Two-Level Morphology

Daniel Zeman

November 6–13, 2020

EUROPEAN UNION
European Structural and Investment Fund
Operational Programme Research,
Development and Education

Charles University
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics

Two-Level (Mor)Phonology

- Kimmo Koskenniemi: PhD thesis (1983)
- Testable using `pc-kimmo` (freely available at `http://software.sil.org/pc-kimmo/`)
- Lauri Karttunen (Xerox Grenoble): two-level compiler, finite-state technology, `xfst`, see `http://www.fsmbook.com/`
- Morphological "classics"

- Larger multi-level toolkits
- XFST (Xerox)
- HFST (Helsinki)
- Foma (Mans Hulden, Colorado)

Kimmo

Finite-State Automaton/Machine (FSA)

- Five-tuple (A, Q, P, q_0, F).
  - A ... finite alphabet of input symbols
  - Q ... finite set of states
  - P ... transition function (set of rules) $A \times Q \to Q$
  - $q_0 \in Q$ ... initial state
  - $F \subseteq Q$ ... set of terminal states

- A word is accepted as correct if we read it as input and we end up in a terminal state.
- An additional action can be bound to the terminal state (output info).

Example of Finite-State Machine

- Checks correct spelling of Czech: *dě, tě, ně...*
- Czech orthographical rules:
  - *di, ti, ni* is pronounced *[ďi, ťi, ňi]*
  - *dě, tě, ně* is pronounced *[ďe, ťe, ňe]*

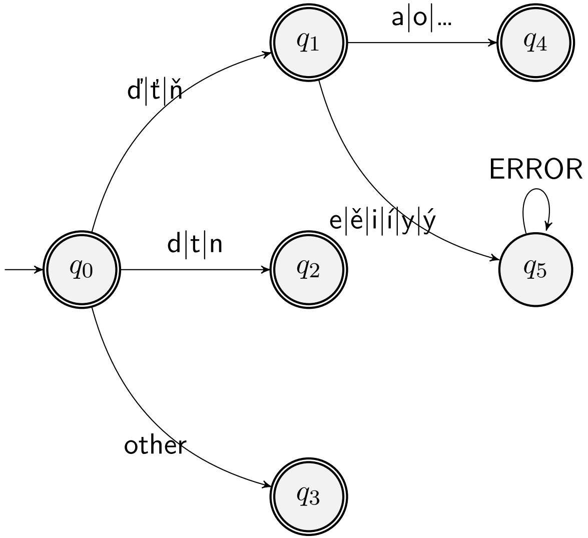
Example of Finite-State Machine

- Checks correct spelling of Czech: *dě, tě, ně...*
- Czech orthographical rules:
  - *di, ti, ni* is pronounced *[ďi, ťi, ňi]*
  - *dě, tě, ně* is pronounced *[ďe, ťe, ňe]*
  - Orthography prohibits strings *ďi, ťi, ňi, ďy, ťy, ňy, ďe, ťe, ňe, ďě, ťě, ňě*

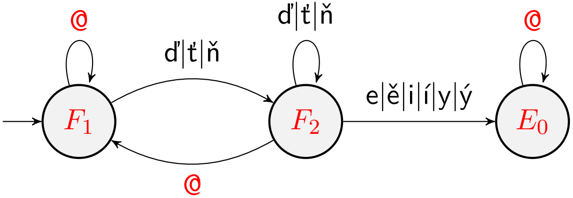
Example of Finite-State Machine

- Checks correct spelling of Czech: *dě, tě, ně…*
- Czech orthographical rules:
  - *di, ti, ni* is pronounced *[ďi, ťi, ňi]*
  - *dě, tě, ně* is pronounced *[ďe, ťe, ňe]*
  - Orthography prohibits strings *ďi, ťi, ňi, ďy, ťy, ňy, ďe, ťe, ňe, ďě, ťě, ňě*
  - Note however that long *ďé, ťé* is permitted: these are the names of the letters *Ď, Ť*. (And *ě* cannot be used for them because it is short.)
- Exception: Czech system of transcription of Mandarin Chinese (used for Chinese names in news and encyclopedias):
  - *ťin* … pinyin equivalent is *jin*

Example of Finite-State Machine



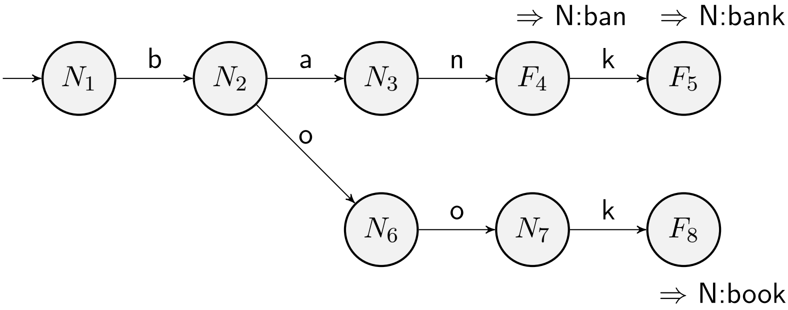
Example of Finite-State Machine (polished, new notation)



- Initial state indexed 1, not 0 (here F_1).
- Index 0 reserved for the error state.
- Terminal states denoted by the letter F.
- The *at* sign ("@") means "other", i.e., characters not found on other transitions from the same state.

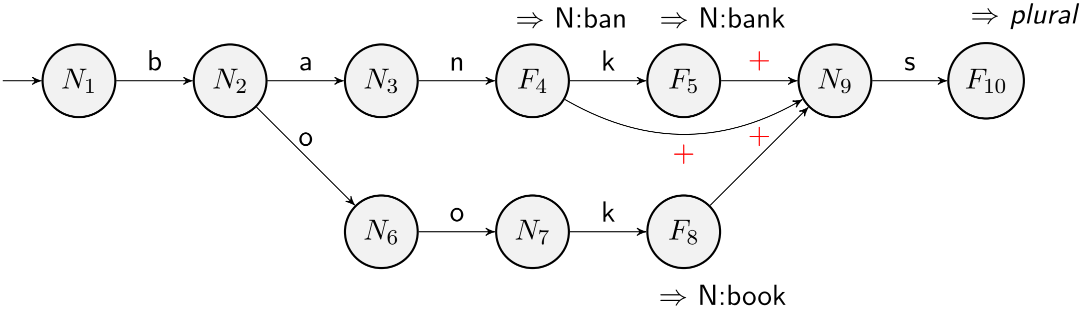
Lexicon

- Implemented as a FSA (trie) *[tri:]*.



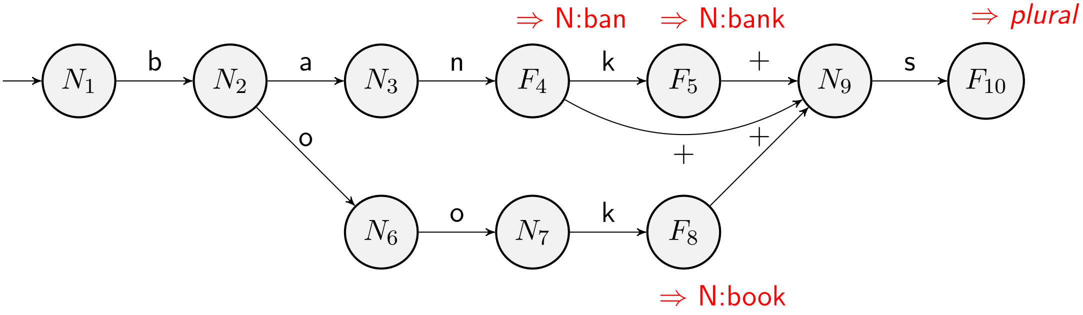
Lexicon

- Implemented as a FSA (trie) *[tri:]*.
- Composed of multiple sublexicons (prefixes, stems, suffixes).
  - **Morphotactics** = what morphemes can occur in what order?



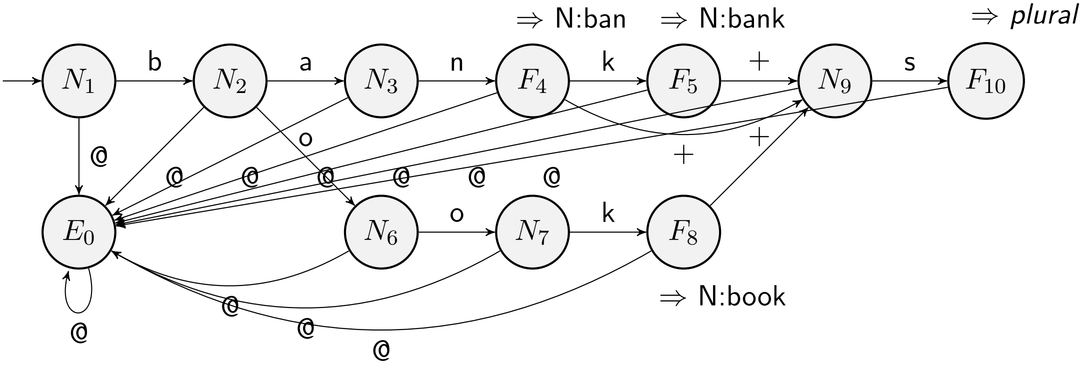
Lexicon

- Implemented as a FSA (trie) *[tri:]*.
- Composed of multiple sublexicons (prefixes, stems, suffixes).
  - **Morphotactics** = what morphemes can occur in what order?
- Notes (glosses) at the end of every sublexicon.
- Compiled from a list of strings and sublexicon references.



Lexicon

- Implemented as a FSA (trie) *[tri:]*.
- Composed of multiple sublexicons (prefixes, stems, suffixes).
  - **Morphotactics** = what morphemes can occur in what order?
- Notes (glosses) at the end of every sublexicon.
- Compiled from a list of strings and sublexicon references.



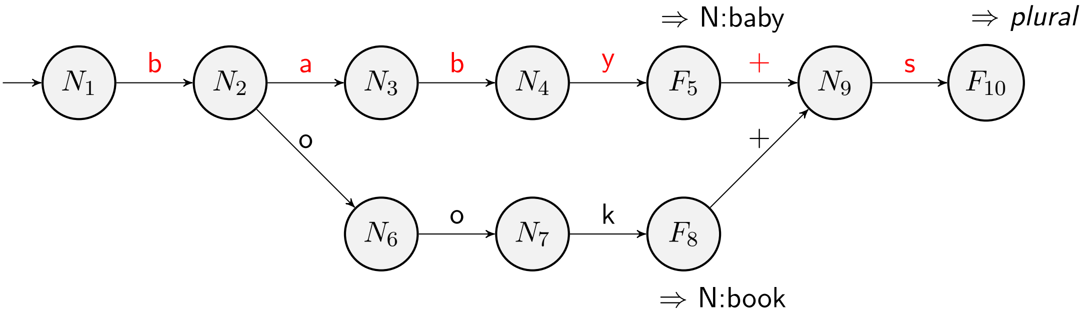
Interlinking Sublexicons

- Unlike trie the lexicon is not a tree but a DAG (directed acyclic graph)
- Each sublexicon entry knows the set of sublexicons we can jump to in the next step ⇒ continuation class or alternation

Sublexicon	Entry	Gloss	Continuation Class
INIT			NounStem AdjStem VerbStem ...
NounStem	muž	N:muž(man)	NMmanSuff
	učitel	N:učitel(teacher)	NMmanSuff
	žen	N:žena(woman)	NFwomSuff
	růž	N:růže(rose)	NFrosSuff
NMmanSuff	+e	Sing:Gen	
	+i	Sing:Dat	
	+e	Sing:Acc	

A Problem Called Phonology

- Sometimes attaching a suffix causes phoneme or grapheme (spelling) changes!
  - For simplicity I will call both *phonology*.
- Plural of *baby* is not **babys* but *babies*!



Two-Level Morphology

- Integration of morphology and phonology is possible and easy.
- Upper (lexical) language
- Lower (surface) language
- Two-level rules:

```
b a b y + 0 s
b a b i 0 e s
```

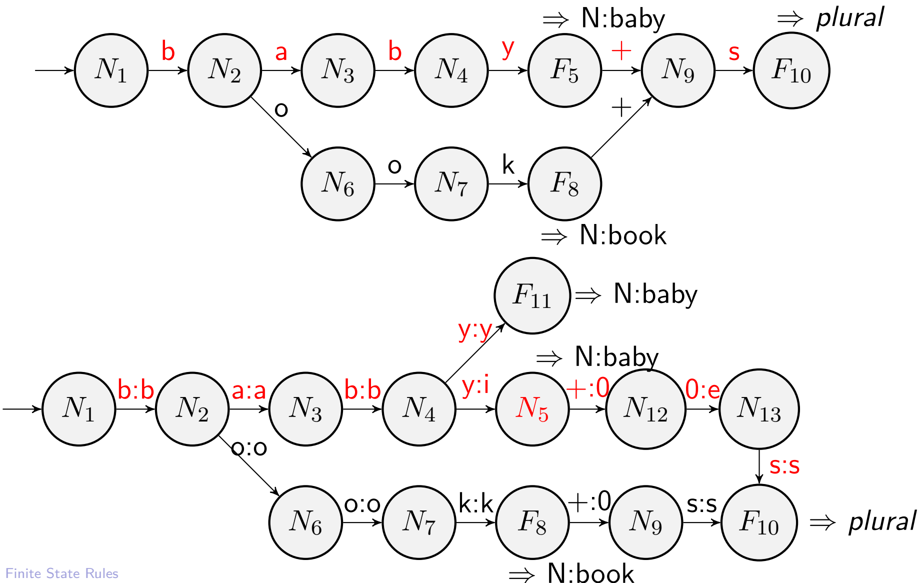
- Alternative notation with colons:

```
b:b a:a b:b y:i +:0 0:e s:s
```

Finite-State Transducer *(převodník)*

- Transducer is a special case of automaton
  - Symbols are pairs (r:s) from finite alphabets R and S.
- Checking (finite-state automaton)
  - Input: sequence of characters
  - Output: yes / no (accept / reject) + state id / gloss
- Analysis (finite-state transducer)
  - Input: sequence $s \in S$ (surface string)
  - Output: sequence $r \in R$ (lexical string) + state id / gloss
    - So how do we obtain it?
- Generation (finite-state transducer)
  - Same as analysis but swapped roles $S \leftrightarrow R$

Automaton vs. Transducer



Another Way of Rule Notation: Two-Level Grammar

- If lexical *y* is followed by +*s*, then on surface the *y* must be replaced by *i* (generation).
- If surface *i* is followed by +*s*, then in lexicon the *i* must be replaced by *y* (analysis).

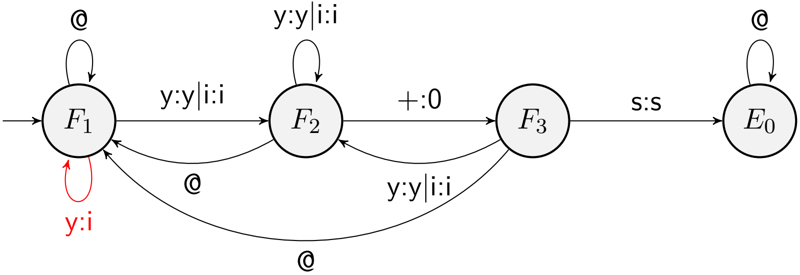
```
y:i <= _ +:0 s:s
```

- We don't require the reverse implication this time. It is possible that *y* corresponds to *i* elsewhere for other reasons.
- In the same context we also require that an *e* is inserted before *s*:

```
0:e <= y:i +:0 _ s:s
```

- Create a transducer (FST) that converts between the surface and lexical layers.
  - More precisely: FST is an automaton that only checks that we are converting the layers correctly.

FST Example: `y:i <= _ +:0 s:s`



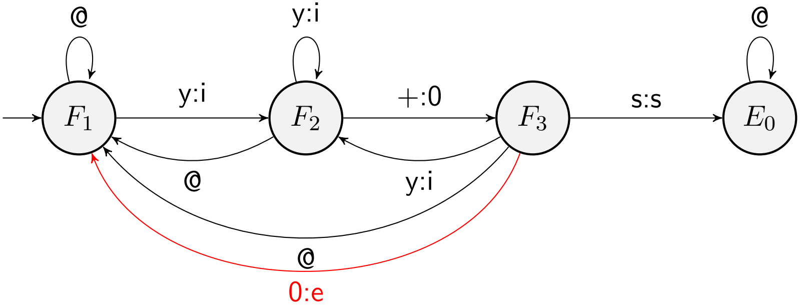
How to Get the FST Input

- FSA simply checked the input.
- With FST we only read half of the input.
- Where do we get the other half?

How to Get the FST Input

- FSA simply checked the input.
- With FST we only read half of the input.
- Where do we get the other half?
- We know it in advance!
  - Typical letter corresponds to itself: i:i, y:y
  - Some letters arise phonologically: y:i
  - We thus know in advance that a surface *i* can correspond either to lexical *i* or *y*.
  - We will check both possibilities. If both are accepted, the analyzed word is ambiguous.

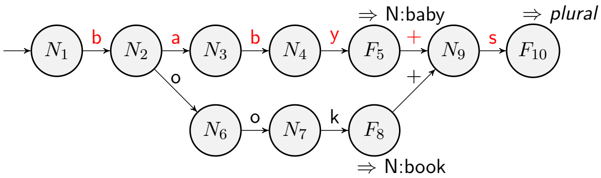
FST Example: `0:e <= y:i +:0 _ s:s`

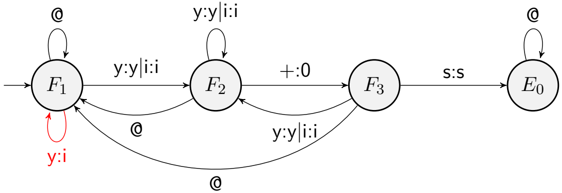


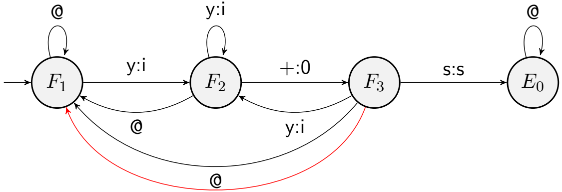
How Does It Work Together

- Parallel FST (including lexicon FSA) can be compiled to one gigantic FST.
- The transducer itself in fact does not convert, it only checks.
- Nevertheless the transducer is a source of information what can be converted to what (i.e. what we can try and have checked by the FST).
  - Besides explicit conversion rules we can also assume for all x the default conversion rule `x:x`.

Lexicon and Rules Together







Two-Level Morphological Analysis

1. Initialize set of paths $P = \{\}$.
2. Read input symbols one-by-one.
3. For each input symbol x generate all lexical symbols y that may correspond to the empty symbol ($\texttt{y:0}$).
4. Extend all paths in P by all corresponding pairs ($\texttt{y:0}$).
5. Check all new extensions against the phonological transducers and the lexical automaton. Remove disallowed paths (partial solutions) from P.
6. Repeat 4–5 until the maximum possible number of subsequent zeros is reached.
7. Generate all possible lexical symbols z for the current input symbol x. Create pairs.
8. Extend each path in P by all such pairs.
9. Check all paths in P (the next transition in FST/FSA). Remove impossible paths.
10. Repeat since step 3 until input finishes.
11. Collect glosses from the lexicon from all paths that survived.

Algorithm Example

- Every letter corresponds to itself
- In addition: `y:i`, `+:0`, `0:e`
- Input: *babies*
- Try inserting lexical + (`+:0`) ... blocked by lexicon (no word starts like that)

Algorithm Example

- Every letter corresponds to itself
- In addition: `y:i`, `+:0`, `0:e`
- Input: *babies*
- Try inserting lexical + (`+:0`) ... blocked by lexicon (no word starts like that)
- Try `b:b` ... OK (neither lexicon nor the transducers object)

Algorithm Example

- Every letter corresponds to itself
- In addition: `y:i`, `+:0`, `0:e`
- Input: *babies*
- Try inserting lexical + (`+:0`) ... blocked by lexicon (no word starts like that)
- Try `b:b` ... OK (neither lexicon nor the transducers object)
- `b:b +:0` ... lexicon error

Algorithm Example

- Every letter corresponds to itself
- In addition: `y:i`, `+:0`, `0:e`
- Input: *babies*
- Try inserting lexical + (`+:0`) … blocked by lexicon (no word starts like that)
- Try `b:b` … OK (neither lexicon nor the transducers object)
- `b:b +:0` … lexicon error
- `b:b a:a` … OK

Algorithm Example

- Every letter corresponds to itself
- In addition: `y:i`, `+:0`, `0:e`
- Input: *babies*
- Try inserting lexical + (`+:0`) ... blocked by lexicon (no word starts like that)
- Try `b:b` ... OK (neither lexicon nor the transducers object)
- `b:b +:0` ... lexicon error
- `b:b a:a` ... OK
- `b:b a:a +:0` ... lexicon error

Algorithm Example

- Every letter corresponds to itself
- In addition: `y:i`, `+:0`, `0:e`
- Input: *babies*
- Try inserting lexical + (`+:0`) ... blocked by lexicon (no word starts like that)
- Try `b:b` ... OK (neither lexicon nor the transducers object)
- `b:b +:0` ... lexicon error
- `b:b a:a` ... OK
- `b:b a:a +:0` ... lexicon error
- `b:b a:a b:b` ... OK

Algorithm Example

- Every letter corresponds to itself
- In addition: `y:i`, `+:0`, `0:e`
- Input: *babies*
- Try inserting lexical + (`+:0`) … blocked by lexicon (no word starts like that)
- Try `b:b` … OK (neither lexicon nor the transducers object)
- `b:b +:0` … lexicon error
- `b:b a:a` … OK
- `b:b a:a +:0` … lexicon error
- `b:b a:a b:b` … OK
- `b:b a:a b:b +:0` … l. error

Algorithm Example

- Every letter corresponds to itself
- In addition: `y:i`, `+:0`, `0:e`
- Input: *babies*
- Try inserting lexical + (`+:0`) ... blocked by lexicon (no word starts like that)
- Try `b:b` ... OK (neither lexicon nor the transducers object)
- `b:b +:0` ... lexicon error
- `b:b a:a` ... OK
- `b:b a:a +:0` ... lexicon error
- `b:b a:a b:b` ... OK
- `b:b a:a b:b +:0` ... l. error
- `b:b a:a b:b i:i` ... l. error

Algorithm Example

- Every letter corresponds to itself
- In addition: `y:i`, `+:0`, `0:e`
- Input: *babies*
- Try inserting lexical + (`+:0`) … blocked by lexicon (no word starts like that)
- Try `b:b` … OK (neither lexicon nor the transducers object)
- `b:b +:0` … lexicon error
- `b:b a:a` … OK
- `b:b a:a +:0` … lexicon error
- `b:b a:a b:b` … OK
- `b:b a:a b:b +:0` … l. error
- `b:b a:a b:b i:i` … l. error
- `b:b a:a b:b y:i` … OK

Algorithm Example

- Every letter corresponds to itself
- In addition: `y:i`, `+:0`, `0:e`
- Input: *babies*
- Try inserting lexical + (`+:0`) ... blocked by lexicon (no word starts like that)
- Try `b:b` ... OK (neither lexicon nor the transducers object)
- `b:b +:0` ... lexicon error
- `b:b a:a` ... OK
- `b:b a:a +:0` ... lexicon error
- `b:b a:a b:b` ... OK
- `b:b a:a b:b +:0` ... l. error
- `b:b a:a b:b i:i` ... l. error
- `b:b a:a b:b y:i` ... OK [p_1]
- ... `b:b y:i +:0` ... OK [$p_1 \to p_2$]

Algorithm Example

- Every letter corresponds to itself
- In addition: `y:i`, `+:0`, `0:e`
- Input: *babies*
- Try inserting lexical + (`+:0`) ... blocked by lexicon (no word starts like that)
- Try `b:b` ... OK (neither lexicon nor the transducers object)
- `b:b +:0` ... lexicon error
- `b:b a:a` ... OK
- `b:b a:a +:0` ... lexicon error
- `b:b a:a b:b` ... OK
- `b:b a:a b:b +:0` ... l. error
- `b:b a:a b:b i:i` ... l. error
- `b:b a:a b:b y:i` ... OK [p_1]
- ... `b:b y:i +:0` ... OK [$p_1 \rightarrow p_2$]
- ... `b:b y:i +:0 +:0` ... error [$p_2 \rightarrow ?$]

Algorithm Example

- Every letter corresponds to itself
- In addition: `y:i`, `+:0`, `0:e`
- Input: *babies*
- Try inserting lexical + (`+:0`) … blocked by lexicon (no word starts like that)
- Try `b:b` … OK (neither lexicon nor the transducers object)
- `b:b +:0` … lexicon error
- `b:b a:a` … OK
- `b:b a:a +:0` … lexicon error
- `b:b a:a b:b` … OK
- `b:b a:a b:b +:0` … l. error
- `b:b a:a b:b i:i` … l. error
- `b:b a:a b:b y:i` … OK [p_1]
- … `b:b y:i +:0` … OK [$p_1 \to p_2$]
- … `b:b y:i +:0 +:0` … error
- … `y:i e:e` … error [$p_1 \to ?$]
- … `y:i 0:e` … OK [$p_1 \to p_3$]
- … `y:i +:0 e:e` … error [$p_2 \to ?$]
- … `y:i +:0 0:e` … OK [$p_2 \to p_4$]

Algorithm Example

- Every letter corresponds to itself
- In addition: `y:i`, `+:0`, `0:e`
- Input: *babies*
- Try inserting lexical + (`+:0`) … blocked by lexicon (no word starts like that)
- Try `b:b` … OK (neither lexicon nor the transducers object)
- `b:b +:0` … lexicon error
- `b:b a:a` … OK
- `b:b a:a +:0` … lexicon error
- `b:b a:a b:b` … OK
- `b:b a:a b:b +:0` … l. error
- `b:b a:a b:b i:i` … l. error
- `b:b a:a b:b y:i` … OK
- … `b:b y:i +:0` … OK
- … `b:b y:i +:0 +:0` … error
- … `y:i e:e` … error
- … `y:i 0:e` … OK [$p_1 \rightarrow p_3$]
- … `y:i +:0 e:e` … error
- … `y:i +:0 0:e` … OK [$p_2 \rightarrow p_4$]
- … `y:i 0:e +:0` … OK [$p_3 \rightarrow p_5$]
- … `y:i 0:e +:0 +:0` … error [$p_5 \rightarrow ?$]
- … `+:0 0:e +:0` … error [$p_4 \rightarrow ?$]

Algorithm Example

- Every letter corresponds to itself
- In addition: `y:i`, `+:0`, `0:e`
- Input: *babies*
- Try inserting lexical + (`+:0`) … blocked by lexicon (no word starts like that)
- Try `b:b` … OK (neither lexicon nor the transducers object)
- `b:b +:0` … lexicon error
- `b:b a:a` … OK
- `b:b a:a +:0` … lexicon error
- `b:b a:a b:b` … OK
- `b:b a:a b:b +:0` … l. error
- `b:b a:a b:b i:i` … l. error
- `b:b a:a b:b y:i` … OK
- … `b:b y:i +:0` … OK
- … `b:b y:i +:0 +:0` … error
- … `y:i e:e` … error
- … `y:i 0:e` … OK [$p_1 \to p_3$]
- … `y:i +:0 e:e` … error
- … `y:i +:0 0:e` … OK [$p_2 \to p_4$]
- … `y:i 0:e +:0` … OK [$p_3 \to p_5$]
- … `y:i 0:e +:0 +:0` … error
- … `+:0 0:e +:0` … error
- … `y:i 0:e s:s` … error [$p_3 \to ?$]
- … `+:0 0:e s:s` … OK [$p_4 \to p_6$]
- … `0:e +:0 s:s` … OK [$p_5 \to p_7$]

Algorithm Example

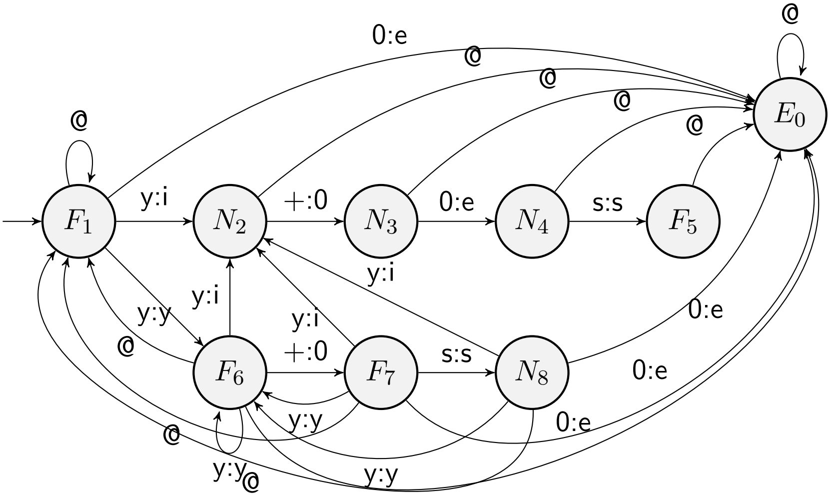
- Every letter corresponds to itself
- In addition: `y:i`, `+:0`, `0:e`
- Input: *babies*
- Try inserting lexical + (`+:0`) … blocked by lexicon (no word starts like that)
- Try `b:b` … OK (neither lexicon nor the transducers object)
- `b:b +:0` … lexicon error
- `b:b a:a` … OK
- `b:b a:a +:0` … lexicon error
- `b:b a:a b:b` … OK
- `b:b a:a b:b +:0` … l. error
- `b:b a:a b:b i:i` … l. error
- `b:b a:a b:b y:i` … OK
- … `b:b y:i +:0` … OK
- … `b:b y:i +:0 +:0` … error
- … `y:i e:e` … error
- … `y:i 0:e` … OK
- … `y:i +:0 e:e` … error
- … `y:i +:0 0:e` … OK
- … `y:i 0:e +:0` … OK
- … `y:i 0:e +:0 +:0` … error
- … `+:0 0:e +:0` … error
- … `y:i 0:e s:s` … error
- … `+:0 0:e s:s` … OK [$p_4 \rightarrow p_6$]
- … `0:e +:0 s:s` … OK [$p_5 \rightarrow p_7$]
- … `+:0 0:e s:s +:0` … error
- … `0:e +:0 s:s +:0` … error

Algorithm Example

- Every letter corresponds to itself
- In addition: `y:i`, `+:0`, `0:e`
- Input: *babies*
- Try inserting lexical + (`+:0`) … blocked by lexicon (no word starts like that)
- Try `b:b` … OK (neither lexicon nor the transducers object)
- `b:b +:0` … lexicon error
- `b:b a:a` … OK
- `b:b a:a +:0` … lexicon error
- `b:b a:a b:b` … OK
- `b:b a:a b:b +:0` … l. error
- `b:b a:a b:b i:i` … l. error
- `b:b a:a b:b y:i` … OK
- … `b:b y:i +:0` … OK
- … `b:b y:i +:0 +:0` … error
- … `y:i e:e` … error
- … `y:i 0:e` … O
- … `y:i +:0 e`
- … `y:i +:0`
- … `y:i`
- … `y:i`
- … `+:0`
- … `y:i`
- … `+:0 0:e`
- … `0:e +:0 s`

One of the hypotheses
could be blocked
by our FSTs
if we designed them
better (⇔)

Fixed and Merged FST



Czech Examples

- *skrýš* “hideaway” — genitive *skrýš+e* → *skrýše*
- *káď* “tun” — genitive *káď+e* → *kádě*

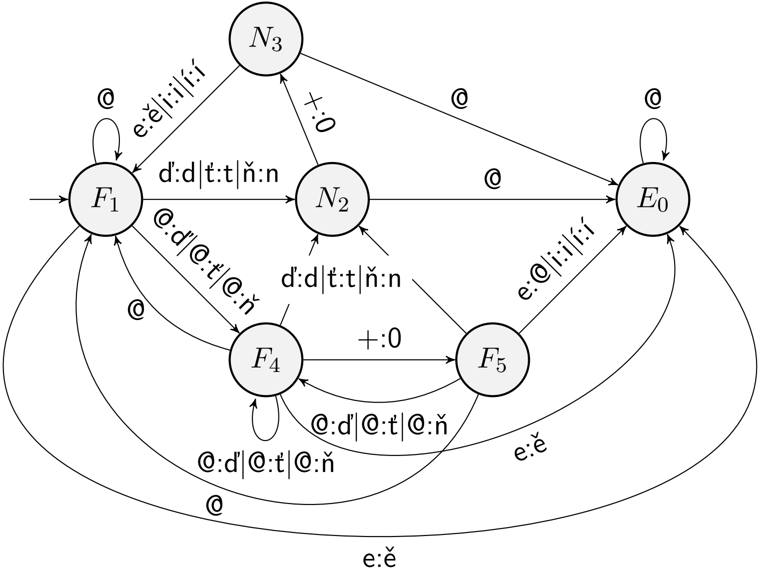
- *ď* and *e* normally cannot occur together...
- ... unless they come from separate morphemes (stem + suffix)!
- We need a rule that will ensure the correct conversion ďe → dě.

```
k á ď + e
k á d 0 ě
```

Example of Transducer: ď, ť, ň on morpheme boundary

- `ď:d +:0 e:ě` is correct, other possibilities are not.
- Assumption: *ďe, ďi* could only occur on morpheme boundary (otherwise it is in the lexicon ⇒ it should be correct).
- We don't fully cover *ďě*. If the character *ě* occurs in a suffix, it must be because of phonology:
  - *brzda brzďe (brzdě), žena žeňe (ženě), máta máťe (mátě), máma mámňe (mámě), bába bábje (bábě), lípa lípje (lípě), chůva chůvje (chůvě), matka matce, váha váze, sprcha sprše, kůra kůře, mula mule, vosa vose, lůza lůze*
- We don't cover *ďy* here (which could arise when inflecting a noun ending in -*ďa*; it is incorrect and should be changed to -*di*).

Example of Transducer: ď, ť, ň on morpheme boundary



Example of Transducer: ď, ť, ň on morpheme boundary

```
RULE "[ď:d | ň:n | ť:t] <=> _ +:0 [e:ě | i:i | í:í]" 5 12
```

	ď	ň	ť	@	@	@	+	e	i	í	e	@
	d	n	t	ď	ň	ť	0	ě	i	í	@	@
1:	2	2	2	4	4	4	1	0	1	1	1	1
2.	0	0	0	0	0	0	3	0	0	0	0	0
3.	0	0	0	0	0	0	0	1	1	1	0	0
4:	2	2	2	4	4	4	5	0	1	1	1	1
5:	2	2	2	4	4	4	1	0	0	0	0	1

Czech Feminine Noun Consonant Changes

The pairs illustrate various stem-final changes in the paradigm *žena* of Czech feminine nouns. All words are **surface** strings—nominative singular on the left, dative singular on the right.

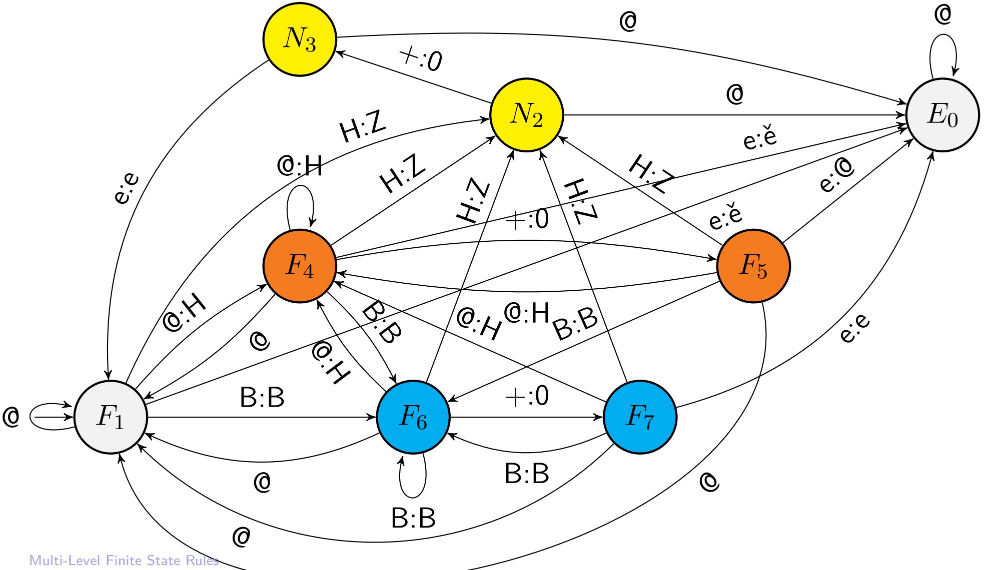
- *váha* – *váze* "weight"
- *sprcha* – *sprše* "shower"
- *matka* – *matce* "mother"
- *kůra* – *kůře* "bark"
- *Olga* – *Olze* "Olga"

- *vláda* – *vládě* "government"
- *máta* – *mátě* "mint"
- *žena* – *ženě* "woman"

- *bába* – *bábě* "old woman"
- *karafa* – *karafě* "carafe"
- *máma* – *mámě* "mom"
- *chrpa* – *chrpě* "cornflower"
- *jíva* – *jívě* "goat willow"

- *Naďa* – *Nadě* "Naďa"
- *Jíťa* – *Jítě* "Jíťa"
- *Áňa* – *Áně* "Áňa"

Czech Feminine Noun Consonant Changes



H:Z = g:z | h:z | ch:š | k:c | r:ř

B:B = b:b | f:f | m:m | p:p | v:v | w:w | q:q | d:d | t:t | n:n | ď:d | ť:t | ň:n

Czech Feminine Noun: Insert *e* in Consonant Clusters

Nom Sing	**Gen Plur**	**Translation**
žena	*žen*	"woman"
pomsta	*pomst*	"revenge"
sprcha	*sprch*	"shower"
matka	*matek*	"mother"
částka	*částek*	"amount"
nozdra	*nozder*	"nostril"
skvrnka	*skvrnek*	"stain"

```
m a t E K        m a t E K + e
m a t e k        m a t 0 c 0 e
```

Some Issues

- Long-distance dependencies
- Flipping analysis and generation
- Transducers are low-level devices

Long-Distance Dependencies

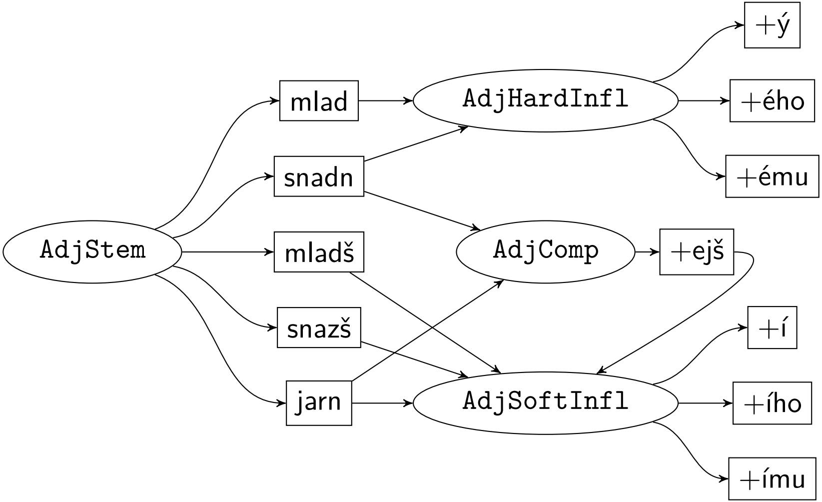
Disadvantage of finite-state morphology:

- Capturing of long-distance dependencies is clumsy!

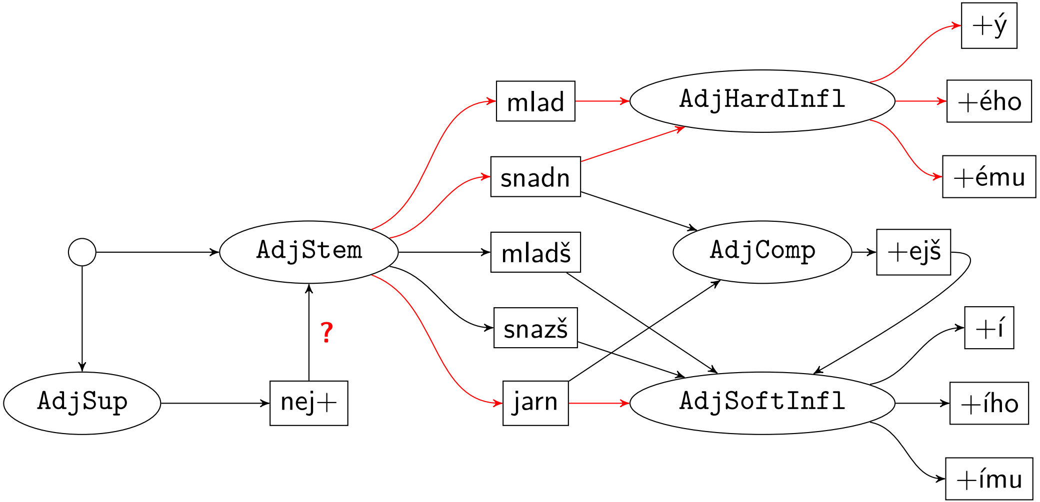
Long-Distance Dependencies: Czech Adjectives

- Two inflection classes:
  - Hard: *černý* "black", *černého*, *černému*, ..., *černá* [Fem], *černé*...
  - Soft: *jarní* "spring", *jarního*, *jarnímu*, ..., *jarní* [Fem], *jarní*...
- Regular comparative:
  - Suffix *-ejš*
  - Comparative is always soft regardless the original class:
    černější, *černějšího*, *černějšímu*, ..., *jarnější*, *jarnějšího*, *jarnějšímu*...
- Irregular comparatives:
  - *mladý* "young" ⇒ *mladší*
  - *snadný* "easy" ⇒ *snadnější* | *snazší*
- Superlative = *nej-* + comparative:
  - *nejmladší* "youngest"
  - **We must remember the prefix until, indefinitely later, we see the suffix!**

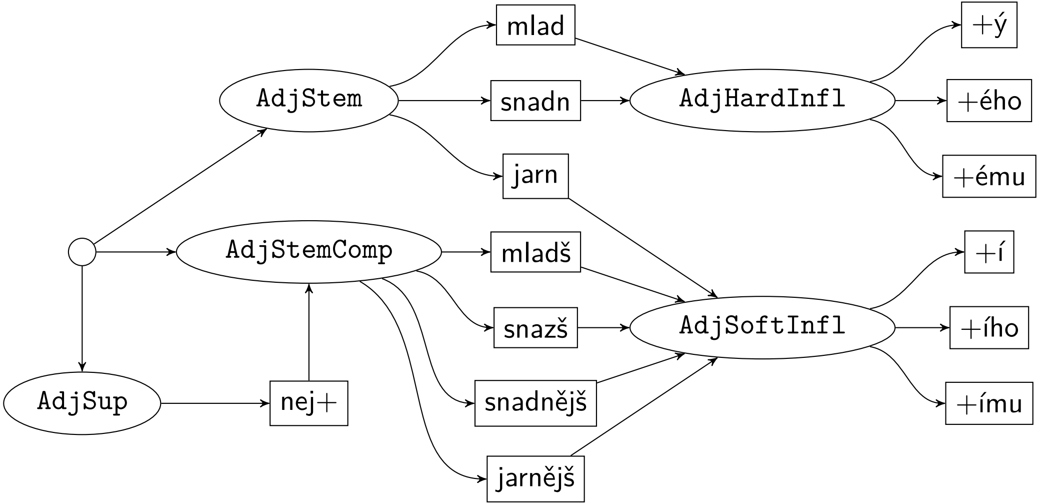
Czech Adjectives without Superlative



Czech Adjectives including Superlative



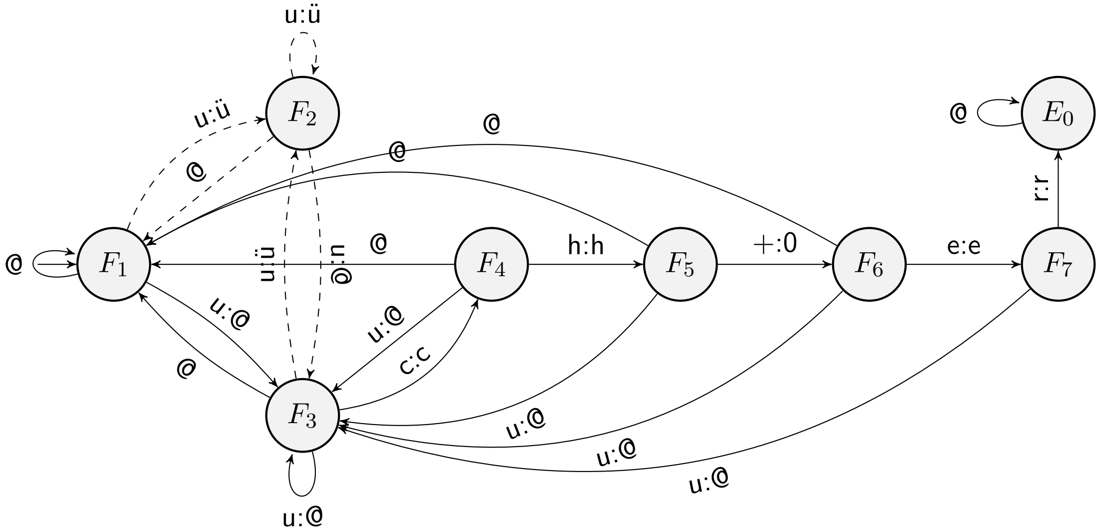
Czech Adjectives including Superlative



Umlauts in German Plurals

- German umlauts (simplified):
  - *u* $\leftrightarrow$ *ü* if (not only if) followed by *cher* (*Buch* $\rightarrow$ *Bücher* "book $\rightarrow$ books")
  - rule: `u:ü` $\Leftarrow$ `_ c:c h:h +:0 e:e r:r`

Umlauts in German Plurals



Umlauts in German Plurals

- *Buch* / *Bücher* "book / books", *Dach* / *Dächer* "roof / roofs", *Loch* / *Löcher* "hole / holes"
- Context should contain `+:0` and perhaps test end of word (#)
  - Otherwise *Sucherei* "searching" will be considered wrong!
  - Not only we must recognize that there is a suffix. It must be a plural suffix and the stem must be marked for plural umlauting.
  - Counterexamples:
    - *Kocher* "cooker", here the *-er* suffix only derives from the verb *kochen* "to cook". Kocher is identical in singular and plural! We don't want to confuse it with *Köcher* "quiver", nor to consider umlautless *Kocher* an error!
    - *Besucher* "visitor", derived from *Besuch* "visit", same singular and plural, there is no **Besücher*!
- Capturing long-distance dependencies is clumsy
  - *Kraut* / *Kräuter* "herb / herbs" has different intervening symbols ⇒ different rule
  - A transducer could be more general and anything until $+$*er* but would it overgenerate?

Long-Distance Effects

- Czech superlatives
- German umlauts etc.: Harald Trost (1990): *The application of two-level morphology to nonconcatenative German morphology.* In COLING-90, Helsinki. (Also Richard Sproat's book p. 170, note 31.)
- Finnish, Turkish etc.: vowel harmony (solved in Koskenniemi's thesis)
- Sanskrit consonant harmony (*uSnatarānām*, example in Sproat's book p. 134)

Two-Levelness and the Lexicon

- Lexicon contains only lexical (upper) symbols
  - Their relation to surface is expressed solely by the transducers
- On the other hand there are *glosses* (output of analysis)
- In fact the system contains 3 levels!

Two-Levelness and the Lexicon

- Lexicon contains only lexical (upper) symbols
  - Their relation to surface is expressed solely by the transducers
- On the other hand there are *glosses* (output of analysis)
- In fact the system contains 3 levels!
  - Surface level (SL):
    - *book*

Two-Levelness and the Lexicon

- Lexicon contains only lexical (upper) symbols
  - Their relation to surface is expressed solely by the transducers
- On the other hand there are *glosses* (output of analysis)
- In fact the system contains 3 levels!
  - Surface level (SL):
    - *book*
  - Lexical level (LL, word segmented to morphemes):
    - *book+s*

Two-Levelness and the Lexicon

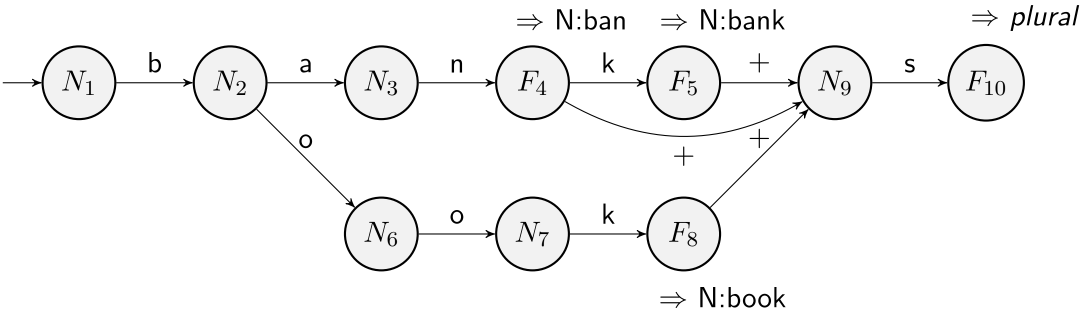
- Lexicon contains only lexical (upper) symbols
  - Their relation to surface is expressed solely by the transducers
- On the other hand there are *glosses* (output of analysis)
- In fact the system contains 3 levels!
  - Surface level (SL):
    - *book*
  - Lexical level (LL, word segmented to morphemes):
    - *book+s*
  - Glosses (lemma, part of speech, tag, anything):
    - *N(book)+plural*

Analysis and Generation

- Analysis is the transition from the surface to the lexical level
  - *books* $\rightarrow$ *book+s*   *book + plural*
- Generation (synthesis) is the transition from the lexical to the surface level
  - Typical input would be glosses rather than morphemes
  - *book + plural* $\rightarrow$ *book+s* $\rightarrow$ *books*

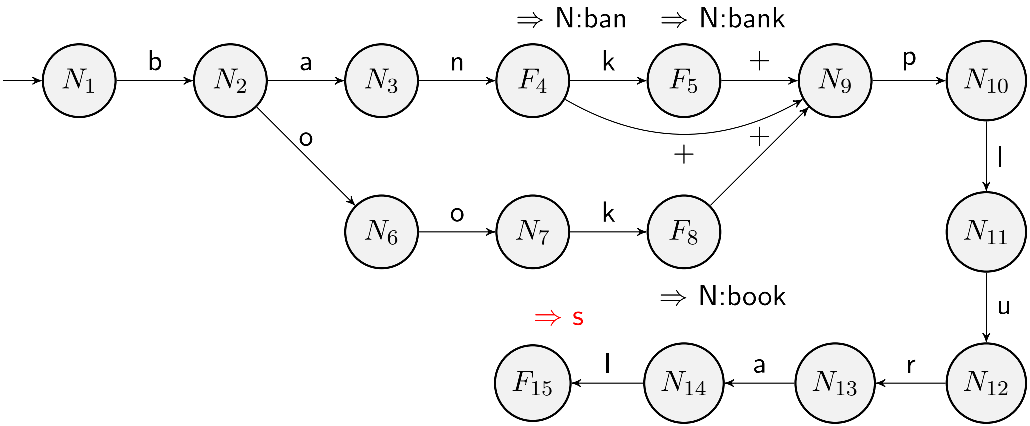
Lexicon for Analysis

- Implemented as a FSA (trie)
- Composed of multiple sublexicons (prefixes, stems, suffixes).
- Notes (glosses) at the end of every sublexicon.
- Compiled from a list of strings and sublexicon references.



Lexicon for Generation

- Swap surface and lexical level (glosses)
- Again, it can be automatically compiled from the same list as the lexicon for analysis
- The rest works the same way



Two-Level Grammar

- Extension of Kimmo (Lauri Karttunen, Xerox)
- Formalism for describing rules for which we need a FST
- Three parts:
  - Pair upper-lower symbol = change
  - Context of the change
  - Relation between change and context (operator)
- Example: in the given right-hand context, we *must* change *ď* to *d*
- Notation:

$$\texttt{ď:d} \Leftarrow \_\ \texttt{+:0 e:@}$$

- (Unless there are other rules, by this we have permitted `ď:d` in other contexts as well.)

Two-Level Grammar

- `x:y` $\Leftarrow$ `lc _ rc`
  If *x* occurs between the left context *lc* and the right context *rc*, then it must surface as *y*. In this context *x* *always* surfaces as *y*

Two-Level Grammar

- `x:y` $\Leftarrow$ `lc _ rc`
  If *x* occurs between the left context *lc* and the right context *rc*, then it must surface as *y*. In this context *x* *always* surfaces as *y*
- `x:y` $\Rightarrow$ `lc _ rc`
  x surfaces as *y* *only* in this context

Two-Level Grammar

- `x:y` $\Leftarrow$ `lc _ rc`
  If *x* occurs between the left context *lc* and the right context *rc*, then it must surface as *y*. In this context *x* *always* surfaces as *y*
- `x:y` $\Rightarrow$ `lc _ rc`
  x surfaces as *y* *only* in this context
- `x:y` $\Leftrightarrow$ `lc _ rc`
  If and only if *x* is found in this context, it surfaces as *y*

Two-Level Grammar

- `x:y` $\Leftarrow$ `lc _ rc`
  If *x* occurs between the left context *lc* and the right context *rc*, then it must surface as *y*. In this context *x* *always* surfaces as *y*
- `x:y` $\Rightarrow$ `lc _ rc`
  x surfaces as *y* *only* in this context
- `x:y` $\Leftrightarrow$ `lc _ rc`
  If and only if *x* is found in this context, it surfaces as *y*
- `x:y` $/\Leftarrow$ `lc _ rc`
  x *never* surfaces as *y* in this context

Discussion of Kimmo and Related Approaches

- Traditional generative phonology (Chomsky's followers): successive application of ordered rules (cf. Foma in this lecture)
- Kay and Kaplan proposed cascaded FSTs. Upper and lower tape, several intermediate tapes in between. Analysis problem: non-determinism can cause the number of intermediate tapes to grow exponentially (Sproat p. 139)
- KIMMO complexity: Barton et al. (1987): Kimmo *generation* is NP-hard. Koskenniemi and Church (1988): Natural languages don't contain phenomena that would exploit the potential for exponentiality (Sproat p. 171)

Multi-Level Finite State Rules

XFST

- Xerox Finite State Toolkit
  - xfst, lexc, tokenize, lookup
  - Binaries and API for multiple operating systems
  - Kenneth R. Beesley, Lauri Karttunen: *Finite State Morphology*. CSLI Publications, 2003
- `http://www.fsmbook.com/`
  - `http://www.stanford.edu/~laurik/.book2software/`
  - `http://cs.haifa.ac.il/~shuly/teaching/06/nlp/xfst-tutorial.pdf`
  - `http://cs.haifa.ac.il/~shuly/teaching/06/nlp/fst2.pdf`
- Current version uses UTF-8 by default
- Some support for reduplication (!)
  - At compile time, morpheme m can be replaced by regex m^2
  - It simulates having two entries in the lexicon: one for the normal form and one for the reduplicated one

HFST (Helsinki)

- Helsinki Finite-State Transducer Technology
  - `http://www.ling.helsinki.fi/kieliteknologia/tutkimus/hfst/`
  - Licensed under GNU LGPL 3.0
  - Finnish lexicon and rules available

Foma

- Open-source finite-state toolkit
  - In contrast, xfst comes without sources and with some copyright restrictions
- Claims compatibility with Xerox tools
  - But also supports Perl-style regular expressions
- Now integrated in Apertium (open-source rule-based machine translation framework)
- Home: `https://code.google.com/p/foma/`
  - Download: `https://bitbucket.org/mhulden/foma/downloads/`
  - Publication: `https://www.aclweb.org/anthology/E09-2008/`

Foma vs. Kimmo

- Multiple levels
  - Sequence of ordered rewrite rules
  - Even lexicon supports two levels (`TAG:suffix`)
- Regular expressions
  - Instead of directly encoding transducers
  - Supports usual FSM algorithms (minimization etc.)
- Sequence of rules still compiled into one FST
  - We still have one upper and one lower language

Compiling Regular Expressions: `regex`

```
regex a+;
regex c a t | d o g;
regex ?* a ?*;
```

```
regex [a:b | b:a]*;
regex [c a t]:[k a t u a];
regex b -> p, g -> k, d -> t || _ .#.;
```

Foma Operators

- (space) ... concatenation
- | ... union
- * ... Kleene star
- & ... intersection
- ~ ... complement
- Single- and multi-character symbols
  - Supports Unicode
- 0 ... empty string (epsilon)
- ? ... any symbol (similar to "." in Perl, grep etc.)
- (a) ... "a" is optional (as "a?" in Perl)

Testing Automata against Words

```
foma[0]: regex ?* a ?*;
261 bytes. 2 states, 4 arcs, Cyclic.
foma[1]:
```

Testing Automata against Words

```
foma[0]: regex ?* a ?*;
261 bytes. 2 states, 4 arcs, Cyclic.
foma[1]: down
apply down>
```

Testing Automata against Words

```
foma[0]: regex ?* a ?*;
261 bytes. 2 states, 4 arcs, Cyclic.
foma[1]: down
apply down> ab
ab
apply down>
```

Testing Automata against Words

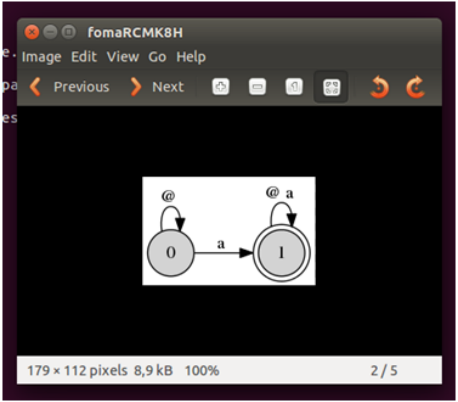
```
foma[0]: regex ?* a ?*;
261 bytes. 2 states, 4 arcs, Cyclic.
foma[1]: down
apply down> ab
ab
apply down> bbx
???
apply down>
```

Testing Automata against Words

```
foma[0]: regex ?* a ?*;
261 bytes. 2 states, 4 arcs, Cyclic.
foma[1]: down
apply down> ab
ab
apply down> bbx
???
apply down> CTRL+D
foma[1]:
```

Graphical Visualization in Linux

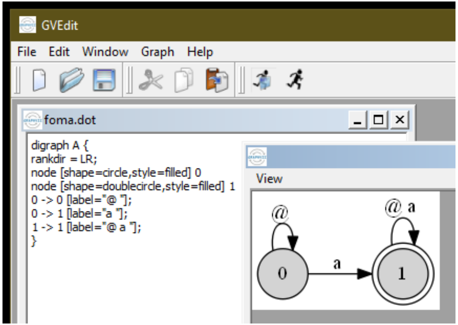
```
foma[0]: regex ?* a ?*;
261 bytes. 2 states, 4 arcs, Cyclic.
foma[1]: view net
foma[1]:
```



Graphviz must be installed.

Graphical Visualization in Windows with Graphviz Installed

```
foma[0]: regex ?* a ?*;
261 bytes. 2 states, 4 arcs, Cyclic.
foma[1]: print dot > foma.dot
foma[1]:
```



Load foma.dot in GVEdit.

Labeling FSMs: `define`

```
foma[0]: define V [a|e|i|o|u];
defined V: 317 bytes. 2 states, 5 arcs, 5 paths.
foma[0]: define StartsWithVowel [V ?*];
defined StartsWithVowel: 429 bytes. 2 states, 11 arcs, Cyclic.
foma[0]:
```

Difference between Colon “:” and Arrow “->”

```
regex [a:b | b:a]*;
regex [c a t]:[k a t u a];
regex b -> p, g -> k, d -> t || _ .#.;
```

- Colon “:” affects a specific position or a sequence of positions
- Regular expressions with colons restrict the set of words that belong to the language
- Regular expressions with arrows yield transducers that accept any string. If the string contains the searched character, it will be replaced
- Arrow is implemented with the help of colon

Rewrite Rules

```
foma[0]: regex a -> b;
290 bytes. 1 states, 3 arcs, Cyclic.
foma[1]: down
apply down> a
b
apply down> axa
bxb
apply down> CTRL+D
```

The FST ("net") accepts any input. It changes *a* to *b*.

Conditional Replacement

```
foma[0]: regex a -> b || c _ d ;
526 bytes. 4 states, 16 arcs, Cyclic.
foma[1]: down cadca
cbdca
foma[1]:
```

Multiple Contexts

```
foma[0]: regex a -> b || c _ d, e _ f;
890 bytes. 7 states, 37 arcs, Cyclic.
foma[1]: down
apply down> cadeaf
cbdebf
apply down> a
a
apply down> CTRL+D
```

Parallel Rules
End of Word Symbol

```
foma[0]: regex b -> p, g -> k, d -> t || _ .#. ;
634 bytes. 3 states, 20 arcs, Cyclic.
foma[1]: down
apply down> cab
cap
apply down> dog
dok
apply down> dad
dot
apply down> CTRL+D
```

Composition of Rules

```
foma[0]: define Rule1 a -> b || c _ ;
defined Rule1: 384 bytes. 2 states, 8 arcs, Cyclic.
foma[0]: define Rule2 b -> c || _ d ;
defined Rule2: 416 bytes. 3 states, 10 arcs, Cyclic.
foma[0]: regex Rule1 .o. Rule2;
574 bytes. 4 states, 19 arcs, Cyclic.
foma[1]: down
apply down> cad
ccd
apply down> ca
cb
apply down> ad
ad
apply down> CTRL+D
```

Review

- `regex` regular-expression;
  - compile regular expression and put it on the stack
- `define` name regular-expression;
  - compile regular expression, give it a name and do not put it on the stack
- `view` (`view net`)
  - (Linux only) display the compiled FST from stack graphically in a window (Graphviz tool)
- `print dot > file.dot`
  - save the compiled FST to `file.dot` in a format that can be read by Graphviz
- `net` (`print net`)
  - textual description of the compiled FST
- `down` <word> (`apply down`)
  - run a lexical word through a FST (generation)
- `up` <word> (`apply up`)
  - run a surface word through a FST (analysis)
- `words` (`print words`)
  - print all the words the FST accepts
- `lower-words`
  - only lower side of the FST
- `upper-words`
  - only upper side of the FST

Lexicon in `lexc` Format

Create the file, then load it to Foma.

```
LEXICON Root
cat   Suff;
dog   Suff;
horse Suff;

LEXICON Suff
s #;
  #;
```

Load Lexicon to Foma

```
foma[0]: read lexc en1.lexc
Root...4, Suff...2
Building lexicon...Determinizing...Minimizing...Done!
755 bytes. 13 states, 15 arcs, 8 paths.
foma[1]: print words
horse horses dog dogs cat cats
foma[1]: define Lexicon;
foma[0]:
```

Or alternatively:

```
foma[0]: define Lexicon [c a t|d o g|...] (s);
```

Example English lexc File

```
Multichar_Symbols
+N +V +PastPart
+Past +PresPart +3P
+Sg +Pl
LEXICON Root
Noun ;
Verb ;
LEXICON Noun
cat  Ninfl;
city Ninfl;
LEXICON Ninfl
+N+Sg:0  #;
+N+Pl:^s #;
```

^ is our morpheme boundary

Put It All Together

- Lexical string = `city+N+Pl`
- Lexicon transducer: `city+N+Pl` → `city^s`
- *y* → *ie* rule: `city^s` → `citie^s`
- Remove ^: `citie^s` → `cities`
- Surface string = `cities`

Put It All Together

```
foma[0]: read lexc en2.lexc
foma[1]: define Lexicon;
foma[0]: define YRepl y -> i e || _ "^" s;
foma[0]: define Cleanup "^" -> 0;
foma[0]: regex Lexicon .o. YRepl .o. Cleanup;
foma[1]: lower-words
cat cats city cities...
```

Irregular Forms

```
LEXICON Verb
beg  Vinfl;
make+V+PastPart:made #; ! bypass Vinfl
make+V               #;
...
```

Priority Union

```
foma[1]: define Grammar;
foma[0]: define Exceptions [m a k e "+V" "+PastPart"]:[m a d e];
foma[0]: regex [Exceptions .P. Grammar];
foma[1]: down
apply down> make+V+PastPart
made
apply down> CTRL+D
```

Alternate Forms

English: *cactus*+N+Pl → *cactuses*, *cacti*

```
foma[0]: define Parallel [c a c t u s "+N" "+Pl"]:[c a c t i];
foma[0]: regex Parallel | Grammar;
...
```

Long-Distance Dependencies

- Constrain co-occurrence of morphemes
- Create a filter before or after lexical level
- Usual format `~$[PATTERN];`
- “The language does not contain PATTERN.”

```
define SUPFILT ~$[ "[Sup]" ?+ "[Pos]" ];
define MORPH SUPFILT .o. LEX .o. RULES;
```

Flag Diacritics

- Invisible symbols to control co-occurrence:
  - U ... unify features @U.*feature.value*@
  - P ... positive set @P.*feature.value*@
  - N ... negate @N.*feature.value*@
  - R ... require feature/value @R.*feature(.value)*@
  - D ... disallow feature/value @D.*feature(.value)*@
  - C ... clear feature @C.*feature*@
  - E ... require equal feature/value @E.*feature.value*@

Recall: Degrees of Czech Adjectives

- positive: *chytr* + *ý* “smart”
- comparative: *chytř* + *ejš* + *í* “smarter”
- superlative: *nej* + *chytř* + *ejš* + *í* “smartest”
- ungrammatical: * *nej* + *chytr* + *ý*

Flag Diacritics to Control Czech Superlatives

- `Multichar_Symbols Sup+ +Pos +Comp @P.SUP.ON@ @D.SUP@`
  - Declare the diacritics we are going to use
- `LEXICON AdjSup @P.SUP.ON@Sup+:@P.SUP.ON@nej^ Adj;`
  - Remember SUP was seen
- `LEXICON AhardDeg @D.SUP@+Pos:@D.SUP@ Ahard;`
  - Disallow if SUP seen
- `+Comp:^ejš Asoft;`
  - Always allowed

Non-interactive Runs

```
foma[1]: save stack en.bin
Writing to file en.bin.
foma[1]: exit

> echo begging | flookup en.bin
begging beg+V+PresPart

> echo beg+V+PresPart | flookup -i en.bin
beg+V+PresPart begging
```

Czech Lexicon Example

```
Multichar_Symbols
+NF +Masc +Fem +Neut +Sg +Pl
+Nom +Gen +Dat +Acc +Voc +Loc +Ins
LEXICON Root
Noun ;
Adj ;
AdjSup ;
LEXICON Noun
žena:žen   NFzena;
matka:matk NFzena;
LEXICON NFzena
+NF+Sg+Nom:^a #;
+NF+Sg+Gen:^y #;
+NF+Sg+Dat:^e #;
+NF+Sg+Acc:^u #;
...
```

Czech Rules Example

- *matk* + *^0* → *matek*

```
define NFPlGenEInsertion [t k]->[t e k] || _ "^" λ;
```

Czech Rules Example

- *matk* + ˆ*0* → *matek*

```
define NFPlGenEInsertion [t k]->[t e k] || _ "^" λ;
```

- *matke* → *matce*, *žene* → *žeňe*

```
define NFSgDatPalatalization k->c, n->ň || _ "^" e;
```

Czech Rules Example

- *matk* + *ˆ0* → *matek*

```
define NFPlGenEInsertion [t k]->[t e k] || _ "^" λ;
```

- *matke* → *matce*, *žene* → *žeňe*

```
define NFSgDatPalatalization k->c, n->ň || _ "^" e;
```

- *ďe ťe ňe* → *dě tě ně*

```
define DeTeNe [ď "^" e]->[d "^" ě], [ť "^" e]->[t "^" ě],
    [ň "^" e]->[n "^" ě];
```

Czech Rules Example

- *matk* + ^*0* → *matek*

```
define NFPlGenEInsertion [t k]->[t e k] || _ "^" λ;
```

- *matke* → *matce*, *žene* → *žeňe*

```
define NFSgDatPalatalization k->c, n->ň || _ "^" e;
```

- *ďe ťe ňe* → *dě tě ně*

```
define DeTeNe [ď "^" e]->[d "^" ě], [ť "^" e]->[t "^" ě],
    [ň "^" e]->[n "^" ě];
```

- Finally erase temporary symbols

```
define Surface "^" -> 0, λ -> 0;
```

Czech Rules Example

- *matk + ^0 → matek*

```
define NFPlGenEInsertion [t k]->[t e k] || _ "^" λ;
```

- *matke → matce, žene → žeňe*

```
define NFSgDatPalatalization k->c, n->ň || _ "^" e;
```

- *ďe ťe ňe → dě tě ně*

```
define DeTeNe [ď "^" e]->[d "^" ě], [ť "^" e]->[t "^" ě],
    [ň "^" e]->[n "^" ě];
```

- Finally erase temporary symbols

```
define Surface "^" -> 0, λ -> 0;
```

```
read lexc cs.lexc
define Lexicon;
regex Lexicon .o. NFPlGenEInsertion .o. NFSgDatPalatalization .o.
    DeTeNe .o. Surface;
```

Homework

- Pick a language
- Cover a “reasonable” part of its morphology
- Details:
  http://ufal.mff.cuni.cz/~zeman/vyuka/morfosynt/lab-twolm/index.html
- Deadline:
  Wednesday January 6, 23:59 CET

- **WARNING: NO CLASS NOVEMBER 20!**