XFST2FSA: Comparing Two Finite-State Toolboxes

Yael Cohen-Sygal
Department of Computer Science
University of Haifa
yaelc@cs.haifa.ac.il

Shuly Wintner
Department of Computer Science
University of Haifa
shuly@cs.haifa.ac.il

Abstract

This paper introduces *xfst2fsa*, a compiler which translates grammars expressed in the syntax of the XFST finite-state toolbox to grammars in the language of the FSA Utilities package. Compilation to FSA facilitates the use of grammars developed with the proprietary XFST toolbox on a publicly available platform. The paper describes the non-trivial issues of the compilation process, highlighting several shortcomings of some published algorithms, especially where replace rules are concerned. The compiler augments FSA with most of the operators supported by XFST. Furthermore, it provides a means for comparing the two systems on comparable grammars. The paper presents the results of such a comparison.

1 Introduction

Finite-state technology is widely considered to be the appropriate means for describing the phonological and morphological phenomena of natural languages since the pioneering works of Koskenniemi (1983) and Kaplan and Kay (1994). Finite state technology has some important advantages, making it most appealing for implementing natural language morphology. One can find it very hard, almost impossible, to build the full automaton or transducer describing some morphological phenomena. This difficulty arises from the fact that there are a great number of morpho-phonological processes combining together to create the full language. However, it is usually very easy to build a finite state machine to describe a specific morphological phenomenon. The closure properties of regular languages make it most convenient to implement each phenomenon independently and combine them together. Moreover, finite state techniques have the advantage of being efficient in their time and space complexity, as the membership problem is solvable in time linear in the length of the input. Furthermore, there are known algorithms for minimizing and determinizing automata and some restricted kinds of transducers.

Several finite state toolboxes (software packages) provide extended regular expression description languages and compilers of the expressions to finite state devices, automata and transducers (Karttunen et al., 1996; Beesley and Karttunen, 2003; Mohri, 1996; van Noord and Gerdemann, 2001a; van Noord and Gerdemann, 2001b). Such toolboxes typically include a language for extended regular expressions and a compiler from regular expressions to finite-state devices (automata and transducers). These toolboxes also include efficient implementations of several standard algorithms on finite state machines, such as union, intersection, minimization, determinization etc. More importantly, they also implement special operators that are useful for linguistic description, such as replacement (Kaplan and Kay, 1994; Mohri and Sproat, 1996; Karttunen, 1997; Gerdemann and van Noord, 1999) or predicates over alphabet symbols (van Noord and Gerdemann, 2001a; van Noord and Gerdemann, 2001b), and even operators for particular linguistic theories such as Optimality Theory (Karttunen, 1998; Gerdemann and van Noord, 2000). Unfortunately, there are no standards for the syntax of extended regular expression languages and switching from one tool-

box to another is a non-trivial task.

This work focuses on two toolboxes, XFST (Beesley and Karttunen, 2003) and FSA Utilities (van Noord, 2000). Both are powerful tools for specifying and manipulating finite state machines (acceptors and transducers) using extended regular expression languages. In addition to the standard operators, XFST also provides advanced operators such as replacement, markup, and restriction (Karttunen, 1995; Karttunen, 1996; Karttunen, 1997; Karttunen and Kempe, 1995), and advanced methods such as compile-replace and flag-diacritics. FSA, on the other hand, supports weighted finite state machines and provides visualization of finite state networks. In addition, FSA is built over Prolog, allowing the additional usage of Prolog predicates. A first significant difference between the two packages is the wide variety of operators that XFST provides in comparison to FSA. However, FSA has the clear advantage of being a free, open source package, whereas XFST is proprietary.

This paper describes *xfst2fsa*, a compiler which translates XFST grammars to grammars in the language of the FSA Utilities package.[1] There is a strong parallelism between the languages, but certain constructs are harder to translate and require more innovation. In particular, all the replace operators that XFST provides do not exist in FSA and had to be re-implemented. In this work we relate only to the core of the finite state calculus – naïve automata and transducers. We do not deal with extended features such as the weighted networks of FSA or with advanced methods such as Prolog capabilities in FSA and compile replace and flag diacritics in XFST.

The contribution of this work is manifold. Our main motivation is to facilitate the use of grammars developed with XFST on publicly available systems. Furthermore, this work gives a closer insight into the theoretical algorithms which XFST is based on. We show that the algorithms published in the literature are incomplete and require refinement in order to be correct for all inputs. Moreover, our compiler enriches FSA with implementations of several replace rules, thereby scaling up the system and improving its expressivity. Finally, this work offers an investigation of two similar, but different systems: the compiler facilitates a comparison of the two systems on very similar benchmarks.

2 The xfst2fsa compiler

Compilation of a given XFST grammar into an equivalent FSA code is done in two main stages: first, the XFST grammar is parsed, and a tree representing its syntax is created. Then, by traversing the tree, the equivalent FSA grammar is generated.

In order to parse XFST grammars, a specification of the XFST syntax is required. Unfortunately, we were unable to obtain a formal specification and we resorted to reconstructing it from available documentation (Beesley and Karttunen, 2003).This turned out to be a minor inconvenience; a more major obstacle was the semantics of XFST expressions, especially those involving advanced operators such as replace rules, markup and restriction. We exemplify in this section some of these issues.

XFST operators can be divided into three groups with respect to their FSA equivalence: those which have an equivalent operator in FSA, those which do not but can be easily constructed from basic FSA operators, and those whose constructions is more complicated. In what follows we refer to operators of the first two groups as basic operators. Figure 1 displays a comparison table of some basic operators in XFST and FSA.[2] For example, consider the XFST operator `$?A` ("contains at most one"). This operator is not provided by FSA but can be constructed as `{$A - ignore([A,A],?*),?* - $A}`. It is now provided by FSA in our publicly available package of extended FSA operators. The same holds for XFST operators such as `A./.B` (internally ignore), `$.A` (contains one) etc. As another example consider the XFST operator `^` (n-ary concatenation). It does not have an equivalent operator in FSA, but it can be simply constructed in FSA by explicitly expressing the concatenation as many times as needed. Thus, the XFST regular expression `[a*|b^3]` is translated into the equivalent FSA regular expression `{a*,[b,b,b]}`. Similar techniques are used for other basic XFST operators such as `A^>n` (more

[1] The system and the source code are available for download from http://cl.haifa.ac.il/projects/fstfsa/index.shtml

[2] The complete list of XFST operators and the way they where translated into FSA can be found in Appendix A.

than n concatenations of A), `A^{n,k}` (n to k concatenations of A) etc.

Another minor issue is the different operator precedence in XFST and FSA. This problem was solved by bracketing each translated operator in XFST with '( )' to force the correct precedence.

Special care is needed in order to deal with XFST operators of the third group, e.g., all the replace, markup and restriction rules in XFST. These rules do not have any equivalents in FSA, and hence the only way to use them in FSA is to implement them from scratch. This was done using the existing documentation (Karttunen, 1995; Karttunen, 1996; Karttunen, 1997; Karttunen and Kempe, 1995) on the construction of these operators from the basic ones. However, not all the operators are fully documented and in some cases some innovation was needed. As an example, consider the XFST operator `A@<-B` (obligatory, lower to upper, left to right, longest match replacement). To the best of our knowledge, this operator is not documented. However, by Karttunen (1995), the operator `A<-B` (obligatory, lower to upper replacement) is defined as `[B->A].i` (where `B->A` is the obligatory, upper to lower replacement of the language B by the language A). We then concluded that `A@<-B` is constructed as `[B@->A].i` (where `[B@->A]` is the obligatory, upper to lower, left to right, longest match replacement of the language B by the language A) and from Karttunen (1996) the construction of the operator `B@->A` is known.

For some of the documented operators, we found that the published algorithms are erroneous in some special cases. Consider the replace operator `A->B || L _ R` (conditional replacement of the language A by the language B, in the context of L on the left and R on the right side, where both contexts are on the upper side). In Karttunen (1997; 1995), a detailed description of the construction of this operator is given. In addition, Karttunen (1997) discusses some boundary cases, such as the case in which the language A contains the empty string. We discovered that there are some cases which are not discussed as boundary ones in Karttunen (1997) and for which the standard algorithm in Karttunen (1997; 1995) does not produce the expected result by the definition of the operator denotation. One such case is a rule of the form `A->B || _ ?`, where A and B are some regular expressions denoting languages. This rule states that any member of the language A on the upper side is replaced by all members of the language B on the lower side when the upper side member is not followed by the end of the string on which the rule operates. For example, the rule `a->b || _ ?` is expected to generate the automaton of Figure 2. However, a direct implementation of the algorithms of Karttunen (1997; 1995) always yields a network accepting the empty language, independently of the way A and B are defined. Other ambiguous cases are discussed in Vaillette (2004).

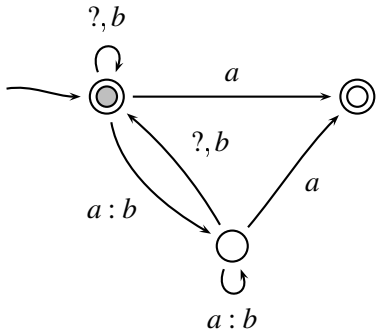

Figure 2: Desired interpretation of the regular expression $a -> b \,||\, \_\,?$

Furthermore, in some cases XFST produces networks that are somewhat different from the ones in the literature: the relations (as sets) are equal but the resulting automata (as graphs) are not isomorphic. For example, consider the replace rule `a->b || c _ d`. This expression is compiled by XFST to the automaton shown in Figure 3. Implementing this rule from basic operators as described in Karttunen (1997; 1995), results in the automaton of Figure 4. Observe that in some cases multiple accepting paths are obtained. This is probably a result of adding ε-self-loops in order to deal correctly with ε symbols, following Pereira and Riley (1997); the multiple paths can then be removed using filters. We assume that the same solution is adopted by XFST. This solution requires direct access to the underlying network, and cannot be applied at the level of the regular expression language. Therefore, we did not utilitize it in our implementation of replace rules.

To validate the construction of the compiler, one would ideally want to check that the obtained FSA

XFST syntax	FSA syntax	Meaning
`A*`	`A*`	Kleene star
`A | B`	`{A,B}`	union
`A & B`	`A & B`	intersection
`A - B`	`A - B`	A minus B
`A/B`	`ignore(A,B)`	A ignoring B
`$A`	`$A`	containment
`$?A`	`{$A - ignore([A,A],?*) , ?* - $A}`	maximum one containment
`A B`	`[A,B]`	concatenation
`A^n`	does not exist	n-ary concatenation
`A.x.B`	`A x B`	crossproduct
`A.o.B`	`A o B`	composition
`(A)`	`A^`	optionality
`[ ]`	`( )`	precedence
`R.i`	`invert(R)` or `inverse(R)`	regular relation inverse

Figure 1: A comparison table of some simple classic operators in XFST and FSA

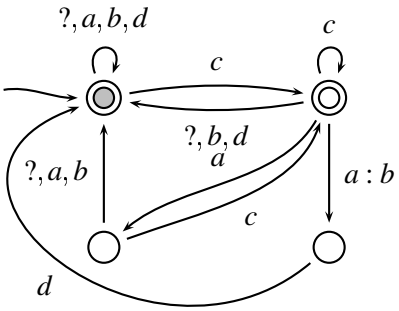

Figure 3: Automaton created from the regular expression $a- > b \,||\, c \_ d$ by XFST

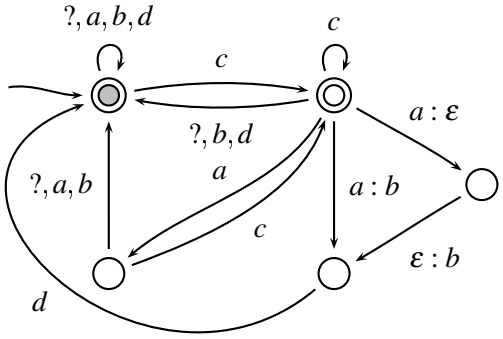

Figure 4: Automaton created from the regular expression $a- > b \,||\, c \_ d$ by the published algorithm

networks are equivalent to the XFST ones from which they were generated. Unfortunately, this is only possible for very small networks, since XFST does not allow to print its networks, when they are significantly large. We could only test XFST networks and their FSA images over test strings to validate the identity of the outputs. In addition to checking each operator by itself for several instances, we tested the compiler on a more comprehensive code, namely HAMSAH (Yona and Wintner, 2005), which was designed and implemented using XFST. We successfully converted the entire network into FSA with the compiler. Exhaustive tests produced the same outputs for both networks.

3 Comparison of XFST and FSA

A byproduct of the compiler is a full implementation, in FSA, of a vast majority of XFST's operators.[3] In addition to the contribution to FSA users, this also facilitates an effective comparison between the two toolboxes on similar benchmarks. We now describe the results of such a comparison, focusing on usability and performance.

3.1 Display of networks

FSA displays networks in two possible formats: as text, by listing the network states and transitions,

[3] We implemented in FSA all the operators of XFST, except parallel conditional replace rules and some direct replacement and markup rules.

and through a graphical user interface. The GUI that FSA employs is user friendly, allows many kinds of manipulations of the networks and significantly helps to the understanding of the networks, especially when they are small. The viewing parameters can be scaled by the user, thus improving the visualization possibilities. Moreover, networks can be saved in many different formats including binary (for fast loading and saving), text (allowing inspection of the network without the necessity to use FSA) and postscript (for printing). FSA also enables generation of C, C++, Java and Prolog code, implementing analysis with a network.

XFST, on the other hand, prints its networks only in text format, and even this is supported for small networks only. Networks in XFST can be saved in binary format only, thus requiring the usage of XFST in order to inspect the network. With respect to visual display and ease of use, therefore, FSA has clear benefits over XFST.

3.2 Performance

A true comparison of the two systems should compare two different grammars, each designed specifically for one of the two toolboxes, yielding the same comprehensive network. However, as such grammars are not available, we compared the two toolboxes using a grammar designed and implemented in XFST and its conversion into FSA. Again we used HAMSAH (Yona and Wintner, 2005) for this purpose. The Hebrew morphological analyzer is a large XFST grammar: the complete network consists of approximately 2 million states and 2.2 million arcs. We also inspected two subnetworks: the Hebrew adjectives network (approximately 100,000 states and 120,000 arcs) and the Hebrew nouns network (approximately 700,000 states and 950,000 arcs). Each of the networks was created by composing a series of rules over a large-scale lexicon. Since Hebrew morphology is non-trivial, the final network is created by composing many intermediate complex regular expressions (including many replace rules and compositions). The grammars were compiled and executed on a 64-bit computer with 16Gb of memory. The table in Figure 5 shows the differences in compilation and analysis times and memory requirements between the two toolboxes. XFST performed immeasurably better than FSA. In particular, we were unable to use the complete FSA network for analysis, compared to analyzing 70 words per second with the full network in XFST. Another issue that should be noticed is the difference in memory requirements.

4 Conclusions

We presented a compiler which translates XFST grammars to grammars in the language of the FSA Utilities package. This work allows a closer look into two of the most popular finite state toolboxes. Although FSA has the advantage of being a publicly available software, we discovered that it does not scale up as well as XFST. However, for the non-expert user or for teaching purposes, where more modest networks are manipulated, FSA seems to be more friendly, especially with regard to graphical representation of networks. With our new implementation of replace rules in FSA, it seems that for such uses FSA is better. However, for larger systems and when time performance is an issue, XFST is preferable.

This work can be extended in several directions. Not all XFST operators are implemented in FSA. Some for lack of documentation and some simply require more time for implementation. Thus, further work can be done to construct more operators (see footnote 3). We believe that replace rules still hide boundary cases which require special treatment. More work is needed in order to locate such cases. Furthermore, other finite state toolboxes exist (Mohri, 1996) which present different operators. Extending the compiler to convert XFST grammars into those formalisms will provide opportunities for better comparison of different finite-state toolboxes. On a different course, an fsa2xfst compiler can be constructed. Such a compiler will enable a reverse performance comparison, i.e. using a larger network for FSA and making it operational in XFST. Notice that in contrast to the xfst2fsa direction, this course is rather trivial: FSA allows the user to save its networks in a readable format (listing the network states and arcs). Although XFST is not capable of reading any format but its own, Kleene (1954) presents a simple algorithm for generating from a given FSA a regular expression denoting it. Using this algorithm, an XFST regular expression denoting the network

		FSA		XFST	
		Time	Memory	Time	Memory
Compilation	complete network	13h 43m	≈11G	27m 41s	≈3G
	nouns network	2h 29m		11m 4s	
	adjectives network	14m 56s		8m 21s	
Analysis	complete network, 350 words	not possible		5s	
	nouns network, 120 nouns	1h 50m		0.17s	
	adjectives network, 50 adjectives	2m 34s		0.17s	

Figure 5: Times and memory requirements

can be generated. The only disadvantage of such an approach is that the resulting XFST expression will be most cumbersome.

Acknowledgments

This research was supported by The Israel Science Foundation (grant no. 136/01). We are grateful to Gertjan van Noord and Shlomo Yona for their help.

References

Kenneth R. Beesley and Lauri Karttunen. 2003. *Finite-State Morphology*. CSLI Publications.

Dale Gerdemann and Gertjan van Noord. 1999. Transducers from rewrite rules with backreferences. In *Ninth Conference of the European Chapter of the Association for Computational Linguistics*, pages 126–133, Bergen Norway.

Dale Gerdemann and Gertjan van Noord. 2000. Approximation and exactness in finite state optimality theory. In *Proceedings of Sigphon Workshop on Finite State Phonology (invited paper)*, pages 34–45, Luxembourg. http://xxx.lanl.gov/ps/cs.CL/0006038 or ROA-403-08100 at http://ruccs.rutgers.edu/roa.html.

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

Lauri Karttunen and Andre Kempe. 1995. The parallel replacement operation in finite state calculus. Technical Report 1995-021, Rank Xerox research centre – Grenoble laboratory.

Lauri Karttunen, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schiller. 1996. Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–328.

Lauri Karttunen. 1995. The replace operator. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics*, pages 16–24.

Lauri Karttunen. 1996. Directed replacement. In *Proceedings of ACL'96*, pages 108–115.

Lauri Karttunen. 1997. The replace operator. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*, Language, Speech and Communication, chapter 4, pages 117–147. MIT Press, Cambridge, MA.

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.

S. C. Kleene. 1954. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–42. Princeton University Press.

Kimmo Koskenniemi. 1983. *Two-Level Morphology: a General Computational Model for Word-Form Recognition and Production*. The Department of General Linguistics, University of Helsinki.

Mehryar Mohri and Richard Sproat. 1996. An efficient compiler for weighted rewrite rules. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pages 231–238, Santa Cruz.

Mehryar Mohri. 1996. On some applications of finite-state automata theory to natural language processing. *Natural Language Engineering*, 2(1):61–80.

Fernando Pereira and Michael Riley. 1997. Speech recognition by composition of weighted finite automata. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*, pages 431–453. MIT Press, Cambridge.

Nathan Vaillette. 2004. *Logical Specification of Finite-State Transductions for Natural Language Processing*. Ph.D. thesis, Ohio State University.

Gertjan van Noord and Dale Gerdemann. 2001a. An extendible regular expression compiler for finite-state approaches in natural language processing. In O. Boldt and H. Jürgensen, editors, *Automata Implementation*, number 2214 in Lecture Notes in Computer Science. Springer.

Gertjan van Noord and Dale Gerdemann. 2001b. Finite state transducers with predicates and identity. *Grammars*, 4(3):263–286.

Gertjan van Noord, 2000. *FSA6 Reference Manual*.

Shlomo Yona and Shuly Wintner. 2005. A finite-state morphological grammar of hebrew. In *Proceedings of the ACL-2005 Workshop on Computational Approaches to Semitic Languages*.

Appendix

A A comparison table of XFST and FSA operators

A.1 Symbols

XFST syntax	FSA syntax	Meaning
a	a	single symbol a
$\%*$ or "*"	escape(*) or '*'	escape literal symbol
abc	abc	multi-character symbol
?	?	any symbol
0 or [ ] or " "	[ ]	epsilon symbol, the empty string
$\{abcd\}$	$[a,b,c,d]$	single character brace

A.2 Basic operators

XFST syntax	FSA syntax	Meaning
$A*$	$A*$	Kleene star
$A+$	$A+$	iteration (Kleene plus)
$A \mid B$	$\{A,B\}$	union
$A \,\&\, B$	$A \,\&\, B$	intersection
$A-B$	$A-B$	A minus B
$\backslash A$	$[\sim A] \,\&\, ?$	term complement
$\sim A$	$\sim A$	complement
A/B	$ignore(A,B)$	A ignoring B
$A./.B$	$ignore(A,B) - \{[B,?*],[?*,B]\}$	A ignoring internally B
$\$A$	$\$A$	containment
$\$.A$	$\$A - ignore([A,A],?*)$	one containment
$\$?A$	$\{\$A - ignore([A,A],?*),?* - \$A\}$	maximum one containment
$A\ B$	$[A,B]$	concatenation
$A\hat{}n$		n-ary concatenation
$A\hat{}\{n,k\}$		n to k concatenations of A
$A\hat{} > n$		more than n concatenations of A
$A\hat{} < n$		less than n concatenations of A
$A.x.B$	$A\ x\ B$	crossproduct
$A.o.B$	$A\ o\ B$	composition
(A)	A^	optionality
$a:b$	$a:b$	symbol pair
[ ]	( )	order control
$R.P.Q$	$\{R,(domain(Q)-domain(R))\ o\ Q\}$	upper-side priority union
$R.p.Q$	$\{R,Q\ o\ (range(Q)-range(R))\}$	lower-side priority union
$R.-u.Q$	$(domain(R)-domain(Q))\ o\ R$	upper-side minus
$R.-l.Q$	$R\ o\ (range(R)-range(Q))$	lower-side minus
$A<B$	$\sim \$[B,A]$	A before B
$A>B$	$\sim \$[A,B]$	A after B
$A.r$	$reverse(A)$	reverse
$R.u$ or $R.1$	$domain(R)$	upper language of the regular relation R
$R.l$ or $R.2$	$range(R)$	lower language of the regular relation R
$R.i$	$invert(R)$ or $inverse(R)$	regular relation inverse

A.3 Restriction

XFST restriction rules are not provided by FSA, nor did we implement them. Therefore, we only present their syntax in XFST.

- $A => L\_R$
- $A => L_1\_R_1, L_2\_R_2, \ldots, L_n\_R_n$

A.4 Replacement

XFST replace rules do not exist in FSA. We present implementation of most of them in FSA, based on (Karttunen, 1995; Karttunen, 1996; Karttunen, 1997; Karttunen and Kempe, 1995). XFST replace rules can be divided into 4 groups:

1. Unconditional replace rules (one rule with no context).
2. Unconditional parallel replace rules (several rules with no context that are performed at the same time).
3. Conditional replace rules (one rule and one condition).
4. Conditional parallel replace rules (several rules and/or several contexts).

A.4.1 $->$ (obligatory, upper to lower replacement)

- XFST syntax: $A -> B$

  Meaning: Unconditional replacement of the language A by the language B.

  Construction: [[NO_ A $[A\ .x.\ B]$]* NO_ A] where NO_ A abbreviates $\sim \$[\, A - [\,]\,]$.

- XFST syntax: $A_1 -> B_1, \ldots, A_n -> B_n$

  Meaning: Unconditional parallel replacement of the language A_1 by the language B_1 and the language A_2 by the language B_2 ... and the language A_n by the language B_n.

  Construction: $[\,[\,N\,R\,]*\,N\,]$ where N denotes the language of strings that do not contain any A_i:

  $$N = \sim \$[\, [A_1 \mid \ldots \mid A_n] - [\,]\,]$$

  and R stands for the relation that pairs every A_i with the corresponding B_i:

  $$R = [\, [A_1\ .x.B_1] \mid \ldots \mid [A_n\ .x.B_n]\,]$$

  .

- XFST syntax: $A -> B \,||\, L\_R$

  Meaning: Conditional replacement of the language A by the language B. This is like the relation $A -> B$ except that any instance of A in the upper string corresponds to an instance of B in the lower string only when it is preceded by an instance of L and followed by an instance of R. Other instances of A in the upper string remain unchanged. A, B, L, and R must all denote simple languages, not relations. The slant of the double bars indicates whether the precede/follow constraints refer to the instance of A in the upper string or to its image in the lower string. In the $||$ version, both contexts refer to the upper string.

  Construction: *InsertBrackets .o. ConstrainBrackets .o. LeftContext .o. RightContext .o. Replace .o. RemoveBrackets* where:

- Let $<$ and $>$ be two symbols not in Σ. The escape character % is used since $<$ and $>$ are saved symbols in XFST.
- $InsertBrackets = [\,[\,] < - \,\% < \mid \% > \,]$
  InserBrackets eliminates from the upper side language all context markers that appear on the lower side.
- $ConstrainBrackets = [\sim \$[\% < \;\; \% >]\,]$
  ConstrainBrackets denotes the language consisting of strings that do not contain $<>$ anywhere.
- $LeftContext = [\sim [\sim [...LEFT]\;\; [< ...]\,] \,\&\, \sim [\,[...LEFT]\;\; \sim [< ...]\,]\,]$
  LeftContext denotes the language in which any instance of $<$ is immediately preceded by LEFT and every LEFT is immediately followed by $<$, ignoring irrelevant brackets. $[...LEFT]$ denotes $[\,[\,?* \; L/[\% < \mid \% >]\,] - [?* \; \% <]\,]$, the language of all strings ending in L, ignoring all brackets except for a final $<$. $[< ...]$ denotes $[\% < /\% > ?*]$, the language of strings beginning with $<$, ignoring the other bracket.
- $RightContext = [\sim [\,[... >]\;\; \sim [RIGHT...]\,] \,\&\, \sim [\sim [... >]\;\; [RIGHT...]\,]\,]$
  RightContext denotes the language in which any instance of $>$ is immediately followed by RIGHT and any RIGHT is immediately preceded by $>$, ignoring irrelevant brackets. $[RIGHT...]$ denotes $[\,[\,R/[\% < \mid \% >]\;\; ?*\,] - [\% < ?*]\,]$, the language of all strings beginning with R, ignoring all brackets except for an initial $>$. $[... >]$ denotes $[?* \; \% > /\% <]$, the language of strings ending with $>$, ignoring the other bracket.
- The definition of Replace divides into three cases:
  1. If A does not contain the empty string (epsilon) then

     $$Replace = [\% < \;\; A/[\% < \mid \% >]\;\; \% > \;\; - > \;\; \% < \;\; B/[\% < \mid \% >]\;\; \% >]$$

     This is the unconditional replacement of $< A >$ by $< B >$, ignoring irrelevant brackets.
  2. If A is the empty string (i.e., $A = \varepsilon$) then

     $$Replace = [\% > \;\; \% < \;\; - > \;\; \% < \;\; B* \;\; \% >]$$

  3. If A contains the empty string but is not equal to it (i.e., contains other strings too) then

     $$Replace =$$

     $$[\% < A/[\% <\mid \% >]\;\% > \;\; - > \;\; \% < \;\; B/[\% <\mid \% >]\;\% >]\;,\; \% > \;\; \% < \;\; - > \;\; \% < \;\; B* \;\; \% >$$

     That is, the first two cases are performed in parallel.
- $RemoveBrackets = [\,\% < \mid \% > \;\; - > \;\; [\,]\,]$. RemoveBrackets denotes the relation that maps the strings of the upper language to the same strings without any context markers.

- XFST syntax: $A - > B \;//\; L \_ R$

  Meaning: Conditional replacement of the language A by the language B. This is like the relation $A - > B \;||\; L \_ R$ except that the // variant requires the left context on the lower side of the replacement and the right context on the upper side.

  Construction: $InsertBrackets \;.o.\; ConstrainBrackets \;.o.\; RightContext \;.o.\; Replace \;.o.\; LeftContext \;.o.\; RemoveBrackets$ where InsertBrackets, ConstrainBrackets, RightContext, Replace, LeftContext and RemoveBrackets are the same as above.

- XFST syntax: $A -> B \setminus\setminus L \_ R$

  Meaning: Conditional replacement of the language A by the language B. This is like the relation $A -> B \,||\, L \_ R$ except that the $\setminus\setminus$ variant requires the left context on the upper side of the replacement and the right context on the lower side.

  Construction: *InsertBrackets .o. ConstrainBrackets .o. LeftContext .o. Replace .o. RightContext .o. RemoveBrackets* where InsertBrackets, ConstrainBrackets, RightContext, Replace, LeftContext and RemoveBrackets are the same as above.

- XFST syntax: $A -> B \setminus/ L \_ R$

  Meaning: Conditional replacement of the language A by the language B. This is like the relation $A -> B \,||\, L \_ R$ except that in the $\setminus/$ variant, both contexts refer to the lower string.

  Construction: *InsertBrackets .o. ConstrainBrackets .o. Replace .o. LeftContext .o. RightContext .o. RemoveBrackets* where InsertBrackets, ConstrainBrackets, RightContext, Replace, LeftContext and RemoveBrackets are the same as above.

The rest of the obligatory upper to lower replace rules are conditional parallel replace rules that where not implemented. An example of such a rule is $A_{11} -> B_{11}, \ldots, A_{1n} -> B_{1n} \,||\, L_{11} \_ R_{11}, \ldots, L_{1m} \_ R_{1m}$.

A.4.2 $(->)$ (optional, upper to lower replacement)

- XFST syntax: $A(->)B$

  Construction: [[?* [*A .x. B*]]* ?*].

- XFST syntax: $A_1(->)B_1, \ldots, A_n(->)B_n$

  Construction: [[?* *R*]* ?*] where R stands for the relation that pairs every A_i with the corresponding B_i: $R =$ [[A_1 *.x.*B_1] | ... | [A_n *.x.*B_n]].

- XFST syntax: $A(->)B \,||\, L \_ R, A(->)B \,//\, L \_ R, A(->)B \setminus\setminus L \_ R, A(->)B \setminus/ L \_ R$

  Construction: The same as the construction for the corresponding rules with the operator $->$ with the difference that in the *Replace* stage, for each one of the three cases the obligatory upper to lower replace operator $->$ is replaced by the optional upper to lower replace operator $(->)$.

The rest of the optional upper to lower replace rules are conditional parallel replace rules and were not implemented.

A.4.3 $<-$ (obligatory, lower to upper replacement)

- XFST syntax: $A <- B$

  Construction: $[B -> A].i$

- XFST syntax: $A_1 <- B_1, \ldots, A_n <- B_n$

  Construction: $[\, B_1 -> A_1, \ldots, B_n -> A_n \,].i$

- XFST syntax: $A <- B \,||\, L \_ R, A <- B \,//\, L \_ R, A <- B \setminus\setminus L \_ R, A <- B \setminus/ L \_ R$

  Construction: The same as the construction for the corresponding rules with the operator $->$ with the difference that in the *Replace* stage, for each one of the three cases the obligatory upper to lower replace operator $->$ is replaced by the operator $<-$.

The rest of the obligatory lower to upper replace rules are conditional parallel replace rules and were not implemented.

A.4.4 $(<-)$ (optional, lower to upper replacement)

- XFST syntax: $A(<-)B$

  Construction: $[B(->)A].i$

- XFST syntax: $A_1(<-)B_1,\ldots,A_n(<-)B_n$

  Construction: $[\,B_1(->)A_1,\ldots,B_n(->)A_n\,].i$

- XFST syntax: $A(<-)B \mid\mid L\_R, A(<-)B \,//\, L\_R, A(<-)B \setminus\setminus L\_R, A(<-)B \setminus/ L\_R$

  Construction: The same as the construction for the corresponding rules with the operator $->$ with the difference that in the *Replace* stage, for each one of the three cases the obligatory upper to lower replace operator $->$ is replaced by the operator $(<-)$.

The rest of the optional lower to upper replace rules are conditional parallel replace rules and were not implemented.

A.4.5 $<->$ (obligatory, upper to lower, lower to upper replacement)

- XFST syntax: $A <-> B$

  Construction: Let @ be a character not in Σ. We use the escape character % to precede @, since @ is a reserved character in XFST. Thus, $A <-> B$ is defined as

$$\sim \$[\%@]$$
$$.o.$$
$$A -> \%@$$
$$.o.$$
$$\%@ <- B$$
$$.o.$$
$$\sim \$[\%@]$$

- XFST syntax: $A_1 <-> B_1,\ldots,A_n <-> B_n$

  Construction: Let $@1,\ldots,@n$ be characters not in Σ. We use the escape character % to precede each $@i$, since @ is a reserved character in XFST. Thus, $A <-> B$ is defined as

$$\sim \$[\%@1 \mid \ldots \mid \%@n]$$
$$.o.$$
$$A_1 -> \%@1\ ,\ \ldots\ ,\ A_n -> \%@n$$
$$.o.$$
$$\%@1 <- B_1\ ,\ \ldots\ ,\ \%@n <- B_n$$
$$.o.$$
$$\sim \$[\%@1 \mid \ldots \mid \%@n]$$

The rest of the obligatory upper to lower and lower to upper replace rules are conditional replace rules and they were not implemented.

A.4.6 $(<->)$ (optional, upper to lower, lower to upper replacement)

- XFST syntax: $A(<->)B$

  Construction: Let @ be a character not in Σ. We use the escape character % to precede @, since @ is a reserved character in XFST. Thus, $A(<->)B$ is defined as

$$\sim \$[\%@]$$
$$.o.$$
$$A\ (->)\ \%@$$
$$.o.$$
$$\%@\ (<-)\ B$$
$$.o.$$
$$\sim \$[\%@]$$

The rest of the optional upper to lower and lower to upper replace rules are conditional and parallel replace rules and they were not implemented.

A.4.7 $@->$ (obligatory, upper to lower, left to right, longest match replacement)

The following operations are the same as in section A.4.1, except that instead of $->$ occurs $@->$. As $->$ represented an obligatory upper to lower replacement, $@->$ represents an obligatory upper to lower left to right longest match replacement. Instances of the language A on the upper side of the relation are replaced selectively. The selection factors each upper language string uniquely into A and non-A substrings. The factoring is based on a left-to-right and longest match regimen. The starting locations are selected from left-to-right. If there are overlapping instances of A starting at the same location, only the longest one is replaced by all strings of B.

- XFST syntax: $A@->B$

  Construction: $InitialMatch\ .o.\ LeftToRight\ .o.\ LongestMatch\ .o.\ Replacement$

  Where:

  - Let $\hat{}, <, >$ be characters not in Σ. We use the escape character % to precede them since they are reserved characters in XFST.
  - $InitialMatch =$

$$\sim \$[\ \%\hat{}\ |\ \%<\ |\ \%>\ ]$$
$$.o.$$
$$[\ .\ .]\ ->\ \%\hat{}\ ||\ \_A$$

    where $[\ .\ .]\ ->\ LOWER\ ||\ LEFT\_RIGHT$ is a version of empty string replacement that allows only one application between any LEFT and RIGHT. The construction for $[\ .\ .]\ ->\ LOWER\ ||\ LEFT\_RIGHT$ is the same as for $UPPER\ ->\ LOWER\ ||\ LEFT\_RIGHT$ except that $Replace = [\%>\ \%<\ ->\ \%<\ LOWER\ \%>]$.

  - $LeftToRight =$

$$[\ \sim \$[\%\hat{}]\ [\%\hat{}:\%<\ \ UPPER'\ \ 0:\%>\ ]\ ]*\ \sim \$[\%\hat{}]$$
$$.o.$$
$$\%\hat{}\ ->\ [\ ]$$

    where $UPPER' = [\ A/[\%\hat{}] - [?*\ \%\hat{}]\ ]$

- $LongestMatch = \sim \$[\ \%<\ [\ UPPER'' \ \&\ \$[\%>]\ ]\ ]$
  where $UPPER'' = A/[\%<\ |\ \%>]\ -\ [\ ?*\ [\ \%<\ |\ \%>]\ ]$
- $Replacement =\ \%<\sim \$[\%>]\ \%>\ ->\ B$

The rest of the obligatory upper to lower left to right longest match replace rules are conditional and parallel replace rules and they were not implemented.

The following XFST operators were not implemented:

- @ > (obligatory, upper to lower, left to right, shortest match replacement)
- − > @ (obligatory, upper to lower, right to left, longest match replacement)
- > @ right (obligatory, upper to lower, right to left, shortest match replacement)

A.5 Markup

Markup rules take an input string and mark it by inserting some strings before and after it. XFST markup rules do not exist in FSA. We present the implementation of most of them in FSA, based on Karttunen (1996).

- XFST syntax: $A->L\ldots R$

  Meaning: Markup. Instances of the language A on the upper side of the relation are selected for markup. Each selected A string is marked by inserting all strings of L to its left and all strings of R to its right. The selected A strings themselves remain unchanged, along with the non-A segments.

  Construction: $A\ ->\ L\,A\,R$

- XFST syntax: $A@->L\ldots R$

  Meaning: Directed markup. Instances of the language A on the upper side of the relation are selected for markup under left-to-right, longest match regimen. Thus, the starting locations are selected from left-to-right. If there are overlapping instances of A starting at the same location, only the longest one is replaced by all strings of B. Each selected A string is marked by inserting all strings of R to its left and all strings of S to its right. The selected A strings themselves remain unchanged, along with the non-A segments.

  Construction: $InitialMatch\ .o.\ LeftToRight\ .o.\ LongestMatch\ .o.\ Insrtion$

  Where:

  - Let $\hat{},<,>$ be characters not in Σ. We use the escape character % to precede them since they are reserved characters in XFST.
  - $InitialMatch =$

    $$\sim \$[\ \%\hat{}\ |\ \%<\ |\ \%>\ ]$$

    $$.o.$$

    $$[\ .\ .]\ ->\ \%\hat{}\ ||\ \_A$$

    where $[\ .\ .]\ ->\ LOWER\ ||\ LEFT\_RIGHT$ is a version of empty string replacement that allows only one application between any LEFT and RIGHT. The construction for $[\ .\ .]\ ->\ LOWER\ ||\ LEFT\_RIGHT$ is the same as for $UPPER\ ->\ LOWER\ ||\ LEFT\_RIGHT$ except that $Replace = [\%>\ \%<\ ->\ \%<\ LOWER\ \%>]$.

– $LeftToRight =$

$$[\sim \$[\%\hat{}]\ [\%\hat{}:\%<\ \ UPPER'\ \ 0:\%>\ ]\ ]* \sim \$[\%\hat{}]$$

$$.o.$$

$$\%\hat{} -> [\,]$$

where $UPPER' = [\,A/[\%\hat{}] - [?* \%\hat{}]\,]$

– $LongestMatch = \sim \$[\,\%< \ [\,UPPER'' \,\&\, \$[\%>]\,]\,]$
where $UPPER'' = A/[\%< \mid \%>] \ - \ [\,?* \,[\,\%< \mid \%>]\,]$

– $Insrtion = \ \%< \ -> \ L\,,\ \%> \ -> \ R$

The rest of the markup rules were not implemented since we could not obtain any documentation of their constructions. These operators are:

- XFST syntax: $A@>L\ldots R$

  Meaning: Directed markup. Instances of the language A on the upper side of the relation are selected for markup under left-to-right, shortest match regimen. Thus, the starting locations are selected from left-to-right. If there are overlapping instances of A starting at the same location, only the shortest one is replaced by all strings of B. Each selected A string is marked by inserting all strings of R to its left and all strings of S to its right. The selected A strings themselves remain unchanged, along with the non-A segments.

- XFST syntax: $A->@L\ldots R$

  Meaning: Directed markup. Instances of the language A on the upper side of the relation are selected for markup under right-to-left, longest match regimen. Thus, the starting locations are selected from right-to-left. If there are overlapping instances of A starting at the same location, only the longest one is replaced by all strings of B. Each selected A string is marked by inserting all strings of R to its left and all strings of S to its right. The selected A strings themselves remain unchanged, along with the non-A segments.

- XFST syntax: $A>@L\ldots R$

  Meaning: Directed markup. Instances of the language A on the upper side of the relation are selected for markup under right-to-left, shortest match regimen. Thus, the starting locations are selected from right-to-left. If there are overlapping instances of A starting at the same location, only the shortest one is replaced by all strings of B. Each selected A string is marked by inserting all strings of R to its left and all strings of S to its right. The selected A strings themselves remain unchanged, along with the non-A segments.

A.6 Boundary symbol for restriction and replacement

In the restriction, $=>$, and conditional replacement, $->, (->), <-, (<-), <->, (<->), @->, @>$ $, -> @, > @$ expressions we can use a special boundary marker, .#., to refer to the beginning or to the end of a string. In the left context, the boundary marker signals the beginning of the string; in the right context it means the end of the string.

Construction: We do not deal with all the cases where the boundary symbol .#. can be used. We only deal with boundary cases contexts that are in one of the following forms (*LeftContext* and *RightContext* are assumed not to contain .#.):

- .#. *LeftContext* _ *RightContext*

- $[.\#.\ LeftContext]\ \_\ RightContext$
- $[.\#.]\ LeftContext\ \_\ RightContext$
- $[[.\#.]\ LeftContext]\ \_\ RightContext$
- $LeftContext\ \_\ RightContext\ .\#.$
- $LeftContext\ \_\ [RightContext\ .\#.]$
- $LeftContext\ \_\ RightContext\ [.\#.]$
- $LeftContext\ \_\ [RightContext\ [.\#.]]$
- $.\#.\ LeftContext\ \_\ RightContext\ .\#.$
- $[.\#.\ LeftContext]\ \_\ RightContext\ .\#.$
- $[.\#.]\ LeftContext\ \_\ RightContext\ .\#.$
- $[[.\#.]\ LeftContext]\ \_\ RightContext\ .\#.$
- $.\#.\ LeftContext\ \_\ [RightContext\ .\#.]$
- $[.\#.\ LeftContext]\ \_\ [RightContext\ .\#.]$
- $[.\#.]\ LeftContext\ \_\ [RightContext\ .\#.]$
- $[[.\#.]\ LeftContext]\ \_\ [RightContext\ .\#.]$
- $.\#.\ LeftContext\ \_\ RightContext\ [.\#.]$
- $[.\#.\ LeftContext]\ \_\ RightContext\ [.\#.]$
- $[.\#.]\ LeftContext\ \_\ RightContext\ [.\#.]$
- $[[.\#.]\ LeftContext]\ \_\ RightContext\ [.\#.]$
- $.\#.\ LeftContext\ \_\ [RightContext\ [.\#.]]$
- $[.\#.\ LeftContext]\ \_\ [RightContext\ [.\#.]]$
- $[.\#.]\ LeftContext\ \_\ [RightContext\ [.\#.]]$
- $[[.\#.]\ LeftContext]\ \_\ [RightContext\ [.\#.]]$

As we do not deal with restriction rules we need to deal with boundary cases only in replace rules. The replace rules were constructed from six stages: InsertBrackets, ConstrainBrackets, LeftContext, RightContext, Replace and RemoveBrackets. In boundary cases where the left context is in the beginning of a string, only the LeftContext stage is changed. The LeftContext stage was defined as

$$LeftContext = [\sim[\sim[...LEFT]\ [<...]\ ]\ \&\ \sim[\ [...LEFT]\ \ \sim[<...]\ ]\ ]$$

where $[...LEFT]$ denoted

$$[\ [\ ?*\ L/[\%<\ |\ \%>]\ ]-[?*\ \%<]\ ]$$

and [< ...] denoted

$$[\% < / \% > ?*]$$

The definition of LeftContext is not changed but the definition of $[...LEFT]$ is changed into

$$[\ [\ L/[\% <\ |\ \% >]\ ] - [?*\ \% <]\ ]$$

In boundary cases where the right context is at the end of a string, only the RightContext stage is changed. The RightContext stage was defined as

$$RightContext = [\ \sim [\ [... >]\ \sim [RIGHT...]\ ]\ \&\ \sim [\ \sim [... >]\ [RIGHT...]\ ]\ ]$$

where $[RIGHT...]$ denoted

$$[\ [\ R/[\% <\ |\ \% >]\ ?*\ ] - [\% < ?*]\ ]$$

and [... >] denoted

$$[?*\ \% > / \% <]$$

The definition of RightContext is not changed but the definition of $[RIGHT...]$ is changed into

$$[\ [\ R/[\% <\ |\ \% >]\ ] - [\% < ?*]\ ]$$

In boundary cases where both the right context and the left context are at the end and in the beginning of a string respectively, both the RightContext and the LeftContext stages are changed as described above. The idea behind these changes is that the context part of replacement expression can be actually seen as $?*\ LEFT\ \_\ RIGHT\ ?*$ and by simply eliminating one of the ?* in one of the ends we can relate to a boundary case. The definitions that were changed above did exactly that: eliminated the appropriate ?* for each case. More complicated cases, for example $a- > b\ ||\ [.\#.\ |\ a]\ \_$ should be dealt by conditional parallel replacement. For example, $a- > b\ ||\ [.\#.\ |\ a]\ \_$, should be interpreted as

$$a- > b\ ||\ [.\#.]\ \_\ ,,\ a- > b\ ||\ [a]\ \_$$

Since we do not deal with conditional parallel replacement, we cannot deal with these cases.

A.7 Order of Precedence

A.7.1 XFST

The following list defines the order of precedence of all XFST operators. Operators of same precedence are evaluated from left to right, except the prefix operators ($\sim$ \ $ $? $.) that are evaluated from right to left. The list begins with the operators of highest precedence, i.e., with the most tightly binding ones. Operators of same precedence are on the same line.

:
$\sim$ \ $ $? $.
+ * ^ .1 .2 .u .l .i .r
/
concatenation
> <
| & −
=> −> (−>) <− (<−) <−> (<−>) @−> @> −>@ >@
.x. .o.

A.7.2 FSA

The following list defines the order of precedence in FSA:

: /
..
+ * ^
& −
o x xx
! #

A.8 Advanced techniques

Both XFST and FSA have advanced techniques that do no exist in the other toolbox. For XFST these techniques include Compile-Replace and Flag-Diacritics; for FSA these techniques include predicates and weighted networks.