SFST Manual

Helmut Schmid

Institute for Natural Language Processing
University of Stuttgart
schmid@ims.uni-stuttgart.de

1 Introduction

The Stuttgart Finite State Transducer (SFST) tools are a collection of software tools for the generation, manipulation and processing of finite-state automata and transducers. A finite state transducer (FST) maps strings from one regular language (surface language) onto strings from another regular language (analysis language). One important application of FSTs is morphological analysis, where a word form such as *translations* might be mapped to the analysis string *translate<V>ion<N><pl>*. The mapping between surface strings and analysis strings is reversible. The same transducer can be used to generate (i) analyses for a surface form (in analysis mode) and (ii) to generate surface forms for an analysis (in generation mode). The number of generated output strings is non necessarily 1, but can be anywhere between 0 and infinite.

Other important applications of FSTs are lemmatization, tokenization, lexicon representation, spell checking, grapheme to phoneme conversion, low-level parsing, and much more.

The SFST tools comprise

- a programming language for the implementation of finite state transducers called "SFST-PL"
- a compiler for SFST programs called fst-compiler
- a set of tools for applying, printing, and comparing finite state transducers and
- a finite state transducer library written in C++

The implementation of the compiler and the other tools is based on the C++ library. The following description of the SFST tools assumes some familiarity with finite state transducers.

2 The SFST Programming Language

SFST transducers are specified by means of the SFST programming language. Any regular expression is already a valid SFST program. An example is the expression:

```
Hello\ world\!
```

It defines a transducer which maps the string *Hello world!* onto itself and rejects any other input. The blank and the exclamation mark have to be quoted with a backslash because unquoted blank and tab characters are ignored and unquoted exclamation marks are interpreted as negation operators.

Because the output of the previous transducer is always identical to its input, it is equivalent to a finite state automaton. The following expression specifies a real transducer which (in generation mode) maps a string of a's, b's, and c's to a string where the c's are unchanged and the a's have been replaced with b's and vice versa. The string *abcba*, for instance, would be mapped to *bacab*.

```
(a:b | b:a | c:c)*
```

This expression uses the union operator (|) and Kleene's star operator (*). The colon operator separates an input symbol and an output symbol. *a:b* maps the input symbol *a* to *b* in generation mode and *b* to *a* in analysis mode. The expression `c:c` could have been abbreviated to `c`.

Note that the analysis level is viewed as the lower level and the surface level as the upper level in SFST. (Other formalism sometimes visualize the surface as the lower level.)

A less trivial mapping is performed by the following transducer for the inflection of the nouns *foot*, *house*, and *mouse*:

```
(house | foot | mouse) <N>:<> <sg>:<> |\
(house<>:s | f o:e o:e t | {mouse}:{mice}) <N>:<> <pl>:<>
```

Strings which are enclosed in angle brackets (such as `<N>`, `<sg>`, and `<pl>` in the previous example) are multi-character symbols. They are treated in the same way as single-character symbols. '`<>`' is a special symbol representing the empty string. The expression '`house<>:s`' therefore maps *house* to *houses* (in generation mode). '`fo:eo:et`' specifies a transducer which maps *foot* to *feet*. The expression `{mouse}:{mice}` is equivalent to `m:mo:iu:cs:ee:<>` and maps *mouse* to *mice*.

Regular expressions terminate at the end of a line unless the end of the line is quoted with a backslash. This was the reason for adding a backslash at the end of the first line in the previous example.

Variables

An SFST program is essentially a regular expression. This expression may be quite complex for transducers which perform sophisticated tasks. Therefore we use variables to build complex expressions from smaller units. The preceding program, for instance, can be reformulated as follows:

```
$Nsg$ = house | foot | mouse
$Npl$ = house<>:s | f o:e o:e t | {mouse}:{mice}
$Nsg$ <N>:<> <sg>:<> | $Npl$ <N>:<> <pl>:<>
```

The first two lines define the variables `$Nsg$` and `$Npl$`. Variable names begin and end with a dollar sign. The right-hand side of a variable definition is any valid regular expression (see the next section).

There is a second type of variables for character ranges. They start and end with the symbol '#'. The following program defines a variable for lower-case characters, a variable for upper-case characters and a transducer which maps strings of lower-case characters to the corresponding strings of upper-case characters (in generation mode).

```
#LC# = a-z
#UC# = A-Z
[#LC#]:[#UC#]*
```

SFST programs usually consist of a long sequence of variable definitions followed by a single expression which specifies the result transducer.

Agreement Variables

Variables whose name starts with the symbol "=" are called *agreement variables*. They are treated specially by the compiler: if an expression contains several instances of the same agreement variable, their values are correlated. Consider the following example program:

```
$=c$ = [abc]
$=c$ X $=c$
```

The result transducer for this program maps the strings aXa, bXb, and cXc onto themselves. Only acyclic transducers (i.e. transducers with a finite set of strings mappings) can be assigned to agreement variables.

There are also agreement variables for character ranges. The following transducer maps aX to aXa, bX to bXx, and cX to cXc (in generation mode).

```
#=c# = abc
[#=c#] X <>:[#=c#]
```

Basic Regular Expressions

The transducer expressions are defined over a set of **symbol pairs**. Symbols are specified in one of the following ways:

- single characters such as `'A'` or `'á'`
- quoted characters such as '`\ `' or '`\!`'
- quoted numbers such as '`\32`' which are translated to the character with the respective code ('`\32`' translates to the blank character.)
- multi-character symbols such as `<N>` or `<k$ln_5>` which are enclosed in angle brackets
- the special symbol `<>` which designates the empty string

The following expressions are examples of **basic regular expressions**:

a:b defines a transducer which maps the symbol **a** to the symbol **b**.

a is identical to **a:a**

a:. maps the symbol **a** to any symbol *b* for which the symbol pair *a:b* is an element of the alphabet. (The alphabet is described below.)

. is identical to **.:.**. The transducer defined by this expression performs any mapping of a single symbol allowed by the alphabet.

[abc]:[de] is identical to **a:d** | **b:e** | **c:e** ("|" is the or-operator which is introduced below.)

[a-d]:[A-C] is identical to **[abcd]:[ABC]**.

[abc] is a short form of **[abc]:[abc]**.

[^abc] maps any symbol other than a, b, or c onto itself. I.e. [^abc] is the complement of [abc] with respect to the alphabet.

{abc}:{de} is identical to **a:d b:e c:<>** This expression maps **abc** to **de**.

var the value of a variable `$var$` is the expression which was previously assigned to it (see below).

Operators

More complex expressions are built by combining transducer expressions with operators. SFST-PL supports the following set of operators:

rs concatenation
If **r** maps the string α to β and **s** maps γ to δ, then **rs** maps the string $\alpha\gamma$ to $\beta\delta$.

r|s or-operator (union, disjunction)
r|s maps α to β iff **r** or **s** maps α to β.

r||s composition
r||s maps α to γ iff **r** maps α to some β and **s** maps β to γ.

r&s and-operator (intersection, conjunction)
r&s maps the string $a_1a_2...a_n$ to $b_1b_2...b_n$ iff both **r** and **s** map $a_1a_2...a_n$ to $b_1b_2...b_n$ by aligning a_i with b_i for all $1 \leq i \leq n$. a_i and b_i are either a single character, a single multi character symbol or the empty symbol <>.
Note: Although **a:b** and **a:<><>:b** both map the string **a** to **b**, the result of **a:b & (a:<><>:b)** is nevertheless empty because the alignment is different.

!r Negation (complement)
!r maps the string $a_1a_2...a_n$ to $b_1b_2...b_n$ iff the alphabet contains a_i:b_i for all $1 \leq i \leq n$, and $r\&(a_1$:b_1a_2:$b_2...a_n$:$b_n)$ is the empty transducer. Either a_i or b_i is allowed to be the empty symbol.

r–l minus (subtraction, difference, relative complement)
`l-r` is equivalent to `l & !r`.

r? optionality
identical to **<>|r**

r* Kleene's star operator
if **r** maps α to β, then **r*** maps n repetitions of α to n repetitions of β for $n = 0, 1, 2, ...$

r+ Kleene's plus
identical to **r r***

^r range (extraction of the surface language)
maps β to β iff **r** maps some string α to β.
Note: SFST considers the surface language as the upper language and the analysis language as the lower language!

_r domain (extraction of the analysis/deep language)
maps α to α iff **r** maps α to some string β.

^_r inversion
maps α to β iff **r** maps β to α.

r<<x insertion
freely inserts the symbol pair **x** into the transducer **r**. The expression `ab << <x>`, for instance, is equivalent to `<x>*a<x>*b<x>*`.

SFST supports two-level rules. However, the syntax differs slightly from that in **?** and multiple contexts are not supported.

l a <= b r obligatorily maps the symbol **a** to **b** if **l** precedes and **r** follows. (Elsewhere, the mapping of **a** to **b** is optional. **l** and **r** are arbitrary regular expressions.)
This expression is identical to **!(.* l (a:. & !a:b) r .*)**
Note that the alphabet must contain the pair a:b here.

l a => b r allows the mapping of symbol **a** to **b** only if **l** precedes and **r** follows. (The mapping of **a** to **b** is optional in this context.)
The expression is equivalent to **!(!(.* l) a:b .* | .* a:b !(r .*))**

l a <=> b r maps the symbol **a** to **b** if and only if **l** precedes and **r** follows.
The expression is identical to **(l a => b r) & (l a <= b r)**

The single characters `a` and `b` in a two-level rule may be replaced by character ranges such as **[a-zäöü]**. The left and right context of a two-level rule should be surrounded by parentheses in order to ensure correct interpretation by the compiler.

The SFST formalism comprises replace operators similar to those described in **?**. (Note the two underscores between the left and right context!)

c ^–> l__r upward replacement
Each substring **s** of the input string which is in the analysis/deep language of the transducer **c** and whose left context is matched by the expression **.*l** and whose right context is matched by **r.*** is mapped to the respective surface strings defined by transducer **c**. Any other character is mapped to the characters specified in the alphabet. The left and right contexts must be automata (i.e. transducers which map strings onto themselves).
The rule `{aa}:{bb} ^-> c__c`, for instance, maps the string **caacac** to **cbbcac** in generation mode.
Note that the alphabet must contain the characters *a* and **b**, but not the pair *a:b* (unless you want wo allow this replacement everywhere in the context).

c _–> l__r downward replacement
Each substring **s** of the input string which is in the surface language of **c** and whose left context is matched by the expression **.*l** and whose right context is matched by **r.*** is mapped to the respective analysis strings defined by **c**.

c /–> l__r rightward replacement
The left context **l** must match the left surface context and the right context **r** must match the right analysis context.

c \–> l__r leftward replacement
The left context **l** must match the left analysis context and the right context **r** must match the right surface context.

If a replace operator is followed by a question mark (?), the replacement becomes optional. Note that replace operations (unlike the two-level rules) have to be combined by composition rather than intersection!

Furthermore, the SFST tools support some of the restriction and coercion operators defined in **?**. These operators can also be used with multiple contexts which are separated by commas.

c => l__r restriction operator
This operator allows any (substring) mapping defined by the transducer *c* only if it occurs in the context *l* and *r*. Symbols outside of the matching substrings are mapped to any symbol allowed by the alphabet.

c <= l__r coercion operator
This operator requires that one of the mappings defined by the transducer *c* must occur in each context *l* and *r*.

c <=> l__r restriction and coercion
This operator is equivalent to the intersection `c => l__r & c <= l__r` and requires that the mappings defined by *c* occur always and only in the given contexts.

c ^=> l__r surface restriction operator
This operator specifies that a string from the analysis language of the transducer *c* may only be mapped to one of its surface strings (according to transducer *c*) if it appears in the context *l* and *r*.

c ^<= l__r surface coercion operator
This operator specifies that a string from the source language of the transducer *c* always has to the mapped to one of its target strings according to transducer *c* if it appears in some context *l* and *r*.

c ^<=> l__r surface restriction and coercion
equivalent to the intersection `c ^=> l__r & c ^<= l__r`.

c _=> l__r deep restriction operator
This operator specifies that a string from the target language of the transducer *c* may only be mapped (in analysis direction) to one of its source strings (according to transducer *c*) if it appears in the context *l* and *r*.

c _<= l__r deep coercion operator
This operator specifies that a string from the target language of the transducer *c* always has to the mapped to one of its source strings according to transducer *c* if it appears in the context *l* and *r*.

c _<=> l__r deep restriction and coercion
equivalent to the intersection `c \_=> l__r & c \_<= l__r`.

Finally, there are two commands which create transducers from files and one command which writes intermediate transducers to a file.

"file" lexicon reader
reads a text file named *file* and returns the union of the lines of the file (see below). Whitespace at the end of the line is ignored unless it is quoted.

"<file>" transducer reader
reads a precompiled transducer from a binary file named *file* and returns it (see below).

t >> "file" output operator
writes the transducer "t" to the file named *file*. *t* is any valid transducer expression.

The Alphabet

The alphabet contains a set of symbol pairs which is required for the interpretation of the wildcard symbol '.'. The following SFST program e.g. defines a transducer which maps a sequence of letters to the same sequence of letters, but with lower-case letters replaced by upper-case characters (in generation mode):

```
ALPHABET = [A-Z] [a-z]:[A-Z]
.*
```

The first line defines the alphabet. The right-hand side of this assignment is a transducer expression. The set of symbol pairs is obtained by (i) compiling the expression to a transducer and (ii) extracting the symbol pairs from all state transitions of the transducer.

The definition of an alphabet is also required by the negation operator, the two-level rules, and Yli-Jyrä's coercion and restriction operators.

Comments

Comments in SFST programs start with a percent character (`%`) and extend up to the end of the line. The following lines of code return the expression `abc`.

```
% This is a comment
abc % This is another comment
% This comment also extends up to the end of the line % cde
```

Lexica

The lexicon entries of morphological analyzers are usually stored in a separate file, one entry per line. SFST-PL provides the operator `"lexicon"` to read lexicon entries from a file called *lexicon*. The result of the operator is a union of all the lines from the file. If the file *lexicon* contains the following entries

```
house
mouse
foot
```

then the result of the operator `"lexicon"` is the expression `house|mouse|foot`. The file reader treats all characters literally except for :, \, and angle brackets which are part of a multi-character symbol. Blanks and tab characters are deleted at the end of a lexicon line because such whitespace is usually added unintentionally.

Include Command

Complex computer programs are usually stored in a set of files rather than a single file, and the compiler combines these files to a single program. The same can be done with SFST programs. The command `#include "file.fst"` instructs the compiler to insert the contents of the file *file.fst* at the current position.

SFST programs create complex transducers by combining simpler transducers. If the compilation of some component transducer is expensive and the respective source code is seldom modified, it is useful to *pre-compile* this transducer. To this end, a separate SFST program has to be written which implements the component transducer. This program is compiled and the resulting transducer is stored e.g. in a file named *inc.a*. The main program reads the precompiled transducer with the command `"<inc.a>"`.

A Simple Example

The following very simple example shows the implementation of an inflectional component for English adjectives such as *easy*, *late*, or *dark*. It will correctly analyze forms such as *easier*, *latest*, or *darkest* and produce the analyses `easy<ADJ><comp>`, `late<ADJ><sup>`, and `dark<ADJ><sup>`.

```
% Define the set of valid symbol pairs for the two-level rules.
% The symbol # is used to mark the boundary between the stem and
% the inflectional suffix. It is deleted here.
ALPHABET = [A-Za-z] y:i [e#]:<>

% Read the lexical items from a separate file
% each line of which contains a form like "dark"
```

```
$WORDS$ = "adj"

% Define a rule which replaces y with i
% if a morpheme boundary and an e follows
% easy#er -> easier
$R1$ = y<=>i (#:<> e)

% Define a rule which eliminates e before "#e"
% late#er -> later
$R2$ = e<=><> (#:<> e)

% Compute the intersection of the two rule transducers
$R$ = $R1$ & $R2$

% Define a transducer for the inflectional endings
$INFL$ = <ADJ>:<> (<pos>:<> | <comp>:{er} | <sup>:{est})

% Concatenate the lexical forms and the inflectional endings and
% put a morpheme boundary in between which is not printed in the analysis
$S$ = $WORDS$ <>:# $INFL$

% Apply the two level rules
% The result transducer is stored in the output file
$S$ || $R$
```

3 Compilation

The SFST compiler translates the SFST source code to a minimized[1] transducer which is stored in the output file. The compiler is quite efficient and was successfully used to compile SMOR, a large German computational morphology.

The SFST tools support three different transducer formats which are optimized for flexibility, processing speed and start-up time/memory efficiency, respectively. The implementation of the SFST tools is based on a C++ class library which facilitates the development of new analysis tools.

4 Unicode

SFST supports UTF8 encoding or characters. In order to compile an SFST program with UTF8 encoding, you have to use the *fst-compiler-utf8* rather than the *fst-compiler* program. Both compilers store the type of encoding (8-Bit ASCII extension vs. UTF8) in the transducer file. Other programs such as fst-infl or fst-mor are therefore able to determine the appropriate character encoding.

[1] The minimization will not change the alignment between surface and analysis symbols. Therefore smaller transducers with different alignments may exist.

5 Usage of the SFST Programs

The command `fst-compiler ex.fst ex.a` compiles the program stored in the file *ex.fst* into a transducer which is written to the file `ex.a`.

The command `fst-mor ex.a` reads the transducer from the file and prompts the user to enter words. Entering an empty line will switch fst-mor into generation mode. Entering another empty line will turn on the analysis mode, again. Entering `q` terminates the session. Here is a sample session:

```
> fst-compiler ex.fst ex.a
> fst-mor ex.a
reading transducer...
finished.
analyze> easy
easy#<JJ>
analyze> easier
easy#er<JJR>
analyze> easiest
easy#est<JJS>
analyze>
generate> easy#est<JJS>
easiest
generate> q
```

`fst-infl` is a similar tool for batch mode analysis. It is used in one of the following ways:

```
> fst-infl ex.a file
> echo easiest | fst-infl ex.a
```

`fst-infl` has no generation mode. In order to use `fst-infl` for generation, you should compile the transducer using the option `-s` which tells the compiler to switch the surface and analysis symbols of the resulting transducer. `fst-infl` processes the input by converting each line into an automaton, composing the transducer with this automaton, extracting the domain of the resulting transducer and printing all strings generated by this automaton.

`fst-infl2` has the same functionality as `fst-infl`, but uses a more compact binary transducer format and a different analysis algorithm (traversal of the transducer with backtracking) which is more efficient if the degree of ambiguity is low. Use the compiler option `-c` or the separate program called *fst-compact* in order to generate the compact transducer format required by `fst-infl2`.

`fst-infl3` is the third member of the fst-infl family which supports the "lowmem" transducer format generated either by the compiler switch *-l* or by the separate conversion program *fst-lowmem*. fst-infl3 avoids reading the transducer into memory by directly accessing it on disc. This program start very quickly but processing is slower than with the other programs.

If you need information about the available options of one of the tools, just call it with the option `-h` or have a look at the man pages.

5.1 Other Tools

The command `fst-print` prints transducers in text form. `fst-compare` checks whether two transducers are equivalent. `fst-generate` enumerates the set of mappings of analysis to surface forms for the argument transducer.

`fst-match` is similar to `fst-infl2` but treats the input as a sequence of words that are to be analyzed, rather than a single word. `fst-match` repeatedly determines the longest prefix of the input which can be analyzed by the transducer, returns the respective analysis[2] and continues processing after the match.

`fst-parse` is able to compose several transducers at runtime. It converts the input into an automaton, composes the first argument transducer with it and then composes the second argument transducer with the result of the first composition and so on. Finally the analysis layer is extracted and the output is generated. `fst-parse` is typically used when the composition of two transducers cannot be computed offline. The transducer resulting from the composition of a two-level morphology and a finite-state grammar e.g. is usually too big to be computed with the compiler. The result of composing the input sentence with the morphological analyzer, on the other hand, is small enough to be composed with the transducer of the finite state grammar.

`fst-text2bin` converts transducers from text form to binary format. The two commands `fst-print transducer.a > transducer.txt` and `fst-text2bin transducer.txt > transducer2.a` should produce a transducer `transducer2.a` which is equivalent to (but not necessarily identical with) `transducer.a`.

Finally, there is a tool called `fst-infl2-daemon` which creates a daemon which communicates with application programs via sockets. It is used analogously to `fst-infl2` but reads and writes from/to a socket. The Perl script `socket-client.pl` in the directory `src` is an example application which communicates with `fst-infl2-daemon`:

```
> fst-infl2-daemon 7100 /corpora/mlex/smor.ca &
> echo "schlief" | ./socket-client.pl
> schlief
schlafen<+V><1><Sg><Past><Ind>
schlafen<+V><3><Sg><Past><Ind>
```

See the man pages for more information on all these commands.

5.2 Tricks

The compiler prints the name of the file and the line number that it is currently processing. It issues a warning when an empty transducer is assigned to a variable.

If some intermediate transducer is rarely changed and its compilation takes a long time, then it is a good idea to compile it separately and to include it in the main file by means of the `"<file>"` command.

During debugging, it can be useful to write intermediate results of the compilation to a file, e.g. by using the command `x >> "file"` which stores the transducer x in a file named *file*. `fst-print`, `fst-generate`, and `fst-mor` can be used to examine the transducer.

[2] If there is more than one mapping for the longest match, only one of them is printed.

5.3 Caveats

SFST operations sometimes produce other results than the user might have expected. This section discusses some typical examples.

Problems with Negation

Consider the following SFST program

```
ALPHABET = [a-z]
!(x)
```

One might expect that this transducer fails to analyze the string *abx*. But this is not the case. It only rejects the string x but accepts any other letter sequence.

Here is another example:

```
ALPHABET = a a:b a:<> b b:a b:<> <>:a <>:b
!{ab}:{ba}
```

Does this transducer generate the string *ba* from *ab*? Yes, it does! There are several ways in which the above transducer can map the string *ab* to *ba*, one of them consisting of the steps (i) mapping of *a* to the empty symbol (ii) mapping of b onto itself, and (iii) insertion of *a* after *b*. The crucial point is that the negation operation here disallows one particular mapping of the analysis string to the surface string, but still allows many others which are equivalent.

(The above transducer generates an infinite number of different surface forms for the string *ab*. Therefore you will get an error message if you try to actually generate from the string *ab*.)

Conjunction of Replace Rules

Now, consider this example:

```
ALPHABET = a b c
$Rule1$ = (a:b+) ^-> (b__b)
$Rule2$ = (a:c+) ^-> (c__c)
$Rule1$ & $Rule2$
```

The first rule generates *bbb* from *bab* and the second rules generates *ccc* from *cac*. It might be expected that the conjunction of the two rules performs both mappings. This is not the case, however. The transducer for the first rule maps *bab* to *bbb*, but *cac* to *cac*. The conjunction of the two rules therefore fails to generate from the string *cac* and similarly from the string *bab*.

In order to obtain the intended mapping for both strings, the two rules have to be combined by composition rather than conjunction.

Insertion With Replace Rules

Replace rules are used to exchange a string in a certain context for another string. The following rule maps an a in between two b's to c.

```
ALPHABET = a b c
a:c ^-> (b__b)
```

It might be expected that a similar replace rule could be used to insert a string in a certain context:

```
ALPHABET = a b c
<>:c ^-> (b__b)
```

This will not work! The replace operation is unable to replace the empty string. A slightly different rule will do the job, however:

```
ALPHABET = a b c
b:{bc} ^-> (__b)
```

5.4 Compilation Efficiency

Sometimes intermediate transducers generated by the compiler are much bigger than the final transducer and the compilation becomes slow. In such cases, it often helps to change the order in which the final transducer is built. Assume for instance that you have a sequence of 10 phonological rules which are stored in the variables `$rule1$` up to `$rule10$`. These rules are applied to a set of wordforms stored in variable `$lexicon$` by a sequence of composition operations. This can be done as follows:

```
$lexicon$ || $rule1$ || $rule2$ || ... || $rule10$
```

This command will compose `$lexicon$` and `$rule1$`. Then it composes the result with `$rule2$` and so on. At the end, the transducer is minimised. The following sequence of commands does basically the same but minimises after each composition operation:

```
$tmp$ = $lexicon$ || $rule1$
$tmp$ = $tmp$ || $rule2$
...
$tmp$ = $tmp$ || $rule9$
$tmp$ || $rule10$
```

It is also possible to first compile all rules into a single transducer which is then applied to the wordforms.

```
$rules$ = $rule1$ || $rule2$ || ... || $rule10$
$lexicon$ || $rules$
```

The rules can also be applied in chunks:

```
$rulesA$ = $rule1$ || $rule2$ || $rule3$
$rulesB$ = $rule4$ || $rule5$ || $rule6$
$rulesC$ = $rule7$ || $rule8$ || $rule9$ || $rule10$
$tmp$ = $lexicon$ || $rulesA$
$tmp$ = $tmp$ || $rulesB$
$tmp$ || $rulesC$
```

The result transducer is always the same. Which version is most efficient depends on the circumstances. If all the rules can easily be compiled into a single transducer (version 3 above), this is often a good choice because the rules transducer can be precompiled and stored in a file. If the rules transducer gets too big, it is better to use version 4 where the rules are applied in chunks. Again the chunks can be precompiled and stored in files.