Finite-state transducers for phonology and morphology

A motivating example:

Example:

Consider the following pairs:

accurate	*adequate*	*balanced*	*competent*
inaccurate	*inadequate*	*imbalanced*	*incompetent*
definite	*finite*	*mature*	*nutrition*
indefinite	*infinite*	*immature*	*innutrition*
patience	*possible*	*sane*	*tractable*
impatience	*impossible*	*insane*	*intractable*

The negative forms are constructed by adding the abstract morpheme iN to the positive forms. N is realized as either n or m.

A linguistically accurate description of this phenomenon could be:

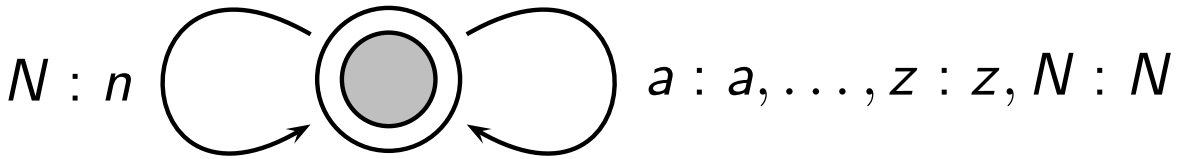
Rule 1: `N -> m || _ [b|m|p]`

Rule 2: `N -> n`

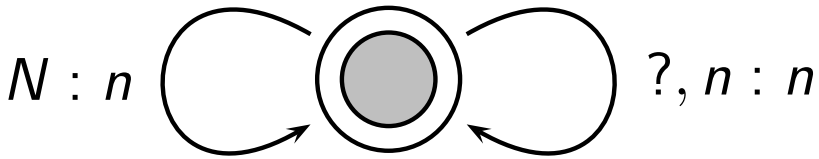
The rules must be interpreted as *obligatory* and applied in the *order* given.

Finite-state transducers for phonology and morphology

A finite-state transducer for Rule 2 (interpreted as optional):



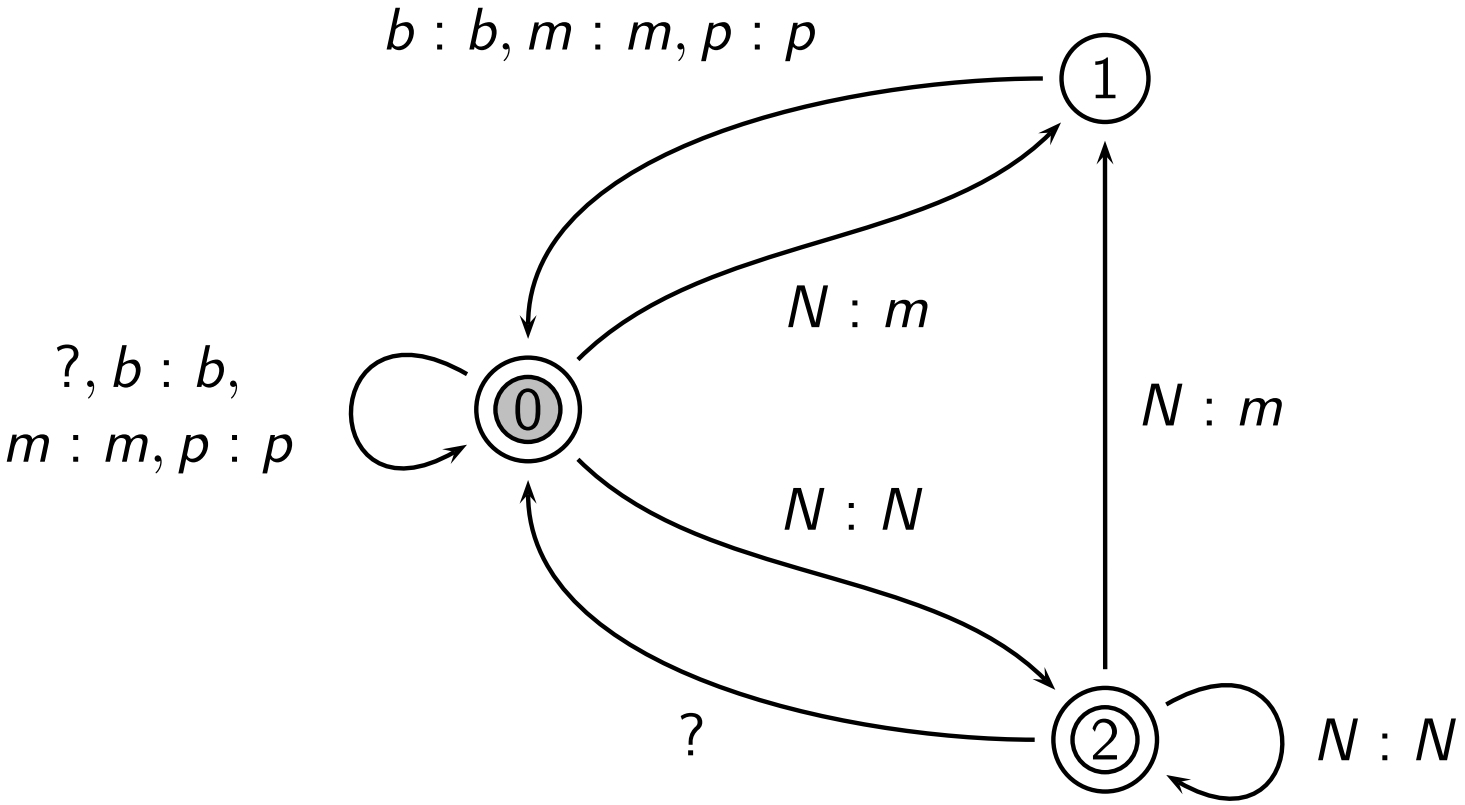
A finite-state transducer for Rule 2 (interpreted as obligatory):



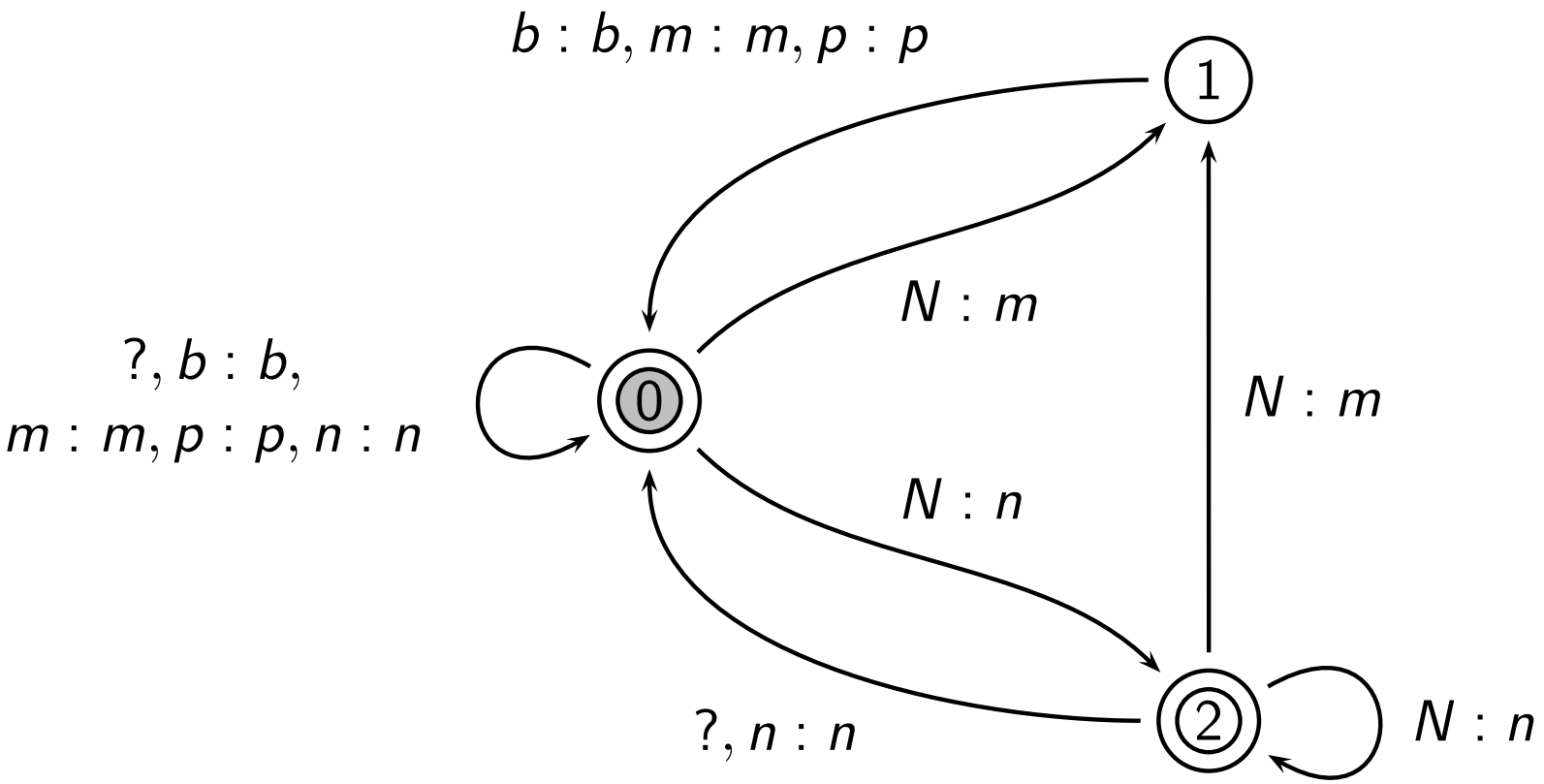
where '?' stands for "any symbol that does not occur elsewhere in this rule."

Finite-state transducers for phonology and morphology

A finite-state transducer for Rule 1 (interpreted as obligatory):



Composition of obligatory Rules 1 and 2:



Implementing composition

Regular relations are closed under composition.
Recall that $R_1 \circ R_1 =$

$$\{\langle x, y\rangle \mid \text{ there exists } z \text{ such that } \langle x, z\rangle \in R_1 \text{ and } \langle z, y\rangle \in R_2\}$$

Let $T_1 = (Q_1, q_1, \Sigma_1, \Sigma_2, \delta_1, F_1)$ and $T_2 = (Q_2, q_2, \Sigma_2, \Sigma_3, \delta_2, F_2)$.
Then

$$T_1 \circ T_2 = T = (Q_1 \times Q_2, \langle q_1, q_2\rangle, \Sigma_1, \Sigma_3, \delta, F_1 \times F_2),$$

where $\delta(\langle s_1, s_2\rangle, a, b) =$

$$\{\langle t_1, t_2\rangle \mid \text{ for some } c \in \Sigma_2 \cup \{\epsilon\}, t_1 \in \delta_1(s_1, a, c) \text{ and } t_2 \in \delta_2(s_2, c, b)\}$$

This definition is flawed! See Challenge 2

Implementing replace rules

We begin with the simplest rule: unconditional, obligatory replacement: `UPPER -> LOWER`.

Example:

- `a -> b`

bcd	abac	aaab	
bcd	bbbc	bbbb	

- `a+ -> b`

bcd	abac	aaab	aaab	aaab
bcd	bbbc	bbbb	bbb	bb

- `[[a b] | [b c]] -> x`

abc	abc
ax	xc

Implementing replace rules

Incremental construction:

- `$A`: The language of all strings which contain, as a substring, a string from `A`. Equivalent to `?* A ?*`.
- `noUPPER`: Strings which do not contain strings from `UPPER`. Naïvely, this would be defined as `~$UPPER`. However, if `UPPER` happens to include the empty string, then `~$UPPER` is empty. A better definition is therefore `~$[UPPER - []]`, which is identical except that if `UPPER` includes the empty string, `noUPPER` includes at least the empty string.

Implementing replace rules

- `UPPER -> LOWER`: Defined as

  `[noUPPER [UPPER .x. LOWER]]* noUPPER`

  A regular relation whose members include any number (possibly zero) of iterations of `[UPPER .x. LOWER]`, interleaved with strings that do not contain `UPPER` which are mapped to themselves.

Implementing replace rules

Special cases:

Example:

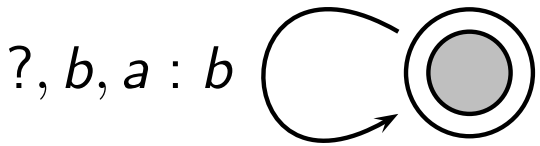
- `[] -> a | b`
  A transducer that freely inserts as and bs in the input string.
- `~$[] -> a | b`
  No replacement; equivalent to `?*`.
- `a | b -> []`
  Removes all as and bs from the input string.
- `a | b -> ~$[]`
  Strings containing as or bs are excluded from the upper language. Equivalent to `~$[a | b]`.

Implementing replace rules

Inverse replacement: `UPPER <- LOWER`.

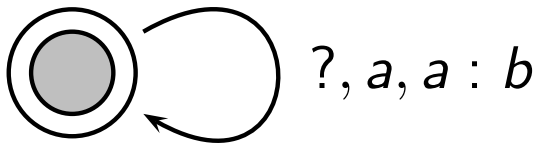
- Meaning: obligatory unconditional inverse replacement.
- Definition and construction: `[LOWER -> UPPER].i`
- (`A.i` is the inverse relation of `A`)
- The difference between `UPPER -> LOWER` and `UPPER <- LOWER`:

$a \rightarrow b$




```
bb ab
bb bb
```

$a \leftarrow b$




```
ab ab
ab bb
```

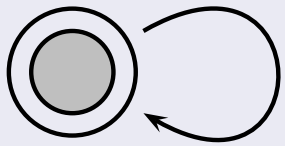
Implementing replace rules

Optional replacement: `UPPER (->) LOWER`.

- Meaning: optionally replace occurrences of strings from `UPPER` by strings from `LOWER`.

Example:

Example: $a\ (\rightarrow)\ b$



$?, a, b, a : b$

```
ab ab
ab bb
```

- The tempting (and incorrect) construction: `[UPPER -> LOWER] | [UPPER]`.
- The correct construction: `[?* [UPPER .x. LOWER]]* ?*` or `[ $[UPPER -> LOWER] ]*`.

Conditional replacement

Conditional replacement: `UPPER -> LOWER || L _ R`.

- Meaning: Replace occurrences of strings from `UPPER` by strings from `LOWER` in the context of `L` on the left and `R` on the right.
- Outstanding issue:
  - the interpretation of "between `L` and `R`"
  - interactions between multiple applications of the same rule at different positions in a string: the part that is being replaced may at the same time serve as the context of another adjacent replacement
  - optionality

Conditional replacement

Conditional replacement:

Example:

- Upward oriented: `a b -> x || a b _ a`

  a b a b a b a

  a b x x a

- Downward oriented: `a b -> x \/ a b _ a`

  a b a b a b a a b a b a b a

  a b x a b a a b a b x a

- Right oriented: `a b -> x // a b _ a`

  a b a b a b a

  a b x a b a

- Left oriented: `a b -> x \\ a b _ a`

  a b a b a b a

  a b a b x a

Implementing conditional replacement

We focus on upward oriented, mandatory replacement rules:
`UPPER -> LOWER || L _ R`.

The idea: make the conditional replacement behave exactly like unconditional replacement except that the operation does not take place unless the specified context is present.

The problem: the part being replaced can be at the same time the context of another replacement.

The solution: first, decompose the complex relation into a set of relatively simple components, define them independently of one another, and finally use composition to combine them.

Implementing conditional replacement

Construction:

1. InsertBrackets
2. ConstrainBrackets
3. LeftContext
4. RightContext
5. Replace
6. RemoveBrackets

Implementing conditional replacement

- Two bracket symbols, < and >, are introduced in (1) and (6).
- < indicates the end of a complete left context. > indicates the beginning of a complete right context.
- Their distribution is controlled by (2), (3) and (4). (2) constrains them with respect to each other, whereas (3) and (4) constrain them with respect to the left and right context.
- The replacement expression (5) includes the brackets on both sides of the relation.
- The final result of the composition does not contain any brackets. (1) removes them from the upper side, (6) from the lower side.

Implementing conditional replacement

- Let < and > be two symbols not in Σ.
- InserBrackets eliminates from the upper side language all context markers that appear on the lower side.
- `InsertBrackets = [ ] <- [< | >]`.
- ConstrainBrackets denotes the language consisting of strings that do not contain <> anywhere.
- `ConstrainBrackets = ~$[ < > ]`.

Implementing conditional replacement

- LeftContext denotes the language in which any instance of < is immediately preceded by `L` and every `L` is immediately followed by <, ignoring irrelevant brackets.
- `[...L]` denotes the language of all strings ending in `L`, ignoring all brackets except for a final <.
- `[...L]` is defined as `[ ?* L/[<|>] ] - [?* <]`.
- (`A/B` is `A` ignore `B`, the language obtained by splicing in strings from `B*` anywhere within strings from `A`)
- Next, `[<...]` denotes the language of strings beginning with <, ignoring the other bracket: `[ </> ?*]`.
- Finally, `LeftContext` is defined as:

```
~[ ~[...L] [<...] ] & ~[ [...L] ~[<...] ] .
```

Implementing conditional replacement

- RightContext denotes the language in which any instance of > is immediately followed by R and any R is immediately preceded by >, ignoring irrelevant brackets.
- [R...] denotes the language of all strings beginning with R, ignoring all brackets except for an initial >.
- [R...] is defined as [R/[<|>] ?*] - [> ?*].
- [...>] denotes the language of strings ending with >, ignoring the other bracket.
- [...>] is defined as [?* >/<].
- Finally, RightContext is defined as

```
~[ [...>] ~[R...] ] & ~[ ~[...>] [R...] ] .
```

Implementing conditional replacement

- `Replace` is the unconditional replacement of `<UPPER>` by `<LOWER>`, ignoring irrelevant brackets.
- `Replace` is defined as:

  ```
  < UPPER/[<|>] > -> < LOWER/[</>] >
  ```

- `RemoveBrackets` denotes the relation that maps the strings of the upper language to the same strings without any context markers.
- `RemoveBrackets=[ <|> ] -> [].`

Implementing conditional replacement

Construction:

```
InsertBrackets
.o.
ConstrainBrackets
.o.
LeftContext
.o.
RightContext
.o.
Replace
.o.
RemoveBrackets
```

Implementing conditional replacement

Special cases:

- $A = \{\epsilon\}$ or $\epsilon \in A$.
- Boundary symbol (.#.): `L _ R` actually means `?* L _ R ?*`.

Introduction to XFST

- XFST is an interface giving access to finite-state operations (algorithms such as union, concatenation, iteration, intersection, composition etc.)
- XFST includes a regular expression compiler
- The interface of XFST includes a lookup operation (*apply up*) and a generation operation (*apply down*)
- The regular expression language employed by XFST is an extended version of standard regular expressions

Introduction to XFST

`a`	a simple symbol
`c a t`	a concatenation of three symbols
`[c a t]`	grouping brackets
`cat`	a single multicharacter symbol
`?`	denotes any single symbol
`%+`	the literal plus-sign symbol
`%*`	the literal asterisk symbol (and similarly for `%?`, `%(`, `%]` etc.)
`` ``+Noun'' ```	single symbol with multicharacter print name
`%+Noun`	single symbol with multicharacter print name

Introduction to XFST

`{cat}`	equivalent to `[c a t]`
`[ ]`	the empty string
`0`	the empty string
`[A]`	bracketing; equivalent to `A`
`A|B`	union
`(A)`	optionality; equivalent to `[A|0]`
`A&B`	intersection
`A B`	concatenation
`A-B`	set difference

Introduction to XFST

`A*`	Kleene-star
`A+`	one or more iterations
`?*`	the universal language
`~A`	the complement of `A`; equivalent to `[?* - A]`
`~[?*]`	the empty language

Introduction to XFST – denoting relations

`A .x. B`	Cartesian product; relates every string in `A` to every string in `B`
`a:b`	shorthand for `[a .x. b]`
`%+Pl:s`	shorthand for `[%+Pl .x. s]`
`%+Past:ed`	shorthand for `[%+Past .x. ed]`
`%+Prog:ing`	shorthand for `[%+Prog .x. ing]`

Introduction to XFST – useful abbreviations

`$A`	the language of all the strings that contain `A`; equivalent to `[?* A ?*]`
`A/B`	the language of all the strings in `A`, ignoring any strings from `B`
`a*/b`	includes strings such as `a, aa, aaa, ba, ab,` aba etc.
`\A`	any single symbol, minus strings in `A`. Equivalent to `[?  - A]`
`\b`	any single symbol, except 'b'. Compare to:
`~A`	the complement of `A`, i.e., `[?* - A]`

Introduction to XFST – user interface

```
prompt% H:\class\data\shuly\xfst

xfst> help
xfst> help union net
xfst> exit
xfst> read regex [d o g | c a t];
xfst> read regex < myfile.regex
xfst> apply up dog
xfst> apply down dog
xfst> pop stack
xfst> clear stack
xfst> save stack myfile.fsm
```

Introduction to XFST – example

```
[ [l e a v e %+VBZ .x. l e a v e s] |
  [l e a v e %+VB .x. l e a v e] |
  [l e a v e %+VBG .x. l e a v i n g] |
  [l e a v e %+VBD .x. l e f t] |
  [l e a v e %+NN .x. l e a v e] |
  [l e a v e %+NNS .x. l e a v e s] |
  [l e a f %+NNS .x. l e a v e s ] |
  [l e f t %+JJ .x. l e f t]
]
```

Introduction to XFST – example of lookup and generation

```
APPLY DOWN> leave+VBD
left
APPLY UP> leaves
leave+NNS
leave+VBZ
leaf+NNS
```

Introduction to XFST – variables

```
xfst> define Myvar;
xfst> define Myvar2 [d o g | c a t];
xfst> undefine Myvar;
```

```
xfst> define var1 [b i r d | f r o g | d o g];
xfst> define var2 [d o g | c a t];
xfst> define var3 var1 | var2;
xfst> define var4 var1 var2;
xfst> define var5 var1 & var2;
xfst> define var6 var1 - var2;
```

Introduction to XFST – variables

```
xfst> define Root [w a l k | t a l k | w o r k];
xfst> define Prefix [0 | r e];
xfst> define Suffix [0 | s | e d | i n g];
xfst> read regex Prefix Root Suffix;
xfst> words
xfst> apply up walking
```

Introduction to XFST – replace rules

Replace rules are an extremely powerful extension of the regular expression metalanguage.
The simplest replace rule is of the form

$$upper \rightarrow lower \parallel leftcontext \_ rightcontext$$

Its denotation is the relation which maps string to themselves, with the exception that an occurrence of *upper* in the input string, preceded by *leftcontext* and followed by *rightcontext*, is replaced in the output by *lower*.

Introduction to XFST – replace rules

Word boundaries can be explicitly referred to:

```
xfst> define Vowel [a|e|i|o|u];
xfst> e -> ’ || [.#.] [c | d | l | s] _ [% Vowel];
```

Introduction to XFST – replace rules

Contexts can be omitted:

```
xfst> define Rule1 N -> m || _ p ;
xfst> define Rule2 N -> n ;
xfst> define Rule3 p -> m || m _ ;
```

This can be used to clear unnecessary symbols introduced for "bookkeeping":

```
xfst> define Rule1 %^MorphmeBoundary -> 0;
```

Introduction to XFST – replace rules

The language Bambona has an underspecified nasal morpheme *N* that is realized as a labial *m* or as a dental *n* depending on its environment: *N* is realized as *m* before *p* and as *n* elsewhere. The language also has an assimilation rule which changes *p* to *m* when the *p* is followed by *m*.

```
xfst> clear stack ;
xfst> define Rule1 N -> m || _ p ;
xfst> define Rule2 N -> n ;
xfst> define Rule3 p -> m || m _ ;
xfst> read regex Rule1 .o. Rule2 .o. Rule3 ;
```

Introduction to XFST – replace rules

Rules can define multiple replacements:

```
[ A -> B, B -> A ]
```

or multiple replacements that share the same context:

```
[ A -> B, B -> A || L _ R ]
```

or multiple contexts:

```
[ A -> B || L1 _ R1, L2 _ R2 ]
```

or multiple replacements and multiple contexts:

```
[ A -> B, B -> A ||  L1 _ R1, L2 _ R2 ]
```

Introduction to XFST – replace rules

Rules can apply in parallel:

```
xfst> clear stack
xfst> read regex a -> b .o. b -> a ;
xfst> apply down abba
aaaa
xfst> clear stack
xfst> read regex b -> a .o. a -> b ;
xfst> apply down abba
bbbb
xfst> clear stack
xfst> read regex a -> b , b -> a ;
xfst> apply down abba
baab
```

Introduction to XFST – replace rules

When rules that have contexts apply in parallel, the rule separator is a double comma:

```
xfst> clear stack
xfst> read regex
b -> a || .#. s ?* _ ,, a -> b || _ ?* e .#. ;
xfst> apply down sabbae
sbaabe
```

Introduction to XFST – English verb morphology

```
define Vowel [a|e|i|o|u] ;
define Cons [? - Vowel] ;
define base [{dip} | {print} | {fuss} | {toss} | {bake} | +
define suffix [0 | {s} | {ed} | {ing}] ;
define form [base %+ suffix];
define FinalS [s %+ s] -> [s %+ e s] ;
define FinalE e -> 0 || _ %+ [e | i];
define DoubleFinal b -> [b b], d -> [d d], f -> [f f],
       g -> [g g], k -> [k k], l -> [l l], m -> [m m],
       n -> [n n], p -> [p p], r -> [r r], s -> [s s],
       t -> [t t], z -> [z z] || Cons Vowel _ %+ Vowel ;
define RemovePlus %+ -> 0 ;
read regex form .o. DoubleFinal .o. FinalS .o. FinalE
                .o. RemovePlus;
```

Introduction to XFST – English spell checking

```
clear;
define A [i e] -> [e i] || c _ ,,
         [e i] -> [i e] || [? - c] _ ;

define B [[e i] -> [i e]] .o.
         [[i e] -> [e i] || c _];
read regex B;
```

Introduction to XFST – marking

The special symbol "..." in the right-hand side of a replace rule stands for whatever was matched in the left-hand side of the rule.

```
xfst> clear stack;
xfst> read regex [a|e|i|o|u] -> %[ ... %];
xfst> apply down unnecessarily
[u]nn[e]c[e]ss[a]r[i]ly
```

Introduction to XFST – marking

```
xfst> clear stack;
xfst> read regex [a|e|i|o|u]+ -> %[ ... %];
xfst> apply down feeling
f[e][e]l[i]ng
f[ee]l[i]ng
xfst> apply down poolcleaning
p[o][o]lcl[e][a]n[i]ng
p[oo]lcl[e][a]n[i]ng
p[o][o]lcl[ea]n[i]ng
p[oo]lcl[ea]n[i]ng
xfst> read regex [a|e|i|o|u]+ @-> %[ ... %];
xfst> apply down poolcleaning
p[oo]lcl[ea]n[i]ng
```

Introduction to XFST – shallow parsing

Assume that text is represented as strings of part-of-speech tags, using ‘d’ for determiner, ‘a’ for adjective, ‘n’ for noun, and ‘v’ verb, etc. In other words, in this example the regular expression symbols represent whole words rather than single letters in a text.
Assume that a noun phrase consists of an optional determiner, any number of adjectives, and one or more nouns:

```
[(d) a* n+]
```

This expression denotes an infinite set of strings, such as “n” (cats), “aan” (discriminating aristocratic cats), “nn” (cat food), “dn” (many cats), “dann” (that expensive cat food) etc.

Introduction to XFST – shallow parsing

A simple noun phrase parser can be thought of as a transducer that inserts markers, say, a pair of braces { }, around noun phrases in a text. The task is not as trivial as it seems at first glance. Consider the expression

```
[(d) a* n+ -> %{ ... %}]
```

Applied to the input "danvn" (many small cats like milk) this transducer yields three alternative bracketings:

```
xfst> apply down danvn
da{n}v{n}
d{an}v{n}
{dan}v{n}
```

Introduction to XFST – longest match

For certain applications it may be desirable to produce a unique parse, marking the maximal expansion of each NP: "{dan}v{n}". Using the left-to-right, longest-match replace operator `@->` instead of the simple replace operator `->` yields the desired result:

```
[(d) a* n+ @-> %{ ... %}]
```

```
xfst> apply down danvn
{dan}v{n}
```

Introduction to XFST – the coke machine

A vending machine dispenses drinks for 65 cents a can. It accepts any sequence of the following coins: 5 cents (represented as ‘n’), 10 cents (‘d’) or 25 cents (‘q’). Construct a regular expression that compiles into a finite-state automaton that implements the behavior of the soft drink machine, pairing “PLONK” with a legal sequence that amounts to 65 cents.

Introduction to XFST – the coke machine

The construction `A^n` denotes the concatenation of `A` with itself *n* times.
Thus the expression `[n .x. c^5]` expresses the fact that a nickel is worth 5 cents.
A mapping from all possible sequences of the three symbols to the corresponding value:

```
[[n .x. c^5] | [d .x. c^10] | [q .x. c^25]]*
```

The solution:

```
[[n .x. c^5] | [d .x. c^10] | [q .x. c^25]]*
.o.
[c^65 .x. PLONK]
```

Introduction to XFST – the coke machine

```
clear stack
define SixtyFiveCents
[[n .x. c^5] | [d .x. c^10] | [q .x. c^25]]* ;
define BuyCoke
SixtyFiveCents .o. [c^65 .x. PLONK] ;
```

Introduction to XFST – the coke machine

In order to ensure that extra money is paid back, we need to modify the lower language of BuyCoke to make it a subset of `[PLONK* q* d* n*]`.
To ensure that the extra change is paid out only once, we need to make sure that quarters get paid before dimes and dimes before nickels.

```
clear stack
define SixtyFiveCents
[[n .x. c^5] | [d .x. c^10] | [q .x. c^25]]* ;
define ReturnChange SixtyFiveCents .o.
[[c^65 .x. PLONK]* [c^25 .x. q]*
[c^10 .x. d]* [c^5 .x. n]*] ;
```

Introduction to XFST – the coke machine

The next refinement is to ensure that as much money as possible is converted into soft drinks and to remove any ambiguity in how the extra change is to be reimbursed.

```
clear stack
define SixtyFiveCents
[[n .x. c^5] | [d .x. c^10] | [q .x. c^25]]* ;
define ReturnChange SixtyFiveCents .o.
[[c^65 .x. PLONK]* [c^25 .x. q]*
[c^10 .x. d]* [c^5 .x. n]*] ;
define ExactChange ReturnChange .o.
[~$[q q q | [q q | d] d [d | n] | n n]] ;
```

Introduction to XFST – the coke machine

To make the machine completely foolproof, we need one final improvement. Some clients may insert unwanted items into the machine (subway tokens, foreign coins, etc.). These objects should not be accepted; they should passed right back to the client. This goal can be achieved easily by wrapping the entire expression inside an ignore operator.

```
define IgnoreGarbage
[ [ ExactChange ]/[\[q | d | n]]] ;
```

Applications of finite-state technology in NLP

- Phonology; language models for speech recognition
- Representing lexicons and dictionaries
- Morphology; morphological analysis and generation
- Shallow parsing
- Named entity recognition
- Sentence boundary detection; segmentation
- Translation...