Developing Computational Morphologies
Using the SFST Tools

Helmut Schmid

schmid@cis.lmu.de
Center for Information and Language Processing
Ludwig-Maximilians-Universität
Munich, Germany

1 Introduction

This tutorial is intended as a hands-on introduction to the implementation of computational morphologies using the Stuttgart Finite State Transducer (SFST) tools. It assumes some basic knowledge of finite automata and formal languages on the reader's side. Although it is primarily concerned with morphology, it might nevertheless be also relevant for users working in other areas who want to learn how to write SFST programs.

The tutorial is organized as follows: Section 2 gives a general answer to the question *What is a finite state transducer?* Section 3 introduces the most basic concepts and operators of the SFST programming language with simple examples. Section 4 is the largest section. It contains a step-by-step introduction to the implementation of a computational morphology using English as the example language.

There is no intention to achieve full coverage of the English morphology, here. Only selected phenomena are picked out as instructive examples and implemented with finite state transducer programs, leaving other similar phenomena as an exercise. Nevertheless, I hope that the tutorial is comprehensive enough for the reader to learn all he/she needs to implement a computational morphology for English or a different language.

This tutorial is preferably read in front of a computer so that the reader can immediately test the examples. Of course, the SFST tools have to be installed first. They are available at `http://www.ims.uni-stuttgart.de/projekte/gramotron/SOFTWARE/SFST.html`.

The tutorial contains exercises as well as excursions where aspects of the SFST syntax and the usage of the SFST tools are addressed in more detail. A comprehensive description of the SFST syntax is contained in the SFST manual which is part of the SFST software package.

2 Finite State Transducers

A finite state transducer (FST) is basically a finite state automaton where each transition is labeled with a symbol pair rather than a single symbol. Figure 1 shows an example. Like finite automata, FSTs have a single start state and a set of final states. They can be applied in three different ways:

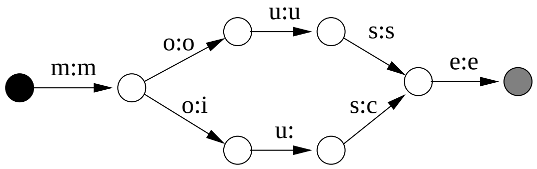

Figure 1: Example transducer in graphical notation with a black start state and a grey end state

Analogously to a finite automaton, a FST *accepts* a sequence s of symbol pairs (instead of symbols) if there is a path from the start state to some final state where the sequence of symbol pairs on the transitions is identical to s. The transducer shown in figure 1, for instance, accepts the two sequences `m:m o:o u:u s:s e:e` and `m:m o:i u:  s:c e:e`. The second symbol of the symbol pair `u:` is the *empty* symbol.

FSTs are mainly used to map strings to other strings. A typical example is morphological analysis where an inflected word form is analyzed and the base form is returned with additional morphosyntactic information.

The transducer in figure 1 *analyzes* the strings `mouse` and `mice` and returns the string `mouse` in both cases. In general, a FST maps a string s to a string t (in *analysis* mode) if there is a path from the start state to some end state where the right-hand side symbols on the transitions are identical to s and the left-hand side symbols are identical to t.

The mapping is reversible: In *generation* mode, a transducer maps a string s to a string t if the left-hand side symbols on the transitions of some path from the start state to an end state are identical to s and the corresponding right-hand side symbols are identical to t. The transducer shown in figure 1 generates the strings `mouse` and `mice` for the input string `mouse`.

It is often helpful to think of a transducer as a (possibly infinite) set of aligned string pairs such as `m:m o:i u:  s:c e:e`, which is formed by the set of symbol pair sequences that the transducer accepts. The disjunction (operator |) of two transducers is then equivalent to the union of the two sets of aligned string pairs; the conjunction (operator &) is equivalent to the conjunction of the two sets, and the subtraction (operator –) of two transducers is equivalent to the set of string pairs appearing in the first set, but not in the second.

The concatenation of two transducers includes any string pair $s = s_1s_2$ where s_1 is in the first set and s_2 is in the second set of string pairs.

3 First Steps

The SFST tools provide a programming language for the implementation of finite state transducers which is based on extended regular expressions. The basic concepts and operators of this language will be introduced in this section. More advanced operators will follow in section 4.

3.1 Simple Regular Expressions

SFST programs are essentially (extended) regular expressions. The expression

```
Hello
```

for instance, defines a transducer which accepts the string *Hello* and returns the same string as the analysis.

Exercise Store this expression in a file called *foo.fst* and compile it with the command:

```
> fst-compiler foo.fst foo.a
```

Start the analyzer program:

```
> fst-mor foo.a
analyze> Hello
Hello
analyze> Hi
no result for Hi
```

In the same way, you can test the transducers presented in the following paragraphs.

The following transducer analyzes several strings, namely the strings “John”, “Mary”, and “James”:

```
John | Mary | James
```

This transducer expression uses the “or” operator (|) to build the disjunction of the three basic expressions. Blank and tab characters are ignored in SFST programs unless they are quoted by a preceding backslash. The next transducer shows the quotation of the blank symbol and the symbol “!” which is otherwise interpreted as an operator. The transducer accepts the string “Hello world!”.

```
Hello\ world\!
```

SFST supports many of the regular expression operators known from Unix tools such as *egrep* or *Perl*. Among them are the optionality operator "?" (for 0 or 1 repetition), the Kleene operators "*" (for 0 or more repetitions) and "+" (for 1 or more repetitions), and the square bracket notation for character ranges (including the negation operator ^). The following expression uses these operators. It defines a transducer which accepts numbers including fractional numbers with an optional sign.

```
[+\-]? [0-9]* (\. [0-9]+)?
```

3.2 The Colon Operator

So far, the transducers simply returned the input string as the "analysis". We will now see how transducers are implemented which map strings to other strings. The colon operator (:) is used to indicate the target symbol to which a single symbol is to be mapped. The transducer `a:b`, for instance, returns an "a" in analysis mode if a single "b" is entered. In generation mode, it is the other way around and the "a" is mapped to a "b".

Attention: In the Xerox finite state tools, the direction of the colon operator is opposite: The Xerox transducer `a:b` would analyze "a" as "b".

The expression below defines a transducer which maps strings consisting of lower-case and upper-case characters to the corresponding string of upper-case characters:

```
[A-ZA-Z]:[A-Za-z]*
```

The expression `[A-ZA-Z]:[A-Za-z]` is equivalent to the disjunction
`A:A | B:B | ... | Z:Z | A:a | B:b | ... | Z:z`.

With a similar expression, we can implement a simple word encryptor which replaces each character of the input string by the next character in the alphabet:

```
[b-zaB-ZA]:[a-zA-Z]*
```

Now assume we want to map each character to the next but one character in the alphabet. We could do that by applying the above transducer twice, mapping the string *cat* first to *dbu* and then to *ecv*. This is exactly what the following transducer does:

```
[b-za]:[a-z]* || [b-za]:[a-z]*
```

The *composition* operator “||” applies its left and right argument transducers in sequence to the input. The transducer eventually generated by the compiler performs the mapping in one step. Thus “a” is directly mapped to “c”. You can see that when you print the compiled transducer with the command `fst-print foo.a`

```
final: 0
0       c:a     0
0       h:f     0
0       q:o     0
0       f:d     0
...
```

The output of the *fst-print* command shows the list of state transitions with the source state, the symbol pair and the target state of each transition. The order in which the transitions are printed is random and could be different from the one shown above.

4 Developing a Morphology

A computational morphology analyzes an inflected word form such as “houses” and usually returns (i) the base form (lemma) “house”, (ii) the part of speech “Noun” and (iii) additional features such as the number “plural”.

The lemmatization of the word *cats* to *cat* is accomplished by the transducer:

```
cat<>:s
```

The operator ':' is here used to map the final symbol “s” to the *empty symbol* which is represented by “`<>`”. The above expression is an abbreviation of the expression[1]:

```
c:c a:a t:t <>:s
```

The symbol on the left-hand side of a colon belongs to the analysis string, the symbol on the right-hand side to the surface string.

In order to add information about the part of speech and the number feature to the analysis string, the transducer should be modified as follows:

```
cat <Noun>:<> <plural>:s
```

[1] Similarly, our first transducer expression `Hello` was an abbreviation of `H:He:el:ll:lo:o`.

<Noun> and <plural> are *multi-character symbols* which are treated like a single character. The above transducer analyzes the string *cats* by (i) mapping the characters “c”, “a”, and “t” to themselves, (ii) inserting the multi-character symbol <Noun>, and (iii) mapping “s” to the symbol <plural>.

Exercise Store the above expression in the file *morph.fst*, compile it and start *fst-mor*. Analyze the string *cats*. Enter an empty line in order to switch to generation mode and generate the inflected word form for the string “cat<Noun><plural>”.

```
analyze> cats
cat<Noun><plural>
analyze>
generate> cat<Noun><plural>
cats
```

The next transducer produces exactly the same mapping as the previous transducer. Nevertheless the two transducers are different because the alignment of surface and analysis symbols is different.

```
cat <Noun>:s <plural>:<>
```

We are know ready to implement a simple computational morphology for English as shown below.

```
a<Det>:<><sg>:<> |\
a\.m\.<Adv>:<> |\
...
zymurgy:i<>:e<>:s<Noun>:<><pl>:<> |\
zymurgy<Noun>:<><sg>:<>
```

The periods have to be quoted because they have a special meaning in SFST. The backslash at the end of the line tells the compiler that the expression continues in the following line.

The compilation of this transducer is rather slow. A more efficient implementation of the same transducer is obtained by storing most of the relevant information in a separate *lexicon* file which is processed much faster than regular SFST code because lexicon files use a restricted syntax: the only operators available are the colon and the backquote. Any other operator symbol and also the blank and tab characters are interpreted literally. Multi-character symbols such as <sg> are recognized and treated as a single symbol.

The lexicon file for our simple English full-form morphology has the following content:

```
a<Det>:<><sg>:<>
a.m.<Adv>:<>
...
zymurgy:i<>:e<>:s<Noun>:<><pl>:<>
zymurgy<Noun>:<><sg>:<>
```

When the compiler reads the lexicon file, it converts each line into a transducer and combines the transducers with an "or" operation. This is the reason why the |\ at the end of the line is missing.

The main program of the transducer looks as follows:

```
"morph.lex"
```

This command reads the lexicon file and returns the resulting transducer.[2]

Exercise Store the above line in the file *morph.fst* and create a file *morph.lex* with lexicon entries. Compile *morph.fst* and try to analyze different word forms using *fst-mor*.

Caveat: The file names should only contain ASCII characters!

Adding all the `:<>` character sequences in the lexicon file is cumbersome and makes the lexicon file less readable. We will later see how they can be inserted automatically.

4.1 The Period as a Wildcard Symbol

The period is a wildcard symbol which stands for any available symbol pair. The set of *available* symbol pairs is listed in the *alphabet*. The alphabet is defined by a command of the form `ALPHABET = ...expression...` and changes whenever this command is used. The compiler computes the actual value of the alphabet by first compiling *...expression...* to a transducer and then extracting all symbol pairs occurring on transitions of the resulting transducer. All of the following commands define the alphabet as the set {a:a,b:b,c:b}:

```
ALPHABET = a b c:b
ALPHABET = a | b | c:b
ALPHABET = [a-c]:[ab]
```

[2] This example will only work with SFST version 1.3 or higher. Previous versions treated the multi-character symbols in the lexicon file literally as a sequence of characters because they were not seen before in the main program.

The expression `[a-c]:[ab]` is first expanded to `[abc]:[ab]` and then to `a:a|b:b|c:b`.

The next transducer program replaces lower-case characters with upper-case characters.

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

The alphabet is here defined as the set {A:A, B:B, ..., Z:Z, A:a, B:b, ..., Z:z}. The wildcard symbol "." is replaced by the disjunction of all these character pairs. The resulting transducer is equivalent to the transducer `[A-ZA-Z]:[A-Za-z]*` seen earlier.

We are now ready to define a new version of our morphology which deletes the part-of-speech and feature symbols in the surface string of the transducer.

```
ALPHABET = [a-zA-Z\.] [<Det><Noun><Verb><Adj><Adv><Prep><sg><pl>]:<>
"morph.lex" || .*
```

The corresponding lexicon file looks as follows:

```
a<Det><sg>
a.m.<Adv>
...
zymurgy:i<>:e<>:s<Noun><pl>
zymurgy<Noun><sg>
```

4.2 Inflection Classes

In order to avoid an explicit enumeration of all the possible inflected word forms for a given lemma (such as *walk*, *walks*, *walked*, *walking* for the verb *to walk*), we can put all lemmas belonging to the same inflection class into a separate lexicon file and add the different inflectional endings in the main program. The lexicon file *verb-reg.lex* for regular verbs might contain the following entries:

```
walk
gain
mention
```

These verbs are inflected by adding either *s*, *ed*, *ing*, or nothing. The following transducer program reads the lexicon and adds the endings.

```
"verb-reg.lex" <Verb>:<> (\
  {<3s>}:{s} |\
  {<past>}:{ed} |\
  {<part>}:{ed} |\
  {<gerund>}:{ing} |\
  {<n3s>}:{} |\
  {<base>}:{})
```

The expression `{<past>}:{ed}` is equivalent to `<past>:e<>:d`. The symbols from the two symbol sequences which are enclosed in curly brackets are pairwise combined, adding empty symbols at the end as needed.

The parentheses are needed to indicate that `<Verb>:<>` is concatenated with the disjunction of all endings and not just with `{<3s>}:{s}`.

Similarly, we create a lexicon file *noun-reg.lex* for nouns with regular nominal inflection:

```
cat
house
door
```

and extend the transducer program. The transducer is now so complex, that it is better to build it in several steps, using the variables `$verb-reg-infl$` and `$noun-reg-infl$` to store intermediate transducers:

```
$verb-reg-infl$ = <Verb>:<> (\
  {<3s>}:{s} |\
  {<past>}:{ed} |\
  {<part>}:{ed} |\
  {<gerund>}:{ing} |\
  {<n3s>}:{} |\
  {<base>}:{})

$noun-reg-infl$ = <Noun>:<> (\
  {<sg>}:{} |\
  {<pl>}:{s})

"verb-reg.lex" $verb-reg-infl$ |\
"noun-reg.lex" $noun-reg-infl$
```

Exercise Extend the above program with adjectival inflection. The transducer should be able to analyze the word forms *quick/quicker/quickest*.

Excursion When a transducer program fails to produce the correct results, the following debugging strategies might be helpful:

Store intermediate transducers `$T$` in a file using commands such as `$T$ >> "file.a"` within the SFST program. Call `fst-generate file.a` in the shell to generate mappings from analysis strings to surface strings allowed by the intermediate transducer. Check whether the results are as expected.

If the compilation of the transducer takes a long time because the lexicon is very large, replace the lexicon with a smaller one containing only entries which are relevant for debugging.

4.3 Negation

SFST has two different complement (or negation) operators. The expression `[^b]` represents the set of all symbols without "b". *All symbols* means *all symbols used in the program up to this point.* Therefore, the transducer `a:[^b]` maps (in generation mode) the character "a" to any symbol encountered before except "b".

"!" is the negation operator for transducers. `!$T$` is equivalent to `.* - $T$` which is the set of all mappings allowed by the alphabet (`.*`) minus the set of mappings performed by the transducer which is stored in `$T$`. Note that `!a:b` is very different from `a:[^b]`.

4.4 Orthographic and Phonological Rules

We can treat verbs such as *to paste* as regular verbs if we add a rule *delete-e* which deletes the final "e" when a vowel follows. The deletion rule is implemented with the two-level-rule operation `e<=><> (<Verb>:<> [ei])` which replaces "e" by the empty symbol if and only if the symbol `<VERB>` and an "e" or an "i" follows.

```
$verb-reg-infl$ = <Verb> ({<3s>}:{s} | {<past>}:{ed} |\
  {<part>}:{ed} |{<gerund>}:{ing} | {<n3s>}:{} | {<base>}:{})

$noun-reg-infl$ = <Noun> ({<sg>}:{} | {<pl>}:{s})

$morph$ = "verb-reg.lex" $verb-reg-infl$ | "noun-reg.lex" $noun-reg-infl$

ALPHABET = [A-Za-z] [<Verb><Noun>]:<> e:<>
$delete-e$ = e <=> <> (<Verb>:<> [ei])

$morph$ || $delete-e$
```

In contrast to the previous version, the `<Verb>` symbol is not immediately deleted at the surface level when the inflection transducer `$verb-reg-infl$` is created. Instead it is preserved here to indicate the position where the deletion rule is to be applied.

Without this marker, the rule would also delete an "e" in verbs such as *to feel* or *to unveil.* The deletion rule is applied by composing the morphology transducer stored in `$morph$` with the rule transducer stored in `$delete-e$`:

Exercise Add the lemma *paste* to the lexicon file *verb-reg.lex*, compile the above transducer, and use it to analyze and generate word forms.

Extend the program such that the word forms *later* and *latest* are correctly analyzed.

Two-level rules have to be preceded by the definition of an alphabet which comprises all symbol pairs that will appear in the two-level-rule transducer. The proper definition of the alphabet is very important. Omissions (or insertions) often have severe consequences. The above alphabet specifies that the `<Verb>` symbol is to be deleted. Therefore, we have to write `<Verb>:<>` rather than `<Verb>` in the context part of the rule. Otherwise, the rule would never match because a two-level rule specifies both, the analysis and the surface layer of the rule context.

Excursion The compiler translates two-level rules to complex transducer expressions. The rule `L a<=>b R` (where L and R are some transducer expressions) is first replaced by a conjunction of two more basic rules: `L a<=b R & L a=>b R`.

The rule `L a<=b R` means: *Whenever "a" appears between "L" and "R", it has to be mapped to "b".* (This rule still allows "a" to be mapped to "b" in other contexts.)

The rule `L a=>b R` means: *The mapping of "a" to "b" is only allowed if "L" precedes and "R" follows.* (It also allows that "a" is mapped to something else in the given context.)

The rule `L a<=b R` is equivalent to the expression `!(.*L (a:. & !a:b) R.*)`. The sub-expression `a:. & !a:b` represents the set of all mappings of "a" allowed by the alphabet except for the mapping "a:b"[3]. The expression`.*L (a:. & !a:b) R.*` comprises all mappings of strings allowed by the alphabet that include an illicit mapping of "a" to a symbol other than "b" in the context `L...R`. Its negation `!(.*L (a:. & !a:b) R.*)` includes all string mappings without such an illicit substring mapping.

The rule `L a=>b R` is replaced by the expression `!(!(.*L) a:b .* | .* a:b !(R.*))`. The sub-expression `!(.*L) a:b .*` includes all string mappings where the mapping a:b occurs with a wrong left context. `.* a:b !(R.*)` includes all mappings where the mapping a:b occurs with an incorrect right context. `!(!(.*L) a:b .* | .* a:b !(R.*))`. `!(.*L) a:b .*` therefore includes the set of string mappings allowed by the alphabet where the mapping a:b neither occurs with the wrong left context nor with the wrong right context. In other words, a:b only occurs with the correct left and right context.

[3] Note that the expression `a:. & !a:b` is not equivalent to `a:[^b]` because the latter expression includes symbol pairs which are not part of the alphabet.

"L" and "R" are any transducer expressions. They should always be enclosed in parentheses in order to avoid problems with operator precedence.

We can write a rule which is similar to the *delete-e* rule to replace "y" with "i" if "e" follows. This rule correctly maps the string *apply¡Verb¿ed* to *applied* (in generation mode).

```
$verb-reg-infl$ = <Verb> ({<3s>}:{s} | {<past>}:{ed} |\
  {<part>}:{ed} | {<gerund>}:{ing} | {<n3s>}:{} | {<base>}:{})
$noun-reg-infl$ = <Noun> ({<sg>}:{} | {<pl>}:{s})
$morph$ = "verb-reg.lex" $verb-reg-infl$ | "noun-reg.lex" $noun-reg-infl$

ALPHABET = [A-Za-z] [<Verb><Noun> e]:<> y:i

$delete-e$ = e <=> <> (<Verb>:<> [ei])
$y-to-i$ = y <=> i (<Verb>:<> e)

$morph$ || ($delete-e$ & $y-to-i$)
```

The above transducer combines the two-level rules with a conjunction operation (&).

Excursion The conjunction (or intersection) of two transducers T1 and T2 maps a string "s" to a string "t" iff both T1 and T2 map "s" to "t" *via the same alignment* of surface and analysis symbols. The last point is important. Although both of the following transducers produce the analysis *cat⟨N⟩⟨pl⟩* for the string *cats*, their conjunction is nevertheless empty:

```
$T1$ = cat<>:s<N>:<><pl>:<>
$T2$ = cat<N>:s<pl>:<>
$T1$ & $T2$
```

The combination of orthographic rules by conjunction often leads to unexpected interactions between the rules which are difficult to foresee, increase rapidly with the number of rules, and complicate the development process. Therefore it is usually better to combine two-level rules with composition (i.e. to apply them in sequence rather than in parallel).

The following transducer composes the two-level rules and is otherwise equivalent to the previous transducer. The part-of-speech labels are deleted in a separate step. Note that the two rules now require different alphabet definitions.

```
$verb-reg-infl$ = <Verb> ({<3s>}:{s} | {<past>}:{ed} |\
  {<part>}:{ed} | {<gerund>}:{ing} | {<n3s>}:{} | {<base>}:{})
$noun-reg-infl$ = <Noun> ({<sg>}:{} | {<pl>}:{s})
$morph$ = "verb-reg.lex" $verb-reg-infl$ | "noun-reg.lex" $noun-reg-infl$

ALPHABET = [A-Za-z] <Verb><Noun> e:<>
$delete-e$ = e <=> <> (<Verb> [ei])

ALPHABET = [A-Za-z] <Verb><Noun> y:i
$y-to-i$ = y <=> i (<Verb> e)

ALPHABET = [A-Za-z] [<Verb><Noun>]:<>
$delete-POS$ = .*

$morph$ || $delete-e$ || $y-to-i$ || $delete-POS$
```

This transducer still fails to produce the word form *applies*. In order to generate it, we need to substitute “y” with “ie”. A two-level rule can only substitute a single symbol with zero or one, but not with two symbols. Thus a more powerful *string replacement operation* (shown in the next example) has to be used instead.

```
...
ALPHABET = [A-Za-z] <Verb><Noun>
$y-to-ie$ = {y}:{ie} ^-> (__ [<Verb><Noun>] s)

$morph$ || $delete-e$ || $y-to-i$ || $y-to-ie$ || $delete-POS$
```

The expression `{y}:{ie} ^-> (__ [<Verb><Noun>] s)` defines a transducer which maps the string “y” to the string “ie” iff it is followed by either `<Noun>` or `<Verb>` and an “s”. Any other symbol than a “y” in this particular context is mapped according to the alphabet, i.e. mapped to itself. This rule will also produce the plural noun form *hobbies* if we add the lemma *hobby* to the lexicon file *noun-reg.lex*.

There are two important differences between the two-level rules and the replace rule: The context of the two-level rules specifies the symbols on the analysis level as well as on the surface level, whereas the replace operation only specifies the symbols on the analysis level.

The other difference between the two types of rules concerns the definition of the alphabet. The two-level rule `y <=> i (<Verb> e)` requires that the symbol pair y:i is added to the alphabet. The alphabet for the rule `{y}:{ie} ^-> (__ [<Verb><Noun>] s)`, on the other hand, only contains the symbol pairs appearing outside of the replaced substring(s).

Excursion Replace operations have the general form `T ^-> (L__R)`, where T is a transducer expression and L and R are automata, i.e. transducers which map strings only to themselves. A replace operation performs the following mapping: If T maps some string “s” to a string “t”, then the replace transducer substitutes any substring “s” of the input string with “t” (in generation mode) if and only if it appears with the left context L and right context R. Any other substring is mapped according to the alphabet.

The replace operator `^->` matches the contexts L and R with the symbols on the analysis level. There are three other, less frequently used variants of the replace operator, namely (1) `_->`, (2) `/->`, and (3) `\->` which match (1) the left and right contexts with the *surface* symbols, (2) the left context with the surface symbols and the right context with the analysis symbols, and (3) the left context with the analysis symbols and the right context with the surface symbols.

Another orthographic rule is needed to duplicate the final consonant in the comparative and superlative forms of adjectives such as *big*, *red*, *thin*, or *flat*. The following partial program shows one possible implementation of such a rule. It includes comments which start with the character “%” and extend up to the end of the line.

```
$verb-reg-infl$ = <Verb> ({<3s>}:{s} | {<past>}:{ed} | {<part>}:{ed} |\
  {<gerund>}:{ing} | {<n3s>}:{} | {<base>}:{})
$noun-reg-infl$ = <Noun> ({<sg>}:{} | {<pl>}:{s})
$adj-reg-infl$ = <Adj>\
  ({<pos>}:{} | {<comp>}:{er} | {<sup>}:{est})

$morph$ = "verb-reg.lex" $verb-reg-infl$ |\
  "noun-reg.lex" $noun-reg-infl$ | "adj-reg.lex"  $adj-reg-infl$

ALPHABET = [A-Za-z] <Verb><Noun><Adj> e:<>
$delete-e$ = e <=> <> (<Verb> [ei])

ALPHABET = [A-Za-z] <Verb><Noun><Adj> y:i
% also covers happy -> happily
$y-to-i$ = y <=> i ([<Verb><Adj>] [el])

ALPHABET = [A-Za-z] <Verb><Noun><Adj>
$y-to-ie$ = {y}:{ie} ^-> (__ [<Verb><Noun>] s)

ALPHABET = [A-Za-z] [<Verb><Noun><Adj>]:<>
$delete-POS$ = .*
```

```
% consonants
$c$ = [bcdfghjklmnprstvwxz]
% vowels
$v$ = [aeiou]
% duplication of the consonant
$T$ = b<>:b | d<>:d | g<>:g | l<>:l | m<>:m | n<>:n | p<>:p | t<>:t
ALPHABET = [A-Za-z] <Verb><Noun><Adj>
% the complete duplication rule
$duplicate$ = $T$ ^-> ($c$$v$ __ <Adj> e(r|st))

$R$ = $delete-e$ || $y-to-i$ || $y-to-ie$ || $duplicate$ || $delete-POS$
$morph$ || $R$
```

Exercise Add morpho-phonological rules for other phenomena such as the e-insertion in the plural forms of nouns ending with “s” or “x”.

4.5 Agreement Variables

Agreement variables are special *synchronized* variables whose name starts with the symbol “=”. All appearances of agreement variables in a regular expression are guaranteed to have identical values. The following program, for instance, defines a transducer which recognizes the strings *bib*, *did*, and *gig* and nothing else.

```
$=1$ = [bdg]
$=1$ i $=1$
```

The following transducer expression shows how agreement variables can

```
#=C# = bdglmnpt
$T$ = {[#=C#]}:{[#=C#][#=C#]}
```

be used to implement the duplication operation `$T$ = b<>:b | d<>:d |...` which was previously used to generate word forms such as *bigger*. We need a new type of variables here whose value is a set of symbols rather than a transducer. Such symbol set variables have to be enclosed with hash symbols. If the name begins with “=”, the variable is in addition an agreement variable.

4.6 Compounding

So far, we have only dealt with inflection. Another important morphological process is *compounding.* It creates words such as *whiteboard*, *sunlight*, or *workday* by concatenating two stems. Only the second stem is inflected. The following transducer will analyze these forms (if the missing stems are added to the respective lexicon files).

```
$noun-reg-infl$ = <Noun>:<> ({<sg>}:{} | {<pl>}:{s})
$noun-reg$ = \
  ("noun-reg.lex" | "verb-reg.lex" | "adj-reg.lex")? "noun-reg.lex"
$noun-reg$ $noun-reg-infl$
```

4.7 Derivation

Derivation is a third morphological process which generates new word forms by modifying a given stem usually with affixation. From the stem *reach*, we can derive the word *reachable* by adding the suffix *able.* Further adding the prefix *un*, we obtain *unreachable*, and with the suffix *ity*, we finally get *unreachability.* The first two derivation steps are purely concatenative, whereas the third step requires the mapping of *able* to *abil.*

The following transducer will correctly analyze the words *reachable* and *unreachable* if the stem *reach* is added to the verb lexicon.

```
...

% lexicon entries are read from the lexicon files
$verb-reg$ = "verb-reg.lex"
$noun-reg$ = "noun-reg.lex"
$adj-reg$  = "adj-reg.lex"

% derivation of adjectives from verbs
$adj-reg$ = $adj-reg$ | $verb-reg$ able
% prefixation of adjectives
$adj-reg$ = (un)? $adj-reg$

$morph$ = $noun-reg$ $noun-reg-infl$ |\
          $verb-reg$ $verb-reg-infl$ |\
          $adj-reg$  $adj-reg-infl$

$morph$ || $delete-e$ || $y-to-i$ || $y-to-ie$ || $duplicate$ || $delete-POS$
```

This SFST program will generate the incorrect form *translateable* instead of *translatable.* The verb-final “e” should be eliminated. We already have a rule which deletes “e” if the symbol `<Verb>` and either “e” or “i” follows. We must add the vowel “a” to this list. Furthermore, we have to make sure that the symbol `<Verb>` appears between *translate* and *able*, because the rule is not applicable, otherwise.

To this end, we reorganize the program. The part-of-speech markers are added at a different location. As a side-effect, we get slightly different compound analyzes: The word *whiteboards* will now be analyzed as `white<Adj>boards<Noun><pl>`. We also define variables for frequently used symbol sets which are then used throughout the program. This modification makes it easier to add new characters like é or ñ, or new part of speech symbols because it suffices to add them to the respective symbol set.

```
#cons# = bcdfghjklmnprstvwxz
#vowel# = aeiou
#letter# = a-z
#LETTER# = A-Z
#Letter# = #LETTER# #letter#
#pos# = <Adj><Noun><Verb>
#sym# = #Letter# #pos#

$verb-reg-infl$ = (\
  {<3s>}:{s} |\
  {<past>}:{ed} |\
  {<part>}:{ed} |\
  {<gerund>}:{ing} |\
  {<n3s>}:{} |\
  {<base>}:{})

$noun-reg-infl$ = (\
  {<sg>}:{} |\
  {<pl>}:{s})

$adj-reg-infl$ = (\
  {<pos>}:{} |\
  {<comp>}:{er} |\
  {<sup>}:{est})

% lexicon entries are read from the lexicon files
$verb-reg$ = "verb-reg.lex" <Verb>
$noun-reg$ = "noun-reg.lex" <Noun>
$adj-reg$  = "adj-reg.lex" <Adj>

% noun compounds
$noun-reg$ = ($noun-reg$ | $verb-reg$ | $adj-reg$)? $noun-reg$
```

```
% derivation of adjectives from verbs
$adj-reg$ = $adj-reg$ | $verb-reg$ able
% prefixation of adjectives
$adj-reg$ = (un)? $adj-reg$

$morph$ = $noun-reg$ $noun-reg-infl$ |\
          $verb-reg$ $verb-reg-infl$ |\
          $adj-reg$  $adj-reg-infl$


ALPHABET = [#sym#] e:<>
$delete-e$ = e <=> <> (<Verb> [aei])

ALPHABET = [#sym#] y:i
$y-to-i$ = y <=> i ([<Verb><Adj>] [el])

ALPHABET = [#sym#]
$y-to-ie$ = {y}:{ie} ^-> (__ [<Verb><Noun>] s)

#=D# = bdglmnpt
$T$ = {[#=D#]}:{[#=D#][#=D#]}

ALPHABET = [#sym#]
$duplicate$ = $T$ ^-> ([#cons#][#vowel#] __ <Adj> e(r|st))

ALPHABET = [#Letter#] [#pos#]:<>
$delete-POS$ = .*

$R$ = $delete-e$ || $y-to-i$ || $y-to-ie$ || $duplicate$ || $delete-POS$

$morph$ || $R$
```

Exercise Extend the morphology with a rule for the derivation of adverbs (*quickly*) from adjectives (*quick*).

4.8 Irregular Inflection

Thus far, the morphology covers regular inflection such as *edit/edited/editing*, or *paste/pasting/pasted*, but not irregular inflection such as *throw/threw/thrown* or *admit/admitting/admitted.*

The inflection of the verb *admit* can be handled with rules if stress information is encoded in the lexicon[4]. If not, a separate inflection class has to be defined, instead. The following fragment shows how to do it.

```
...
#pos# = <Adj><Noun><Verb>
#trigger# = <dup>
#mcs# = #pos# #trigger#
#sym# = #Letter# #mcs#
...
$verb-dup-infl$ = (\
  {<3s>}:{s} |\
  {<past>}:{<dup>ed} |\
  {<part>}:{<dup>ed} |\
  {<gerund>}:{<dup>ing} |\
  {<n3s>}:{} |\
  {<base>}:{})
...
$verb-dup$ = "verb-dup.lex" <Verb>
...
ALPHABET = [#sym#]
$duplicate$ = $T$ ^-> ([#cons#][#vowel#] __ (<Adj> e(r|st) | <Verb><dup>))

ALPHABET = [#Letter#] [#mcs#]:<>
$delete-POS$ = .*
...
```

The inflection rules for the new verb class *verb-dup* insert the symbol `<dup>` in the gerund, past tense, and past participle forms. This symbol triggers the application of the modified `duplicate` rule. It is later deleted by the `delete-pos` rule and doesn't appear in the result transducer. The insertion of trigger symbols is a frequently used trick in finite-state morphology.

If some verb forms are completely irregular such as *sit/sat/sat*, it is often easier to add special lexical entries for the irregular forms than to implement complex morphophonological rules. We could define two new inflection classes: a class `verb-pres-ger2` which derives *sit/sits/sitting* from the lexicon entry *sit* and another class `verb-past-part` with the entry `si:at` for the past tense and past participle forms.

```
...
$verb-pres-ger2-infl$ = (\
  {<3s>}:{s} |\
```

[4] The consonant has to be reduplicated if the last syllable is stressed.

```
  {<gerund>}:{<dup>ing} |\
  {<n3s>}:{} |\
  {<base>}:{})

$verb-past-part-infl$ = (\
  {<past>}:{} |\
  {<part>}:{})
...
$verb-pres-ger2$ = "verb-pres-ger2.lex" <Verb>
$verb-past-part$ = "verb-past-part.lex" <Verb>
...
$morph$ = ...
          $verb-pres-ger2$ $verb-pres-ger2-infl$ |\
          $verb-past-part$ $verb-past-part-infl$ |\
...
```

Exercise Extend the morphology with two new inflection classes, one to generate the forms *throw/throws/throwing/thrown*, *see/sees/seeing/seen*, and *fall/falls/falling/fallen* and another class for the past tense forms *threw*, *saw*, and *fell.*

You have to add a morpho-phonological rule to insert the "e" in *fallen.* Note that you cannot use two-level rules and replace rules to replace the empty symbol with "e". Use a rule replacing `n` with `en`, instead.

4.9 A Single Lexicon

Having a separate lexicon file for each inflection class is not ideal. A single lexicon file is easier to maintain. Therefore we merge all the sublexica and add a multi-character symbol to each entry which specifies the inflection class. The new lexicon `morph.lex` looks as follows:

```
board<N-reg>
...
big<A-reg>
...
apply<V-reg>
...
admit<V-dup>
...
sit<V-pres-ger2>
```

```
...
si:at<V-past-part>
```

Now, the transducer program has to be modified. We split it into several smaller sections. The main file morph.fst includes the auxiliary file symbols.fst via the command #include "symbols.fst" which literally inserts the contents of symbols.fst at the current position.

The other new files, inflection.fst and phon.fst, are separately compiled. The main file includes the compiled binary transducers with the expressions "<inflections.a>" and "<phon.a>". The separate compilation of different parts of the transducer program speeds up the whole compilation process if the recompilation of transducers whose source files are unchanged is avoided.

The file symbols.fst contains the definitions of the symbol set variables. Corresponding transducer variables are also added. The new variable #infl# comprises the set of inflection class labels.

The inflectional endings for each inflection class are defined in inflection.fst. The endings now include inflectional class labels such as V-reg. All endings are merged into a single transducer which is stored in the result file inflection.a.

The main program morph.fst reads the lexicon from morph.lex and composes the resulting transducer with the expression $Letter$* <>:[#infl#] in order to delete the inflection class labels in the analysis layer.

The next line concatenates the lexicon transducer and the pre-compiled inflection transducer which is read from inflection.a. The result includes many incorrect combinations of stems and endings such as clean<>:<A-reg><>:<N-reg><Noun><sg>:<>.

These will be eliminated by the inflection filter which is defined next. The filter transducer maps a sequence of letters followed by two identical inflectional class labels and arbitrary other symbols to a new string where the class labels have been deleted. The composition of the filter transducer with the previous transducer returns a new transducer containing only the correct combinations of stems and endings. The other combinations are rejected by the filter transducer.

The last command composes the morphology transducer with the morpho-phonological rules and returns the result transducer.

```
%%%%%%%%%%%%%%% symbols.fst %%%%%%%%%%%%%%%%

#cons# = bcdfghjklmnprstvwxz
#vowel# = aeiou
#letter# = a-z
#LETTER# = A-Z
#Letter# = #LETTER# #letter#
```

```
#pos# = <Adj><Noun><Verb>
#infl# = <A-reg><N-reg><V-reg><V-dup><V-pres-ger2><V-past-part>
#trigger# = <dup>
#mcs# = #pos# #trigger#
#sym# = #Letter# #mcs# #infl#

$cons$ = [#cons#]
$vowel$ = [#vowel#]
$letter$ = [#letter#]
$LETTER$ = [#LETTER#]
$Letter$ = [#Letter#]
$pos$ = [#pos#]
$infl$ = [#infl#]
$trigger$ = [#trigger#]
$mcs$ = [#mcs#]
$sym$ = [#sym#]

%%%%%%%%%%%%%% inflection.fst %%%%%%%%%%%%%%

$noun-reg-infl$ = <>:<N-reg> <Noun> (\
  {<sg>}:{} |\
  {<pl>}:{s})

$adj-reg-infl$ = <>:<A-reg> <Adj> (\
  {<pos>}:{} |\
  {<comp>}:{er} |\
  {<sup>}:{est})

$verb-reg-infl$ = <>:<V-reg> <Verb> (\
  {<3s>}:{s} |\
  {<past>}:{ed} |\
  {<part>}:{ed} |\
  {<gerund>}:{ing} |\
  {<n3s>}:{} |\
  {<base>}:{})

$verb-dup-infl$ = <>:<V-dup> <Verb> (\
  {<3s>}:{s} |\
  {<past>}:{<dup>ed} |\
  {<part>}:{<dup>ed} |\
  {<gerund>}:{<dup>ing} |\
  {<n3s>}:{} |\
  {<base>}:{})
```

```
$verb-pres-ger2-infl$ = <>:<V-pres-ger2> <Verb> (\
  {<3s>}:{s} |\
  {<gerund>}:{<dup>ing} |\
  {<n3s>}:{} |\
  {<base>}:{})

$verb-past-part-infl$ = <>:<V-past-part> <Verb> (\
  {<past>}:{} |\
  {<part>}:{})

$noun-reg-infl$ | $adj-reg-infl$ | $verb-reg-infl$ |\
$verb-dup-infl$ | $verb-pres-ger2-infl$ | $verb-past-part-infl$


%%%%%%%%%%%%%%%% phon.fst %%%%%%%%%%%%%%%%%%

#include "symbols.fst"

ALPHABET = [#sym#] e:<>
$delete-e$ = e <=> <> (<Verb> [aei])

ALPHABET = [#sym#] y:i
$y-to-i$ = y <=> i ([<Verb><Adj>] [el])

ALPHABET = [#sym#]
$y-to-ie$ = {y}:{ie} ^-> (__ [<Verb><Noun>] s)

#=D# = bdglmnpt
$T$ = {[#=D#]}:{[#=D#][#=D#]}

ALPHABET = [#sym#]
$duplicate$ = $T$ ^-> ([#cons#][#vowel#] __ (<Adj> e(r|st) | <Verb><dup>))

ALPHABET = [#Letter#] [#mcs#]:<>
$delete-POS$ = .*

$delete-e$ || $y-to-i$ || $y-to-ie$ || $duplicate$ || $delete-POS$


%%%%%%%%%%%%%%% morph.fst %%%%%%%%%%%%%%%%%%

#include "symbols.fst"

% Read the lexicon and
% delete the inflection class  on the analysis layer

$lex$ = ($Letter$* <>:[#infl#]) || "morph.lex"

% Concatenate stems with the inflectional endings

$morph$ = $lex$ "<inflection.a>"

% Eliminate incorrect combinations with a filter transducer

$=C$ = [#infl#]:<>
$inflection-filter$ = $Letter$+ $=C$ $=C$ $sym$*

$morph$ = $morph$ || $inflection-filter$

$morph$ || "<phon.a>"
```

Exercise Store the above code in the files `symbols.fst`, `inflection.fst`, `phon.fst`, and `morph.fst`, respectively. Create a lexicon file `morph.lex` with entries as shown before. Use `fst-compiler` to create `inflection.a`, `phon.a`, and `morph.a`. Start `fst-mor morph.a` and try to analyze different word forms.

The Unix utility `make` can be used to determine automatically which transducers need to be recompiled. `make` needs a control file which looks as follows:

```
morph.a: symbols.fst inflection.a phon.a morph.lex

phon.a: symbols.fst

%.a: %.fst
        fst-compiler $< $@
```

The first command tells `make` that the file `morph.a` depends on the files `symbols.fst`, `inflection.a`, `morph.lex`, and `phon.a`. Whenever one of these files changes, `morph.a` needs to be recompiled. The second command adds the dependency of `phon.a` on `symbols.fst`. The third command consists of two lines and informs the `make` program that a file such as `foo.a` is produced from a corresponding file `foo.fst` by calling `fst-compiler foo.fst foo.a`.

The new version of the morphology no longer covers derived word forms and compounds. This functionality will be added in the next section.

Exercise Store the above control data in a file called `makefile` and call `make`. If all the files are up to date, nothing is done. Otherwise, `make` calls `fst-compiler` to update the compiled transducer files.

4.10 Compounding Revisited

German is a language which is well-known for the complexity of its compounds. A moderate example is the word *Bundesausbildungsf”orderungsgesetz* (federal education advancement law). English is less productive in this respect. Nevertheless, the compounding mechanism that we will implement next is able to produce English compounds of similar complexity.

The compounds are generated by the following code which is stored in the file `compounding.fst`. The second command reads the lexicon, deletes the inflection labels in the surface layer, and replaces the inflection labels in the analysis layer with part-of-speech symbols. The following command extracts nouns and adjectives from the lexicon stored in *lex*. Compounds are created by concatenating one or more compounding stems (`$T1$+`) and a noun or adjective entry (`$T2$`). The compounds are then added to the other lexicon entries.

The compounding code is included in `morph.fst` via an include command, as shown below.

```
%%%%%%%%%%%% compounding.fst %%%%%%%%%%%%%%%

% generate compounding stems

$T$ = <Adj>:<A-reg> | <Noun>:<N-reg> | <Verb>:[<V-reg><V-dup><V-pres-ger2>]
$T1$ = ($Letter$* $T$) || "morph.lex" || ($Letter$* [#infl#]:<>)

% extract adjective and noun entries

$T2$ = $lex$ || ($Letter$+ [<A-reg><N-reg>])

% produce the compounds

$comp$ = $T1$+ $T2$

% add the compounds to the lexicon

$lex$ = $lex$ | $comp$
```

```
%%%%%%%%%%%%%%% morph.fst %%%%%%%%%%%%%%%%%%
...
$lex$ = ($Letter$* <>:[#infl#]) || "morph.lex"
#include "compounding.fst"
...
```

Exercise Extend the `makefile` such that `morph.a` is recompiled when `compounding.fst` is modified. Compile the morphology and try to analyze different compounds.

4.11 Derivation Revisited

Many Germanic languages possess two different derivational mechanisms. The *native* derivation of word forms is exemplified by the English noun *darkness* which is obtained by adding the suffix *-ness* to the adjective *dark*. Similarly, the *neo-classical* nominalization *reality* is composed of the adjective *real* and the suffix *-ity*. Neo-classical word formation is mostly used with loan words of Latin origin.

We have to make sure that our morphology only combines neo-classical affixes with neo-classical stems and native affixes with native stems. Thus, neither *darkity* nor *realness* should be generated.

We add special entries for derivation stems and derivation suffixes as shown below. Derivation stems are annotated with their part of speech and origin. Lexicon entries for derivation suffixes start with the part of speech and the origin feature of the derivation stem that they combine with, and they end with the label of the inflectional class of the resulting word form.

```
...
dark<Adj><native>
real<Adj><classic>
<Adj><native>ness<N-reg>
<Adj><classic>ity<N-reg>
```

The morphology program is extended with the new source code file `derivation.fst` which is included in the main file `morph.fst` as shown below.

```
%%%%%%%%%%%%% derivation.fst %%%%%%%%%%%%%%%
```

```
% extract derivation stems and suffixes

$derivstems$ = $Letter$+ $pos$ <>:[#origin#] || "morph.lex"
$suffixes$ = <>:[#pos#]<>:[#origin#] $Letter$+ <>:[#infl#] || "morph.lex"

$suffixes$ = <Suff>:<> $suffixes$

% concatenate derivation stems and suffixes and

$deriv$ = $derivstems$ $suffixes$

% filter out incorrect combinations

$=pos$ = [#pos#]:<>
$=origin$ = [#origin#]:<>
$filter$ = $sym$* $=pos$ $=origin$ $=pos$ $=origin$ $sym$* $infl$

$deriv$ = $deriv$ || $filter$

% add the derived stems to the lexicon

$lex$ = $lex$ | $deriv$

%%%%%%%%%%%%%%% morph.fst %%%%%%%%%%%%%%%%%%
...
#include "derivation.fst"
#include "compounding.fst"
...
```

Exercise Extend the morphology such that the word *reachability* receives the analysis:

```
reach<Verb><Suff>abel<Adj><Suff>ity<Noun><sg>
```

The lexicon entry for the first derivational suffix might look as follows:

```
<V><native>ab<>:ile:<><Adj><classic>
```