological` into `top#o#logical`. But unfortunately this transducer would also transduce `polotopogical` into `polo#top#o#gical`, since the notion of left-right ordering is lost in this expression.

[6] The bracketing operator of Karttunen (1996), on the other hand, does not provide for left and right contexts.

```
macro(lm_concat(Ts),mark_boundaries(Domains) o ConcatTs):-
   domains(Ts,Domains), concatT(Ts,ConcatTs).

domains([],[]).
domains([F|R0],[domain(F)|R]):- domains(R0,R).

concatT([],[]).
concatT([T|Ts], [inverse(non_markers) o T,lb1 x []|Rest]):- concatT(Ts,Rest).

%% macro(mark_boundaries(L),Exp): This is the central component of lm_concat. For our
%% "toplological" example we will have:
%% mark_boundaries([domain([{[t,o],[t,o,p]},[]: #]),
%%                  domain([{o,[p,o,l,o]},[]: #]),
%%                  domain({[g,i,c,a,l],[o^,l,o,g,i,c,a,l]})])
%% which simplifies to:
%% mark_boundaries([{[t,o],[t,o,p]}, {o,[p,o,l,o]}, {[g,i,c,a,l],[o^,l,o,g,i,c,a,l]}]).
%% Then by macro expansion, we get:
%% [{[t,o],[t,o,p]} o non_markers,[]x lb1,
%%  {o,[p,o,l,o]} o non_markers,[]x lb1,
%%  {[g,i,c,a,l],[o^,l,o,g,i,c,a,l]} o non_markers,[]x lb1]
%%              o
%% % Filter 1: {[t,o],[t,o,p]} gets longest match
%% ~ [ignx_1(non_markers({[t,o],[t,o,p]}),lb1),
%%    ign(non_markers({o,[p,o,l,o]}),lb1),
%%    ign(non_markers({[g,i,c,a,l],[o^,l,o,g,i,c,a,l]}),lb1)]
%%                  o
%% % Filter 2: {o,[p,o,l,o]} gets longest match
%% ~ [non_markers({[t,o],[t,o,p]}),lb1,
%%    ignx_1(non_markers({o,[p,o,l,o]}),lb1),
%%    ign(non_markers({[g,i,c,a,l],[o^,l,o,g,i,c,a,l]}),lb1)]

macro(mark_boundaries(L),Exp):-
   boundaries(L,Exp0), % guess boundary positions
   greed(L,Exp0,Exp).  % filter non-longest matches

boundaries([],[]).
boundaries([F|R0],[F o non_markers, [] x lb1 |R]):- boundaries(R0,R).

greed(L,Composed0,Composed) :-
   aux_greed(L,[],Filters), compose_list(Filters,Composed0,Composed).

aux_greed([H|T],Front,Filters):- aux_greed(T,H,Front,Filters,_CurrentFilter).

aux_greed([],F,_,[],[ign(non_markers(F),lb1)]).
aux_greed([H|R0],F,Front,[~L1|R],[ign(non_markers(F),lb1)|R1]) :-
   append(Front,[ignx_1(non_markers(F),lb1)|R1],L1),
   append(Front,[non_markers(F),lb1],NewFront),
   aux_greed(R0,H,NewFront,R,R1).

%% ignore at least one instance of E2 except at end
macro(ignx_1(E1,E2), range(E1 o [[? *,[] x E2]+,? +])).

compose_list([],SoFar,SoFar).
compose_list([F|R],SoFar,Composed):- compose_list(R,(SoFar o F),Composed).
```

Figure 2: Definition of `lm_concat` operator.

parsing is robustness. A constituent is found bottom up in an early level in the cascade even if that constituent does not ultimately contribute to an S in a later level of the cascade. While this is undoubtedly an advantage for certain applications, our approach would allow the introduction of some top-down filtering while maintaining the robustness of a bottom-up approach. A second advantage for robust finite state parsing is that bracketing could also include the notion of "repair" as in Abney (1990). One might, for example, want to say something like: $xy \Rightarrow [_{NP}$ *RepairDet(x) RepairN(y)*$]/\lambda_\rho$[7] so that an *NP* could be parsed as a slightly malformed *Det* followed by a slightly malformed *N*. RepairDet and RepairN, in this example, could be doing a variety of things such as: contextualized spelling correction, reordering of function words, replacement of phrases by acronyms, or any other operation implemented as a transducer.

Finally, we should mention the problem of complexity. A critical reader might see the nine steps in our algorithm and conclude that the algorithm is overly complex. This would be a false conclusion. To begin with, the problem itself is complex. It is easy to create examples where the resulting transducer created by any algorithm would become unmanageably large. But there exist strategies for keeping the transducers smaller. For example, it is not necessary for all nine steps to be composed. They can also be cascaded. In that case it will be possible to implement different steps by different strategies, e.g. by deterministic or non-deterministic transducers or bimachines (Roche and Schabes, 1997b). The range of possibilities leaves plenty of room for future research.

References

Steve Abney. 1990. Rapid incremental parsing with repair. In *Proceedings of the 6th New OED Conference: Electronic Text Rese arch*, pages 1–9.

Steven Abney. 1996. Partial parsing via finite-state cascades. In *Proceedings of the ESSLLI '96 Robust Parsing Workshop*.

Jeffrey Friedl. 1997. *Mastering Regular Expressions*. O'Reilly & Associates, Inc.

C. Douglas Johnson. 1972. *Formal Aspects of Phonological Descriptions*. Mouton, The Hague.

Ronald Kaplan and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–379.

L. Karttunen, J-P. Chanod, G. Grefenstette, and A. Schiller. 1996. Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–238.

Lauri Karttunen. 1995. The replace operator. In *33th Annual Meeting of the Association for Computational Linguistics*, M.I.T. Cambridge Mass.

Lauri Karttunen. 1996. Directed replacement. In *34th Annual Meeting of the Association for Computational Linguistics*, Santa Cruz.

Lauri Karttunen. 1997. The replace operator. In Emannual Roche and Yves Schabes, editors, *Finite-State Language Processing*, pages 117–147. Bradford, MIT Press.

Lauri Karttunen. 1998. The proper treatment of optimality theory in computational phonology. In *Finite-state Methods in Natural Language Processing*, pages 1–12, Ankara, June.

Nils Klarlund. 1997. Mona & Fido: The logic automaton connection in practice. In *CSL '97*.

Mehryar Mohri and Richard Sproat. 1996. An efficient compiler for weighted rewrite rules. In *34th Annual Meeting of the Association for Computational Linguistics*, Santa Cruz.

Emmanuel Roche and Yves Schabes. 1995. Deterministic part-of-speech tagging with finite-state transducers. *Computational Linguistics*, 21:227–263. Reprinted in Roche & Schabes (1997).

Emmanuel Roche and Yves Schabes, editors. 1997a. *Finite-State Language Processing*. MIT Press, Cambridge.

Emmanuel Roche and Yves Schabes. 1997b. Introduction. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*. MIT Press, Cambridge, Mass.

Gertjan van Noord and Dale Gerdemann. 1999. An extendible regular expression compiler for finite-state approaches in natural language processing. In *Workshop on Implementing Automata 99*, Potsdam Germany.

Gertjan van Noord. 1997. Fsa utilities. The *FSA Utilities* toolbox is available free of charge under Gnu General Public License at http://www.let.rug.nl/~vannoord/Fsa/.

[7] The syntax here has been simplified. The rule should be understood as: replace(lm_concat([[]:'[np', repair_det, repair_n, []:']'],lambda, rho).