# Tutorial Introduction to the SFST Finite-State Morphology Toolkit

Stefan Evert

9 December 2013

## Contents



## 1 Preliminary remarks

### 1.1 The SFST tools

The **SFST finite-state morphology toolkit** is a collection of command-line programs that can be used to compile **finite-state automata** (**FSA**) and **finite-state transducers** (**FST**) specified by grammar files, and to apply them to a list of input strings. For the FST exercises, you will need to use some of the following tools:

**fst-compiler-utf8** Creates a FSA or FST from a specification in the SFST grammar format.
*Input:* Text file with FSA/FST specification (customary extension: `.fst`)
*Output:* Compiled FSA or FST in internal binary format (customary extension: `.dat`)
*Example:*[1] `$ fst-compiler example.fst example.dat`

**fst-mor** Reads a compiled FSA or FST and applies it to input strings entered by the user in an interactive session. Enter an empty line to toggle between *analyze* and *generate* modes
*Input:* Compiled FSA or FST in binary format (`.dat`)
*Example* `$ fst-mor example.dat`

[1] The `$` sign marks a line that has to be typed into a Unix command shell. The `$` itself is a customary input prompt of such shells and is not part of the command line.

**fst-infl** Reads a compiled FSA/FST and a text file, then applies the FSA/FST to each input string listed in the text file. An FST is always applied in *analyze* mode, i.e. in inverse direction to its specification (see Section 3 for details.)
*Input*$_1$*:* Compiled FSA or FST in binary format (`.dat`)
*Input*$_2$*:* Text file (`.txt`) containing a list of input strings, one string on each line (without surrounding whitespace).
*Output:* A text file (`.txt`) containing the list of input strings (on lines starting with `>` character) followed by their analyses according to the FSA/FST. A full description of the output format is given in Sections 2 and 3, respectively. If the third argument is omitted, the output is printed on screen.
*Example:* `$ fst-infl example.dat wordlist.txt analyses.txt`

**fst-format-infl** An optional post-processor for the `fst-infl` output, which is included in the examples package (and pre-installed in the comptuer lab).
*Example:* `$ fst-infl example.dat wordlist.txt analyses.txt | fst-format-infl`

**fst-print** Prints a transition table for a compiled FSA/FST.
*Input:* Compiled FSA or FST in binary format (`.dat`)
*Output:* State transition table for the FSA/FST, printed on screen. The output can be *redirected* to a file as shown in the example below.
*Example:* `$ fst-print example.dat > transition-table.txt`

**fst-draw** An additional tool included in the examples package (and pre-installed in the computer lab). You can use `fst-draw` to display a compiled FSA or FST in diagram notation (see example below). Alternatively, you can save the diagram as a vector (`.eps`, `.pdf`) or bitmap (`.png`) image. Try `fst-draw -h` for more information.
*Input:* FSA or FST either in source (`.fst`) or compiled (`.dat`) form.
*Output:* A diagram representing the compiled FSA or FST, displayed on screen (`-v` option). Alternatively, a vector (`-e`, `-p`) or bitmap image (`-b`), saved to a file with the extension `.eps` (`-e`), `.pdf` (`-p`) or `.png` (`-b`).
*Example:* `$ fst-draw --view example.fst` (for on-screen viewing)

**fst-match** Scans a text file for all non-overlapping matches of a FST in *analyze* mode. Each match is replaced by the corresponding FST input string, i.e. its analysis.
*Input*$_1$*:* Compiled FST in compact encoding (compiler option `-c`, customary extension: `.ca`). *Input*$_2$*:* Text file (`.txt`) to be analyzed.
*Output:* Text with all FST matches replaced by their analyses. If the third argument is omitted, the modified text will be printed on the screen.
*Example:* `$ fst-match example.ca input.txt output.txt`

**fst-scanner** An additional tool included in the examples package (and pre-installed in the computer lab). You can use `fst-scanner` to mark all non-overlapping matches of a FSA in a text file. It wraps the FSA in a suitable transducer to insert markers around each match, then runs the `fst-match` program.
*Input*$_1$*:* Text file with FSA/FST specification (customary extension:`.fst`)
*Input*$_2$*:* Text file (`.txt`) to be analyzed.
*Output:* Text with all FSA/FST matches marked by `<[...]>`; any substitutions made by the original FST are also applied. If the third argument is omitted, the modified text will be printed on the screen.
*Example:* `$ fst-scanner regexp.fst input.txt output.txt`

Most of the SFST tools will print some progress and information messages while they are running. You can specify the command-line option `-q` in order to suppress these messages.

### 1.2 SFST grammar files

The SFST tools define a special grammar format for FSA/FST specifications. A detailed description of this format can be found in Sections 2 and 3. Grammar files customarily have the filename extension `.fst` (although most tools will also accept other extensions, except for `fst-draw`).

Each grammar file defines a single FSA or FST. Expressions always have to be written on a single line in these files. If you want to distribute a complex expression over multiple lines, you have to end each line but the last one with a backslash `\` (which *must not* be followed by a blank). All whitespace is ignored by default and must be protected with a backslash in order to be matched literally. For instance, a regular expression that matches the string *ab ab* would have to be written `ab\ ab`. Comments are introduced by a `%` character and extend to the end of the line.

SFST grammars use a wide range of meta-characters, including the familiar regular expression operators (`? * + |`), but also many others such as `& / : - ! ^ "` and all types of bracketing symbols (`( ) { } [ ] < >`). In order to process these characters with a FSA or FST (i.e. to match them *literally*), they have to be protected with a backslash. A regular expression for the string *<a:b>* would have to be written as `\< a \: b \>` (note the use of whitespace, which is ignored by the compiler, to make the expression more readable). When in doubt, you should protect any non-alphabetic character in this way.

The SFST tools have traditionally been used with 8-bit character sets such as ASCII, Latin-1, and other character sets from the ISO-8859 family. If your FST includes accented letters or other non-ASCII characters in Unicode (UTF-8) encoding, which is the default setting in the computer lab, please make sure to use `fst-compiler-utf8`, which makes special provisions for this encoding. If you need to process 8-bit legacy data, you can use `fst-compiler` instead. Some non-ASCII characters may not be rendered correctly by the `fst-draw` program, depending on the operating system and version.

## 2 Writing finite-state automata

Finite-state automata are specified in the form of regular expressions in a SFST grammar. Only the basic operators `? * + |` are supported, as well as the matchall `.` and character sets (`[ ... ]`). Any whitespace in the regular expression is ignored and should be used to improve readability. A grammar file should always start with an alphabet definition (see Figure 1). The alphabet is specified as a list of characters and character sets. It is mandatory when the matchall symbol `.` or the complementation operator `!` is used in the regular expression. As usual, any whitespace is ignored and meta-characters have to be protected by backslashes.

The example in Figure 1 shows a FSA that matches date strings such as *24-April-2005*. Note that the regular expression defining the FSA must be written on a single line. Parentheses and whitespace have been used as visual cues to the structure of the regular expression. They will be ignored by the compiler (parentheses are normally used for grouping and precedence, but do not have any function in this example), and the same holds for empty lines.

```
ALPHABET = [A-Za-z] [0-9] \-

([1-9]? [0-9]) \- ([A-Z][a-z]+) \- ([12][0-9][0-9][0-9])
```

Figure 1: Example of a FSA specification (file `regex.fst`)

When you have created a text file named `regex.fst` with the FSA specification shown in Figure 1, you can compile it with the following command:

```
$ fst-compiler-utf8 regex.fst regex.dat
```

If successful, this command will create a file `regex.dat` with an internal binary encoding of the compiled FSA. Otherwise, a short error message will be printed, indicating the line number of the grammar file on which the error occurred. You can now load the compiled FSA into the `fst-mor` tool in order to apply it to input strings entered interactively. Type:

```
$ fst-mor regex.dat
```

When the FSA has been loaded, the input prompt `analyze>` is displayed. You can enter an input string for the FSA now. If accepted by the FSA, the input string is simply echoed on the screen. Otherwise, the message `No result for` ... is printed. Enter `q` or type `Ctrl-C` to exit the program.

You can now use this regular expression to search a text file for date specifications, similar to the Unix tool `grep` (but with a the powerful design features of SFST for complex search patterns). A suitable sample text is included in the examples package as `input_events.txt`.

```
$ fst-scanner regex.fst input_events.txt
```

Note that you must call the `fst-scanner` utility with the source code file (`.fst`), not the pre-compiled FSA (`.dat`).

In a SFST grammar, the FSA does not have to be entered as a single complex regular expression. It can also be combined from smaller and simpler automata. In order to do so, any FSA specified by a regular expression can be assigned to a variable (an identifier enclosed in `$` characters, e.g. `$temp$`). Such automata can then be inserted into regular expressions or combined with the standard set operations (see below). Figure 2 shows an alternative specification for the FSA `regex.fst`, using sub-automata and meaningful variable names to make the grammar file self-documenting. The last line defines the FSA that will be compiled from the grammar file. It may not be assigned to a variable (which would be pointless anyway). Note that any sub-automata that are not used (either directly or indirectly) in the final expression will not be included in the compiled FSA.

```
ALPHABET = [A-Za-z] [0-9] \-

$Day$ = [1-9]? [0-9]
$Month$ = [A-Z] [a-z]+
$Year$ = [12] [0-9] [0-9] [0-9]

$Day$ \- $Month$ \- $Year$
```

Figure 2: Equivalent specification for the FSA in Figure 1

We will now look at a simpler example, shown in Figure 3. Save this grammar in a text file with the name `aba.fst`, then compile it with the following command:

```
$ fst-compiler-utf8 aba.fst aba.dat
```

```
ALPHABET = a b

a+ b+ a+
```

Figure 3: SFST grammar for the regular expression `a+b+a+` (file `aba.fst`)

Use `fst-mor` to check that this FSA accepts strings such as *aabba*, but not *bab* or *aaaa*. You can apply the FSA to a list of input strings with the `fst-infl` program. Create a text file `input.txt` with the strings *aabaa*, *aaab* and *abba* (or use the file `input_aba.txt` in the examples package). Each string must be on a separate line *without* surrounding whitespace. Typing the following command should print the output shown in Figure 4:

```
$ fst-infl aba.dat input.txt
```

You can also save the output to a file (e.g. `output.txt`) specified as an optional third argument.

```
> aabaa
aabaa
> aaab
no result for aaab
> abba
abba
```

Figure 4: Sample output of the `fst-infl` program

In order to render the output of `fst-infl` in a more readable and concise format, you can pipe it through the `fst-format-infl` utility included in the examples package. Use the `-q` option to suppress `fst-infl`'s startup messages:

```
$ fst-infl -q aba.dat input.txt | fst-format-infl
```

In order to print a transition table for the compiled FSA, type:

```
$ fst-print aba.dat
```

You can also display the FSA in diagram notation with the `fst-draw` program (see Fig. 5).

```
$ fst-draw --view aba.dat
```

If you pass a grammar file (e.g. `aba.fst`) to `fst-draw`, it will automatically be compiled and stored in a temporary file. Instead of viewing the diagram, you can also write it to a file in any of these formats:

- PDF (suitable for printing and inclusion in PDF documents ) with the `--pdf` option (creates file `aba.pdf`). This is the default operation when no option is specified.
- Encapsulated PostScript (suitable for inclusion in PostScript documents), with the `--eps` option (creates file `aba.eps`).
- PNG bitmap (suitable for on-screen viewing, e.g. in HTML pages or PowerPoint presentations) with the `--bitmap` option (creates file `aba.png`).

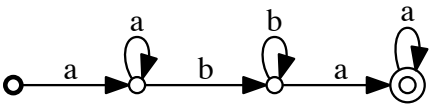


Figure 5: Graphical representation of a FSA created with `fst-draw`

Type `fst-draw --help` for a list of further options.

Partial automata $A$ and $B$ that have been assigned to variables (`$A$` and `$B$` in the examples below) can be combined with all standard **set operations**:

| | |
|---|---|
| `$A$ \| $B$` | union of $A$ and $B$ |
| `$A$ $B$` | concatenation of $A$ and $B$ |
| `($A$)*` | Kleene closure of $A$ |
| `$A$ & $B$` | intersection of $A$ and $B$ |
| `$A$ & (!$B$)` | difference of $A$ and $B$ ($A \setminus B$) |
| `! $A$` | complement of $A$ ($\Sigma^* \setminus A$) |

For the negation operator `!`, it is essential that the correct alphabet $\Sigma$ has been defined on the first line of the grammar file. You can also combine multiple set operations into a complex expression, using parentheses for grouping and precedence, and assign the resulting FSA to another variable. Figure 6 defines a FSA that accepts strings of the form $a^i b^j a^k$ (with $i, j, k \in \mathbb{N}_0$) which contain at least three $a$'s, but do not consist of $a$'s only.

```
ALPHABET = a b

$A$ = a* b* a*
$B$ = .* a .* a .* a .*
$C$ = a*

($A$ & $B$) & (! $C$)
```

Figure 6: Combination of FSA with set operations

The date string automaton in Figure 2 can be improved by replacing the generic expression for `$Month$` with a list of valid month names. While this list could be coded directly as a regular expression, a separate list of (literal) strings is both more readable and easier to maintain (e.g. when translating month names to a different language). Such a list can be read from a file by enclosing its name in double quotes:

```
$Month$ = "monthnames.txt"
```

The file `monthnames.txt` must contain a list of literal strings (using the alphabet declared for the FSA) on separate lines, without surrounding whitespace. This list is automatically compiled into a FSA matching any of the strings, which is then stored in the variable `$Month$`.

In order to match full monthnames (*April*) as well as three-letter abbreviations (*Apr*), it is tempting to use regular expressions such as `Apr(il)?` rather than listing both forms explicitly

in the file `monthnames.txt`. However, word lists are always interpreted as literal strings by the SFST tools, so that the regular expression above would only match the precise string *Apr(il)?*. A solution to this problem is the inclusion of a compiled FSA from a separate file. In order to do so, create a FSA definition as shown in Figure 7 (note the continuation backslash `\` at the end of the first line), compile it with the command

```
$ fst-compiler-utf8 months.fst months.dat
```

and replace the definition of the variable `$Month$` in `regex.fst` by

```
$Month$ = "<months.dat>"
```

Note that only compiled FSA can be included in this way. If you modify the grammar file `months.fst`, you have to make sure yourself that `fst-compiler-utf8` is run in order to update the pre-compiled file. Otherwise, the old version of the FSA will be included from the file `months.dat`.

```
Jan(uary)? | Feb(ruary)? | Mar(ch)? | Apr(il)? \
| May | June? | July? | Aug(ust)? \
| Sep(tember)? | Oct(ober)? | Nov(ember)? | Dec(ember)?
```

Figure 7: FSA definition for month names with abbreviations (file `months.fst`)

It is highly recommended to split complex FSA definitions into word lists and sub-automata that are stored and maintained in separate files.

## 3 Writing finite-state transducers

Finite-state transducers are an extension of FSA that can also transform accepted strings, i.e. they replace the input string $w$ with an output string $f(w)$. Formally, FST define a (partial) regular mapping between strings in the input alphabet $\Sigma$ and strings in the output alphabet $\Gamma$, $f : \Sigma^* \rightarrow \Gamma^*$. The mapping $f$ is partial because $f(w)$ is only defined for input strings that are accepted by the FST. For a non-deterministic FST, an input string $w$ may be mapped to several different output strings, so that $f(w)$ is a multi-valued function, i.e. a relation $f \subseteq \Sigma^* \times \Gamma^*$ between input and output strings. (In most cases, we will use the same alphabet for input and output strings, i.e. $\Sigma = \Gamma$).

In order to generate $f(w)$, FST can output a symbol $a' \in \Gamma$ whenever they process an input symbol $a \in \Sigma$ (by traversing a state transition). In an SFST grammar, such transition labels are written in the form $a$`:`$a'$, e.g. `x:u` to convert an `x` in the input string to a `u` in the output string. Both the **input label** $a$ and the **output label** $a'$ must consist of a single symbol. In order to map $w$ to a string of different length, it must be possible to (i) insert output symbols without processing any input and (ii) process input symbols without generating any output. This is achieved by specifying the empty string $\epsilon$ as an input label (e.g. $\epsilon$`:`$a'$ to insert symbol $a'$ in the output, case (i)) or output label (e.g. $a$`:`$\epsilon$ to delete symbol $a$ in the input, case (ii)). The empty string is written as `<>` in the SFST formalism (e.g. `x:<>` and `<>:u`).

Figure 8 shows a FST that translates the English sheep language (*baa!, baaa!, baaaa!, ...*) to a German sheep language (*määh!, määääh!, määäääh!, ...*). This FST defines a function $f$ with $f(baa!) = määh!$, $f(baaa!) = määääh!$, etc. Figure 9 gives a specification of the transducer in the SFST formalism. Save this grammar to the file `sheep.fst` and compile it as usual with the command

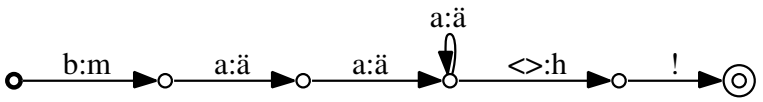


Figure 8: FST that translates between English and German sheep

```
ALPHABET = b m a ä h \!

b:m a:ä (a:ä)+ <>:h \!:\!
```

Figure 9: SFST specification for the FST in Figure 8 (file `sheep.fst`)

```
$ fst-compiler-utf8 sheep.fst sheep.dat
```

Next, load the FST into `fst-mor`. You will notice that the program starts up in **analyze** mode, i.e. it will accept *output* strings and map them to *input* strings (by applying the inverse mapping $f^{-1}$). You can now type German sheep sounds and have them translated into English. In order to switch to **generate** mode (which applies the original mapping $f$), press the return key once (without any input). The input prompt changes to show that you can now translate English sheep sounds to German. Enter `q` to exit the program.

You can also translate an entire list of strings with the `fst-infl` program, as described in Section 2 for FSA. The output of `fst-infl` is the same as shown in Figure 4, except that for any accepted string all possible transformations are printed (rather than just echoing the string). It is important to realize that `fst-infl` *always* operates in analyze mode, i.e. it can only translate German to English according to the mapping $f^{-1}$.[2] In order to translate according to the mapping $f$ with `fst-infl`, the FST has to be compiled with the option `-s`, which swaps input and output labels on all transitions:

```
$ fst-compiler-utf8 -s sheep.fst sheep.dat
```

As you can see from Figure 9, the SFST tools use the same grammar formalism for FST as described in Section 2 for FSA. The only difference is that **atoms**, i.e. single characters in the regular expressions which are mapped to transition labels in the FSA, are now replaced by pairs of input and output characters (which are mapped to the labels of the FST). In many applications (and especially in computational morphology), only some symbols are translated while most others are just passed through, corresponding to **default pairs** of the form $a$`:`$a$ (e.g. `\!:\!` in the example above). Such identity labels can be abbreviated to the symbol $a$ itself (cf. the rightmost transition in Figure 8). Thus, a regular expression or FSA (as defined in Section 2) corresponds to a FST that performs the identity mapping $f(w) = w$ for all input strings it accepts.

When mixing pair labels and simple labels in a FST specifiction, it is important to remember that the input and output labels consist of a single symbol each. Thus, the expression `bb:cc` will *not* translate *bb* to *cc*, but rather the string *bbc* to the string *bcc*. If you want to translate a literal string $u$ into a string $v$, you can enclose them in curly braces: `{`$u$`}:{`$v$`}`. Note that both $u$ and $v$ must be a single literal string (with meta-characters properly escaped). It is also invalid to put braces around one of the labels only, even if the other one is a single symbol (or the empty

[2] This behaviour makes sense for most implementations of finite-state morphology, which define a mapping $f$ from a root with morphological features (such as `mouse<N><Pl>`) to a surface string (such as `mice`). In order to use such a FST for morphological analysis, the inverse mapping $f^{-1}$ has to be applied.

string). Thus, to insert the string *+NN* in the output you would have to write the expression `{<>}:{\+NN}`. As a special case, input and output labels may be character sets. This notation maps characters from the input set to the corresponding characters from the output set, e.g. `[a-zA-Z]:[A-ZA-Z]` to map lowercase letters to uppercase and pass through uppercase letters unaltered.

All set operations defined in Section 2 can be used to combine transducers stored in variables. Complementation (`!`) should be used with great caution and is only meaningful when a special alphabet including all feasible label pairs has been defined (see the SFST documentation for details). An important additional operation available for FST is the **composition**. Given a transducer `$A$` for the mapping $f$ and a transducer `$B$` for the mapping $g$, the transducer `$A$ || $B$` represents the mapping $g \circ f$ with $(g \circ f)(w) = g(f(w))$.[3] It is also possible to combine FST and FSA with these operations, since the latter represent an identity mapping on the accepted strings.

FST composition can be used to break down a complex mapping into smaller steps that can easily be described by FST. The full mapping is then obtained by composition of the individual FST. The resulting automaton will be relatively compact and can be simulated more efficiently than a cascade of separate FST. As a special case, composition with a FSA can be used to filter the input or output of a FST mapping. For example, the FST shown in Figure 10 implements fast text entry using the keypad of a mobile phone. Each keypad button stands for one of several letters, and the phone uses context information to decide which letter to insert. In order to enter the string `DOG`, for example, one would tap the keys `3`, `6` and `4`.

```
ALPHABET = [A-Z] [0-9]

$T9$ = (2:[A-C] | 3:[D-F] | 4:[G-I] | 5:[J-L] \
        | 6:[M-P] | 7:[Q-T] | 8:[U-W] | 9:[X-Z])*

$Lexicon$ = "wordlist_en.txt"

$T9$ || $Lexicon$
```


Figure 10: FST for fast keypad text entry, where e.g. button `2` stands for the letter `A`, `B` or `C`. The mapping FST is composed with a FSA encoding a large word list in order to filter out replacements that do not correspond to English words (file `t9.fst`).

The sub-FST `$T9$` implements the mapping from numeric keys to letters. It accepts a sequence of digits as input, mapping each digit to one of the corresponding letters. By itself, `$T9$` describes a multi-valued mapping: `364` would be transformed into a set of $3 \times 4 \times 3 = 36$ different output strings `DMG`, `DMH`, `DMI`, `DNG`, etc. In a second stage, the output is filtered against a large list of English word forms included in the examples package, which is encoded into a sub-FSA `$Lexicon$`. The composition of the FST and the FSA generates only valid English words from the numeric input, e.g. `DOG` and `FOG` for the input `364`.

The mapping defined by a FST can easily be inverted by swapping the input and output symbol of each transition. In SFST syntax, this operation is performed with the prefix operator `^_`. If an English-to-German translator for animal sounds has been stored in the variable `$Animals$`, the FST `^_ $Animals$` will translate German to English (see file `animals_inverse.fst`). The result is identical to running `fst-compiler-utf8` with the `-s` option. Most SFST tools operate by default in *analyze* mode. For example, the keypad entry FST shown in Figure 10 will map

[3] Note that the transducers are specified left-to-right in SFST syntax, which is opposite to the mathematical notation for function composition (right-to-left).

words to numeric codes when simulated with `fst-mor` or `fst-infl`. In order to obtain the desired mapping direction, the last line of the file has to be inverted: `^_ ($T9$ || $Lexicon$)`.

Every FST defines a mapping between two regular languages: the set of accepted input strings (i.e. the **domain** of the mapping $f$) and the set of output strings that can be generated (i.e. the **image** or **range** of $f$). FSA describing these languages can be obtained from a given FST with the prefix operators `_` (domain) and `^` (image). This functionality is particularly useful to implement fallback / default behaviour. For instance, if the FST `$General$` defines a general mapping and the FST `$Exceptions$` a list of special cases, the following specification defaults to the general FST whenever the input string is not found among the special cases:

```
$Exceptions$ | ( !(_ $Exceptions$) || $General )
```

For a morphological analyser, it often makes more sense to specify a fallback rule `$Fallback$` that is applied if the output string (i.e. the surface string to be analyzed) cannot be generated by the full-fledged morphology FST `$Morph$`:

```
$Morph$ | ($Fallback$ || !(^ $Morph$))
```

# 4 A simple morphology example

## 4.1 Concatenative morphology

This section describes a (very) simple example of a morphological transducer. The goal here is to analyze singular and plural forms of the English nouns *dog, cat, mouse, house, car* and *bus*. Our transducer will translate root+feature representations to surface forms, e.g. `dog<N><Sg>` → *dog* and `cat<N><Pl>` → *cats*. A first version of this FST is shown in Figure 11, and consists of the following three lines:

- an alphabet declaration, including the letters `a-z` plus special symbols for the morphosyntactic features `<N>`, `<Sg>` and `<Pl>` (the angle brackets denote special multi-character symbols in SFST, which are treated as atomic elements of the alphabet)
- a list of 6 roots, written as a disjunction and stored in the variable `$Roots$`;
- the FST expression, which accepts one of the specifed roots and passes it through unchanged, deletes the word class marker `<N>`, and then translates the number feature `<Sg>` to the empty string $\epsilon$ or the feature `<Pl>` to the plural suffix *-s*.

Note that the FST has been designed to generate surface forms from root+feature specifications. When loaded into `fst-mor` in *analyze* mode or used with `fst-infl`, it acts as a morphological analyzer that rejects invalid forms and prints root+feature representations for all valid forms.

```
ALPHABET = [a-z] <N> <Sg> <Pl>

$Roots$ = dog | cat | mouse | house | car | bus

$Roots$ <N>:<> ( <Sg>:<> | <Pl>:s )
```

Figure 11: First version of the morphological transducer for English nouns (file `plural1.fst`)

This first version of the FST is far from perfect. It does not analyze the irregular plural *mice*, but accepts incorrect forms such as *mouses* and *buss*. The improved FST in Figure 12 assigns the original transducer to the variable `$Regular$` and defines a list of exceptions to the regular mapping (viz., `mouse<N><Pl>` → *mice* and `bus<N><Pl>` → *buses*). Note that we only need to list forms that do not follow the regular pattern, not the entire paradigms of "irregular" roots. If we simply computed the disjunction `$Regular$ | $Exceptions$`, the resulting FST would recognize *mice* and *buses*, but it would still accept the incorrect forms *mouses* and *buss*. In order to delete these from the regular mapping, we need to restrict the *domain* of the regular mapping, i.e. the set of root+feature descriptions to which it applies. This is achieved by composition of `$Regular$` with a FSA that describes all root+feature descriptions that do *not* belong to the domain (`_ $Exceptions$`) of the exception mapping, viz. its complement (`! _ $Exceptions$`). Note the order of composition: if we wanted to restrict the *range* of the regular mapping, we would have to use an expression of the form `($Regular$ || ...)`.

```
ALPHABET = [a-z] <N> <Sg> <Pl>

$Roots$ = dog | cat | mouse | house | car | bus

$Regular$ = $Roots$ <N>:<> ( <Sg>:<> | <Pl>:s )

$Exceptions$ = {mouse<N><Pl>}:{mice} | {bus<N><Pl>}:{buses}

$Exceptions$ | ( (! _ $Exceptions$) || $Regular$ )
```

Figure 12: A version of the morphological transducer with exception rules for irregular forms (file `plural2.fst`)

## 4.2 Rewrite rules

The simple approach above misses an important generalization: the plural morpheme is always realized as *-es* if the root ends in a sibilant (*s, z, x, sh* or *ch*), for obvious phonological reasons. This phenomenon is known as *e*-epenthesis. The plural *buses* is not an irregular form that needs to be included in a list of exceptions after all—it can be predicted from the form of the root.

In order to exploit such generalizations, one possibility is to subdivide the list of noun roots into those ending in a sibiliant (plural allomorph *-es*), the other regular noun roots (plural allomorph *-s*), and a list of irregular nouns with their respective plural forms. Each list can then be combined with the respective inflectional paradigm. A FST following this approach is shown in Figure 13.

There is a second regular pattern of variation in the plural forms of English nouns: a spelling convention requires that nouns ending in *y* form the plural with *-ies*, unless the *y* is preceded by a vowel (*party* → *parties*, but *boy* → *boys*). This generalization cannot be represented with a purely concatenative approach to morphology as in Figure 13 because the root has to be modified. Moreover, sorting noun roots into different lists depending on their final letters is tedious and redundant: the relevant information is already contained in the form of the root.

The two-level approach to morphology uses **rewrite rules** to capture such regular phonological patterns. In a first step, a purely concatenative transducer maps grammatical words (root+features) to sequences of morphemes. In a second step, one or more context-dependent rewrite rules modify the root and affixes. For example, the plural *buses* would be derived as follows:

```
ALPHABET = [a-z] <N> <Sg> <Pl>

$RootsRegular$ = dog | cat | house | car
$RootsSibilant$ = bus | crash | fox

$Regular$ = $RootsRegular$ <N>:<> ( <Sg>:<> | <Pl>:s )
$Sibilant$ = $RootsSibilant$ <N>:<> ( <Sg>:<> | <Pl>:{es} )

$Exceptions$ = {mouse<N><Sg>}:{mouse} | {mouse<N><Pl>}:{mice}

$Regular$ | $Sibilant$ | $Exceptions$
```

Figure 13: A better approach to English noun morphology represents *e*-epenthesis as regular *-s*/*-es* allomorphy (file `english_nouns.fst`).

```
bus<N><Pl>
    ⇓
 bus^s#
    ⇓
 buses
```

In the intermediate representation, individual morphemes are customarily separated with `^` and `#` marks the end of the word. The rewrite rules of the second step replace *-s* by *-es* (because the preceding root ends in a sibiliant), then remove the morpheme and word boundary markers. The SFST notation for the rewrite rule implementing *e*-epenthesis is

```
{s}:{es} ^-> ([szx]|sh|ch) \^ __ \#
```

The replacement to the left of the rewrite operator `^->` is carried out in the context specified on the right-hand side. The position of the replacement itself is indicated by a double underscore (`__`), with regular expressions specifying the required left and right context. In the case of *e*-epethensis, the *-s* must be the last morpheme of the word (i.e. the right context is a word boundary) and be preceded by a morpheme boundary and a sibilant (left context). Note that the literal symbols `\^` and `\#` have to be escaped in SFST notation. Both left and right context are mandatory and have to be specified as `.* __ .*` for an unconditional rewrite rule such as the final deletion of morpheme and word boundaries.

Figure 14 shows a two-level implementation of English noun morphology. Rewrite rules for *e*-epenthesis, roots ending in *y* and deletion of morpheme and word boundaries are stored in the sub-FST `$Epenthesis$`, `$Y$` and `$Clean$`, respectively. These rules are then applied in sequence by FST composition.[4] Irregular nouns are represented by a separate sub-FST `$Irregular$`.

When defining rewrite rules, make sure that the alphabet only consists of symbols (i.e. default pairs) and does not include any other feasible pairs—the rules may have unexpected side effects otherwise. Note that the alphabet can be redefined at any point in a SFST source file, e.g. before a section of rewrite rules.

A small exercise for readers: (i) Modify the FST in Figure 14 so that the list of noun roots is read from a separate text file (containing one root per line). (ii) Can you combine the two-level approach with ideas from Sec. 4.1 so that only irregular *forms* have to be specified as exceptions, rather than full inflectional paradigms for irregular noun roots?

[4] Keep in mind that ordering matters. If the deletion rule were applied first, none of the other rules would match any more.

```
ALPHABET = [a-z] <N> <Sg> <Pl> \^ \#

$Roots$ = bus | bush | car | cat | crash | day | dog | fox | house | party

$Lexical$ = $Roots$ <N>:<> ( <Sg>:<> | <Pl>:{\^s} ) <>:\#

$Irregular$ = \
  {mouse<N><Sg>}:{mouse} | {mouse<N><Pl>}:{mice} \
| {woman<N><Sg>}:{woman} | {woman<N><Pl>}:{women} \
| {fish<N><Sg>}:{fish} | {fish<N><Pl>}:{fish}

$Epenthesis$ = {s}:{es} ^-> ([szx]|sh|ch) \^ __ \#

$Y$ = {y}:{ie} ^-> [^aeiou] __ \^ s \#

$Clean$ = (\^:<> | \#:<>) ^-> .* __ .*

$Regular$ = $Lexical$ || $Epenthesis$ || $Y$ || $Clean$

$Regular$ | $Irregular$
```

Figure 14: Two-level FST for English noun morphology (file `english_nouns_rewrite.fst`).