A Method for Compiling Two-level Rules with Multiple Contexts

Kimmo Koskenniemi
University of Helsinki
Helsinki, Finland
kimmo.koskenniemi@helsinki.fi

Miikka Silfverberg
University of Helsinki
Helsinki, Finland
miikka.silfverberg@helsinki.fi

Abstract

A novel method is presented for compiling two-level rules which have multiple context parts. The same method can also be applied to the resolution of so-called right-arrow rule conflicts. The method makes use of the fact that one can efficiently compose sets of two-level rules with a lexicon transducer. By introducing variant characters and using simple pre-processing of multi-context rules, all rules can be reduced into single-context rules. After the modified rules have been combined with the lexicon transducer, the variant characters may be reverted back to the original surface characters. The proposed method appears to be efficient but only partial evidence is presented yet.

1 Introduction

Two-level rules can be compiled into length-preserving transducers whose intersection effectively reflects the constraints and the correspondences imposed by the two-level grammar. Two-level rules relate input strings (lexical representations) with output strings (surface representations). The pairs of strings are treated as character pairs `x:z` consisting of *lexical (input) characters* `x` and *surface (output) characters* `z`, and regular expressions based on such pairs. Two-level rule transducers are made length-preserving (epsilon-free) by using a place holder *zero* (0) within the rules and in the representations. The zero is then removed after the rules have been combined by (virtual) intersection, before the result is composed with the lexicon. There are four kinds of two-level rules:

1. *right-arrow rules* or restriction rules, (`x:z => LC _ RC`) saying that the correspondence pair is allowed only if immediately preceded by left context `LC` and followed by right context `RC`,
2. *left-arrow rules* or surface coercion rules, (`x:z <= LC _ RC`) which say that in this context, the lexical character `x` may only correspond to the surface character `z`,
3. *double-arrow rules* (`<=>`), a shorthand combining these two requirements, and
4. *exclusion rules* (`x:z /<= LC _ RC`) which forbid the pair `x:z` to occur in this context.

All types of rules may have more than one context part. In particular, the right-arrow rule `x:z => LC1 _ RC1; LC2 _ RC2` would say that the pair `x:z` (which we call the *centre* of the rule) may occur in either one of these two contexts. For various formulations of two-level rules, see e.g. (Koskenniemi, 1983), (Grimley-Evans et al., 1996), (Black et.al., 1987), (Ruessink, 1989), (Ritchie, 1992), (Kiraz, 2001) and a comprehensive survey on their formal interpretations, see (Vaillette, 2004).

Compiling two-level rules into transducers is easy in all other cases except for right-arrow rules with multiple context-parts; see e.g. Koskenniemi (1983). Compiling right-arrow rules with multiple context parts is more difficult because the compilation of the whole rule is not in a simple relation to the component expressions in the rule; see e.g. Karttunen et al. (1987).

The method proposed here reduces multi-context rules into a set of separate simple rules, one for each context, by introducing some auxiliary variant characters. These auxiliary characters are then normalized back into the original surface characters after the intersecting composition of the lexicon and the modified rules. The method is presented in section 3. The compilation of multiple contexts using the proposed scheme appears to be very simple and fast. Preliminary results and discussion about the computational complexity are presented in section 4.

Proceedings of the 11th Meeting of the ACL-SIGMORPHON, ACL 2010, pages 38–45,
Uppsala, Sweden, 15 July 2010.

1.1 The compilation task with an example

We make use of a simplified linguistic example where a stop k is realized as v between identical rounded close vowels (u, y). The example resembles one detail of Finnish consonant gradation but it is grossly simplified. According to the rule in the example, the lexical representation pukun would be realized as the surface representation puvun. This correspondence is traditionally represented as:

```
p u k u n
p u v u n
```

where the upper tier represents the lexical or morphophonemic representation which we interpret as the input, and the lower one corresponds to the surface representation which we consider as the output.[1] This two-tier representation is usually represented on a single line as a sequence of input and output character pairs where pairs of identical characters, such as p:p are abbreviated as a single p. E.g. the above pair of strings becomes a string of pairs:

```
p u k:v u n
```

In our example we require that the correspondence k:v may occur only between two identical rounded close vowels, i.e. either between two letters u or between two letters y. Multiple contexts are needed in the right-arrow rule which expresses this constraint. As a two-level grammar, this would be:

```
Alphabet a b … k … u v w …
 k:v;
Rules
k:v => u _ u;
       y _ y;
```

This grammar would permit sequences such as:

```
p u k:v u n
k y k:v y n
p u k:v u k:v u n
l u k:v u n k y k:v y n
t u k k u
```

but it would exclude sequences:

```
p u k:v y n
t u k:v a n
```

Whereas one can always express multi-context left-arrow rules (<=) and exclusion rules (/<=) equivalently as separate rules, this does not hold for right-arrow rules. The two separate rules

```
k:v => u _ u;
k:v => y _ y;
```

would be in conflict with each other permitting no occurrences of k:v at all, (unless we apply so-called conflict resolution which would effectively combine the two rules back to a single rule with two context parts).

2 Previous compilation methods

The first compiler of two-level rules was implemented by the first author in 1985 and it handled also multi-context rules (Koskenniemi, 1985). The compiler used a finite-state package written by Ronald Kaplan and Martin Kay at Xerox PARC, and a variant of a formula they used for compiling cascaded rewrite rules. Their own work was not published until 1994. Koskenniemi's compiler was re-implemented in LISP by a student in her master's thesis (Kinnunen, 1987).

Compilation of two-level rules in general requires some care because the centres may occur several times in pair strings, the contexts may overlap and the centres may act as part of a context for another occurrence of the same centre. For other rules than right-arrow rules, each context is yet another condition for excluding ungrammatical strings of pairs, which is how the rules are related to each other. The context parts of a right-arrow rule are, however, permissions, one of which has to be satisfied. Expressing unions of context parts was initially a problem which required complicated algorithms.

Some of the earlier compilation methods are mentioned below. They all produce a single transducer out of each multi-context right-arrow rule.

2.1 Method based on Kaplan and Kay

Kaplan and Kay (1994) developed a method around 1980 for compiling rewriting rules into finite-state transducers[2]. The method was adapted by Koskenniemi to the compilation of two-level rules by modifying the formula

[1] In Xerox terminology, the input or lexical characters are called the upper characters, and the output or surface characters are called the lower characters. Other orientations are used by some authors.

[2] Douglas Johnson (1972) presented a similar technique earlier but his work was not well known in early 1980s.

slightly. In this method, auxiliary left and right bracket characters (<1, >1, <2, >2, ...) were freely added in order to facilitate the checking of the context conditions. A unique left and right bracket was dedicated for each context part of the rule. For each context part of a rule, sequences with freely added brackets were then filtered with the context expressions so that only such sequences remained where occurrences of the brackets were delimited with the particular left or right context (allowing free occurrence of brackets for other context parts). Thereafter, it was easy to check that all occurrences of the centre (i.e. the left hand part of the rule before the rule operator) were delimited by some matching pair of brackets. As all component transducers in this expression were length-preserving (epsilon-free), the constraints could be intersected with each other resulting in a single rule transducer for the multi-context rule (and finally the brackets could be removed).

2.2 Method of Grimley-Evans, Kiraz and Pulman

Grimley-Evans, Kiraz and Pulman presented a simpler compilation formula for two-level rules (1996). The method is prepared to handle more than two levels of representation, and it does not need the freely added brackets in the intermediate stages. Instead, it uses a marker for the rule centre and can with it express disjunctions of contexts. Subtracting such a disjunction from all strings where the centre occurs expresses all pair strings which violate the multi-context rule. Thus, the negation of such a transducer is the desired result.

2.3 Yli-Jyrä's method

Yli-Jyrä (Yli-Jyrä et al., 2006) introduced a concept of Generalized Restriction (GR) where expressions with auxiliary boundary characters ♦ made it possible to express context parts of rules in a natural way, e.g. as:

```
Pi* LC ♦ Pi ♦ RC Pi*
```

Here Pi is the set of feasible pairs of characters and LC and RC are the left and right contexts. The two context parts of our example would correspond to the following two expressions:

```
Pi* u ♦ Pi ♦ u Pi*
Pi* y ♦ Pi ♦ y Pi*
```

Using such expressions, it is easy to express disjunctions of contexts as unions of the above expressions. This makes it logically simple to compile multi-context right-arrow rules. The rule centre x:z can be expressed simply as:

```
Pi* ♦ x:z ♦ Pi*
```

The right-arrow rule can be expressed as an implication where the expression for the centre implies the union of the context parts. Thereafter, one may just remove the auxiliary boundary characters, and the result is the rule-transducer. (It is easy to see that only one auxiliary character is needed when the length of the centres is one.)

The compilation of rules with centres whose length is one using the GR seems very similar to that of Grimley-Evans et al. The nice thing about GR is that one can easily express various rule types, including but not limited to the four types listed above.

2.4 Intersecting compose

It was observed somewhere around 1990 at Xerox that the rule sets may be composed with the lexicon transducers in an efficient way and that the resulting transducer was roughly similar in size as the lexicon transducer itself (Karttunen et al., 1992). This observation gives room to the new approach presented below.

At that time, it was not practical to intersect complete two-level grammars if they contained many elaborate rules (and this is still a fairly heavy operation). Another useful observation was that the intersection of the rules could be done in a joint single operation with the composition (Karttunen, 1994). Avoiding the separate intersection made the combining of the lexicon and rules feasible and faster. In addition to Xerox LEXC program, e.g. the HFST finite-state software contains this operation and it is routinely used when lexicons and two-level grammars are combined into lexicon transducers (Lindén et al., 2009).

Måns Huldén has noted (2009) that the composing of the lexicon and the rules is sometimes a heavy operation, but can be optimized if one first composes the output side of the lexicon transducer with the rules, and thereafter the original lexicon with this intermediate result.

3 Proposed method for compilation

The idea is to modify the two-level grammar so that the rules become simpler. The modified grammar will contain only simple rules with single context parts. This is done at the cost that the grammar will transform lexical representations into slightly modified surface representations.

The surface representations are, however, fixed after the rules have been combined with the lexicon so that the resulting lexicon transducer is equivalent to the result produced using earlier methods.

3.1 The method through the example

Let us return to the example in the introduction. The modified surface representation differs from the ultimate representation by having a slightly extended alphabet where some surface characters are expressed as their variants, i.e. there might be `v1` or `v2` in addition to `v`. In particular, the first variant `v1` will be used exactly where the first context of the original multi-context rule for `k:v` is satisfied, and `v2` where the second context is satisfied. After extending the alphabet and splitting the rule, our example grammar will be as follows:

```
Alphabet a b … k … u v w x y …
  k:v1 k:v2;
Rules
k:v1 => u _ u;
k:v2 => y _ y;
```

These rules would permit sequences such as:

```
p u k:v1 u n
k y k:v2 y n
p u k:v1 u k:v1 u n
```

but exclude a sequence

```
p u k:v2 u n
```

The output of the modified grammar is now as required, except that it includes these variants `v1` and `v2` instead of `v`. If we first perform the intersecting composition of the rules and the lexicon, we then can compose the result with a trivial transducer which simply transforms both `v1` and `v2` into `v`.

It should be noted that here the context expressions of these example rules do not contain `v` on the output side, and therefore the introduction of the variants `v1` and `v2` causes no further complications. In the general case, the variants should be added as alternatives of `v` in the context expressions, see the explanation below.

3.2 More general cases

The strategy is to pre-process the two-level grammar in steps by splitting more complex constructions into simpler ones until we have units whose components are trivial to compile. The intersection of the components will have the desired effect when composed with a lexicon and a trivial correction module. Assume, for the time being, that all centres (i.e. the left-hand parts) of the rules are of length one.

(1) Split double-arrow (<=>) rules into one right-arrow (=>) rule and one left-arrow (<=) rule with centres and context parts identical to those of the original double-arrow rule.

(2) Unfold the iterative `where` clauses in left-arrow rules by establishing a separate left-arrow rule for each value of the iterator variable, e.g.

```
V:Vb <= [a | o | u] ?* _;
 where V in (A O U)
       Vb in (a o u) matched;
```

becomes

```
A:a <= [a | o | u] ?* _;
O:o <= [a | o | u] ?* _;
U:u <= [a | o | u] ?* _;
```

Unfold the `where` clauses in right-arrow rules in either of the two ways: (a) If the `where` clauses create disjoint centres (as above), then establish a separate right-arrow rule for each value of the variable, and (b) if the clause does not affect the centre, then create a single multi-context right-arrow rule whose contexts consist of the context parts of the original rule by replacing the `where` clause variable by its values, one value at a time, e.g.

```
k:v => Vu _ Vu; where Vu in (u y);
```

becomes

```
k:v => u _ u;
       y _ y;
```

If there are set symbols or disjunctions in the centres of a right-arrow rule, then split the rule into separate rules where each rule has just a single pair as its centre, and the context part is identical to the context part (after the unfolding of the `where` clauses).

Note that these two first steps would probably be common to any method of compiling multi-context rules. After these two steps, we have right-arrow, left-arrow and exclusion rules. The right-arrow rules have single pairs as their centres.

(3) Identify the right-arrow rules which, after the unfolding, have multiple contexts, and record each pair which is the centre of such a rule. Suppose that the output character (i.e. the surface character) of such a rule is z and there are n context parts in the rule, then create n new auxiliary characters $z_1, z_2, \ldots, z_n$ and denote the set consisting of them by $S(z)$.

Split the rule into n distinct single-context right-arrow rules by replacing the z of the centre by each z_i in turn.

Our simple example rule becomes now.

```
k:v1 => u _ u;
k:v2 => y _ y;
```

(4) When all rules have been split according to the above steps, we need a post-processing phase for the whole grammar. We have to extend the alphabet by adding the new auxiliary characters in it. If original surface characters (which now have variants) were referred to in the rules, each such reference must be replaced with the union of the original character and its variants. This replacement has to be done throughout the grammar. For any existing pairs $x{:}z$ listed in the alphabet, we add there also the pairs $x{:}z_1$, ..., $x{:}z_n$. The same is done for all declarations of sets where z occurs (as an output character). Insert a declaration for a new character set corresponding to $S(z)$. In all define clauses and in all rule-context expressions where z occurs as an output character, it is replaced by the set $S(z)$. In all centres of left-arrow rules where z occurs as the output character, it is replaced by $S(z)$.

The purpose of this step is just to make the modified two-level grammar consistent in terms of its alphabet, and to make the modified rules treat the occurrence of any of the output characters $z_1, z_2, \ldots, z_n$ in the same way as the original rule treated z wherever it occurred in its contexts.

After this pre-processing we only have right-arrow, left-arrow and exclusion rules with a single context part. All rules are independent of each other in such a way that their intersection would have the effect we wish the grammar to have. Thus, we may compile the rule set as such and each of these simple rules separately. Any of the existing compilation formulas will do.

After compiling the individual rules, they have to be intersected and composed with the lexicon transducer which transforms base forms and inflectional feature symbols into the morphophonemic representation of the word-forms. The composing and intersecting is efficiently done as a single operation because it then avoids the possible explosion which can occur if intermediate result of the intersection is computed in full.

The rules are mostly independent of each other, capable of recurring freely. Therefore something near the worst case complexity is likely to occur, i.e. the size of the intersection would have many states, roughly proportional to the product of the numbers of the states in the individual rule transducers.

The composition of the lexicon and the logical intersection of the modified rules is almost identical to the composition of the lexicon and the logical intersection of the original rules. The only difference is that the output (i.e. the surface) representation contains some auxiliary characters z_i instead of the original surface characters z. A simple transducer will correct this. (The transducer has just one (final) state and identity transitions for all original surface characters and a reduction $z_i{:}z$ for each of the auxiliary characters.) This composition with the correcting transducer can be made only after the rules have been combined with the lexicon.

3.3 Right-arrow conflicts

Right-arrow rules are often considered as permissions. A rule could be interpreted as "this correspondence pair may occur if the following context condition is met". Further permissions might be stated in other rules. As a whole, any occurrence must get at least one permission in order to be allowed.

The right-arrow conflict resolution scheme presented by Karttunen implemented this through an extensive pre-processing where the conflicts were first detected and then resolved (Karttunen et al., 1987). The resolution was done by copying context parts among the rules in conflict. Thus, what was compiled was a grammar with rules extended with copies of context parts from other rules.

The scenario outlined above could be slightly modified in order to implement the simple right-arrow rule conflict resolution in a way which is equivalent to the solution presented by Karttunen. All that is needed is that one would first split the right-arrow rules with multiple context parts into separate rules. Only after that, one would consider all right-arrow rules and record rules with identical centres. For groups of rules with identical centres, one would introduce the further variants of the surface characters, a separate variant for each rule. In this scheme, the conflict resolution of right-arrow rules is implemented fairly naturally in a way analogous to the handling of multi-context rules.

3.4 Note on longer centres in rules

In the above discussion, the left-hand parts of rules, i.e. their centres, were always of length one. In fact, one may define rules with longer centres by a scheme which reduces them into

rules with length one centres. It appears that the basic rule types (the left and right-arrow rules) with longer centres can be expressed in terms of length one centres, if we apply conflict resolution for the right-arrow rules.

We replace a right-arrow rule, e.g.

$x_1:z_1\ x_2:z_2\ \ldots\ x_k:z_k$ => LC _ RC;

with k separate rules

$x_1:z_1$ => LC _ $x_2:z_2\ \ldots\ x_k:z_k$ RC;
$x_2:z_2$ => LC $x_1:z_1$ _ $\ldots\ x_k:z_k$ RC;

...
$x_k:z_k$ => LC $x_1:z_1\ x_2:z_2\ \ldots$ _ RC;

Effectively, each input character may be realized according to the original rule only if the rest of the centre will also be realized according to the original rule.

Respectively, we replace a left-arrow rule, e.g.

$x_1:z_1\ x_2:z_2\ \ldots\ x_k:z_k$ <= LC _ RC;

with k separate rules

$x_1:z_1$ <= LC _ x_2: $\ldots\ x_k$: RC;
$x_2:z_2$ <= LC x_1: _ $\ldots\ x_k$: RC;

...
$x_k:z_k$ <= LC x_1: x_2: ... _ RC;

Here the realization of the surface string is forced for each of its character of the centre separately, without reference to what happens to other characters in the centre. (Otherwise the contexts of the separate rules would be too restrictive, and allow the default realization as well.)

4 Complexity and implementation

In order to implement the proposed method, one could write a pre-processor which just transforms the grammar into the simplified form, and then use an existing two-level compiler. Alternatively, one could modify an existing compiler, or write a new compiler which would be somewhat simpler than the existing ones. We have not implemented the proposed method yet, but rather simulated the effects using existing two-level rule compilers. Because the pre-processing would be very fast anyway, we decided to estimate the efficiency of the proposed method through compiling hand-modified rules with the existing HFST-TWOLC (Lindén et al., 2009) and Xerox TWOLC[3] two-level rule compilers. The HFST tools are built on top of existing open source finite-state packages OpenFST (Allauzen et al., 2007) and Helmut Schmid's SFST (2005).

It appears that all normal morphographemic two-level grammars can be compiled with the methods of Kaplan and Kay, Grimley-Evans and Yli-Jyrä.

Initial tests of the proposed scheme are promising. The compilation speed was tested with a grammar of consisting of 12 rules including one multi-context rule for Finnish consonant gradation with some 8 contexts and a full Finnish lexicon. When the multi-context rule was split into separate rules, the compilation was somewhat faster (12.4 sec) to than when the rule was compiled a multi-context rule using the GR formula (13.9 sec). The gain in the speed by splitting was lost at the additional work needed in the intersecting compose of the rules and the full lexicon and the final fixing of the variants. On the whole, the proposed method had no advantage over the GR method.

In order to see how the number of context parts affects the compilation speed, we made tests with an extreme grammar simulating Dutch hyphenation rules. The hyphenation logic was taken out of TeX hyphenation patterns which had been converted into two-level rules. The first grammar consisted of a single two-level rule which had some 3700 context parts. This grammar could not be compiled using Xerox TWOLC which applies the Kaplan and Kay method because more than 5 days on a dedicated Linux machine with 64 GB core memory was not enough for completing the computation. When using of GR method of HFST-TWOLC, the compilation time was not a problem (34 minutes). The method of Grimley-Evans et al. would probably have been equally feasible.

Compiling the grammar after splitting it into separate rules as proposed above was also feasible: about one hour with Xerox TWOLC and about 20 hours with HFST-TWOLC. The difference between these two implementations depends most likely on the way they handle alphabets. The Xerox tool makes use of a so-called 'other' symbol which stands for characters not mentioned in the rule. It also optimizes the computation by using equivalence classes of character pairs. These make the compilation less sensitive to the 3700 new symbols added to the alphabet than what happens in the HFST routines.

Another test was made using a 50 pattern subset of the above hyphenation grammar. Using

[3] We used an old version 3.4.10 (2.17.7) which we thought would make use of the Kaplan and Kay formula. We suspected that the most recent versions might have gone over to the GR formula.

the Xerox TWOLC, the subset compiled as a multi-context rule in 28.4 seconds, and when split according to the method proposed here, it compiled in 0.04 seconds. Using the HFST-TWOLC, the timings were 3.1 seconds and 5.4 seconds, respectively. These results corroborate the intuition that the Kaplan and Kay formula is sensitive to the number of context parts in rules whereas the GR formula is less sensitive to the number of context parts in rules.

There are factors which affect the speed of HFST-TWOLC, including the implementation detail including the way of treating characters or character pairs which are not specifically mentioned in a particular transducer. We anticipate that there is much room for improvement in treating larger alphabets in HFST internal routines and there is no inherent reason why it should be slower than the Xerox tool. The next release of HFST will use Huldén's FOMA finite-state package. FOMA implements the 'other' symbol and is expected to improve the processing of larger alphabets.

Our intuition and observation is that the proposed compilation phase requires linear time with respect to the number of context parts in a rule. Whether the proposed compilation method has an advantage over the compilation using the GR or Grimley-Evans formula remains to be seen.

5 Acknowledgements

Miikka Silfverberg, a PhD student at Finnish graduate school Langnet and the author of HFST-TWOLC compiler. His contribution consists of making all tests used here to estimate and compare the efficiency of the compilation methods.

The current work is part of the FIN-CLARIN infrastructure project at the University of Helsinki funded by the Finnish Ministry of Education.

References

Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut and Mehryar Mohri. 2007. OpenFst: A General and Efficient Weighted Finite-State Transducer Library. In *Implementation and Application of Automata*, Lecture Notes in Computer Science. Springer, Vol. 4783/2007, 11-23.

Alan Black, Graeme Ritchie, Steve Pulman, and Graham Russell. 1987. "Formalisms for morphographemic description". In *Proceedings of the Third Conference of the European Chapter of the Association for Computational Linguistics*, 11–18.

Edmund Grimley-Evans, Georg A. Kiraz, Stephen G. Pulman. 1996. Compiling a Partition-Based Two-Level Formalism. In *COLING 1996, Volume 1: The 16th International Conference on Computational Linguistics*, pp. 454-459.

Huldén, Måns. 2009. *Finite-State Machine Construction Methods and Algorithms for Phonology and Morphology*. PhD Thesis, University of Arizona.

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

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

Lauri Karttunen. 1994. Constructing lexical transducers. In Proceedings of the 15th conference on Computational linguistics, Volume 1. pp. 406-411.

Lauri Karttunen, Ronald M. Kaplan, and Annie Zaenen. 1992. Two-Level Morphology with Composition. *Proceedings of the 14th conference on Computational linguistics, August 23-28, 1992, Nantes, France*. 141-148.

Lauri Karttunen, Kimmo Koskenniemi, and Ronald. M. Kaplan. 1987. A Compiler for Two-level Phonological Rules. In Dalrymple, M. et al. *Tools for Morphological Analysis*. Center for the Study of Language and Information. Stanford University. Palo Alto.

Maarit Kinnunen. 1987. *Morfologisten sääntöjen kääntäminen äärellisiksi automaateiksi*. (Translating morphological rules into finite-state automata. Master's thesis.). Department of Computer Science, University of Helsinki

George Anton Kiraz. 2001. *Computational Nonlinear Morphology: With Emphasis on Semitic Languages*. Studies in Natural Language Processing. Cambridge University Press, Cambridge.

Kimmo Koskenniemi. 1983. *Two-Level Morphology: A General Computational Model for Word-form Recognition and Production*. University of Helsinki, Department of General Linguistics, Publications No. 11.

Kimmo Koskenniemi. 1985. Compilation of automata from morphological two-level rules. In F. Karlsson (ed.), *Papers from the fifth Scandinavian Conference of Computational Linguistics, Helsinki, December 11-12, 1985*. pp. 143-149.

Krister Lindén, Miikka Silfverberg and Tommi Pirinen. 2009. HFST Tools for Morphology – An Efficient Open-Source Package for Construction of Morphological Analyzers. In *State of the Art in Computational Morphology* (Proceedings of Workshop on Systems and Frameworks for Computational Morphology, SFCM 2009). Springer.

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

H. A. Ruessink. 1989. *Two level formalisms*. Utrecht Working Papers in NLP. Technical Report 5.

Helmut Schmid. 2005. A Programming Language for Finite State Transducers. In *Proceedings of the 5th International Workshop on Finite State Methods in Natural Language Processing (FSMNLP 2005)*. pp. 50-51.

Nathan Vaillette. 2004. *Logical Specification of Finite-State Transductions for Natural Language Processing*. PhD Thesis, Ohio State University.

Anssi Yli-Jyrä and Kimmo Koskenniemi. 2006. Compiling Generalized Two-Level Rules and Grammars. *International Conference on NLP: Advances in natural language processing*. Springer. 174 – 185.