Two-Level Morphology

Daniel Zeman

http://ufal.mff.cuni.cz/course/npfl094
Two-Level (Mor)Phonology

- Testable using \texttt{pc-kimmo} (freely available at \url{http://www.sil.org/pckimmo/}).
- Lauri Karttunen (Xerox Grenoble): two-level compiler, finite state technology, \texttt{xfst}, see \url{http://www.xrce.xerox.com/}.
- Morphological “classics”
Finite-State Automaton

- 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) \(A \times Q \rightarrow Q\).
  - \(q_0 \in Q\) ... 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 **cs**: dě, tě, ně…
- Czech orthographical rules:
  - dí, tí, ní is pronounced [dɨ̞, tɨ̞, ňɨ̞]
  - dě, tě, ně is pronounced [dě̞, tě̞, ňe̞]
  - Orthography prohibits strings dī, tī, ňi, dŷ, tŷ, ňy, dē̞, tē̞, ňe̞, dě̞, tě̞, ňe̞
  - Note however that long dě̞, tě̞ is permitted: these are the names of the letters Ď, Ć. (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):
  - tǐn … pinyin equivalent is jin
Example of Finite-State Machine

- Checks correct spelling of cs: dě, tě, ně…
- Ignores official exceptions (“ť’in” … Czech transcription of Chinese “jin”) a|o|…

\[ q_0 \xrightarrow{d|t|n} q_1 \xrightarrow{d'|t'|ň} q_4 \]
\[ q_0 \xrightarrow{d|t|n} q_2 \xrightarrow{e|ē|i|í|y|ý} q_5 \]
\[ q_0 \xrightarrow{\text{other}} q_3 \]

† ERROR
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 $F$.
- At sign ("@") means “other”, i.e. characters not found on other transitions with the same start.
Lexicon

- Implemented as a finite-state automaton (trie) \([trie:\])
- Compiled from a list of strings and sublexicon references.
- Sublexicons for stems, prefixes, suffixes.
- Notes (glosses) at the end of every sublexicon.
- Example: (edges labeled same way as nodes they lead to)

\[ N:\text{bank} \]

\[ N:\text{book} \]

\[ \text{plural} \]
Lexicon

- Implemented as a finite-state automaton (trie) [tri:].
- Compiled from a list of strings and sublexicon references.
- Sublexicons for stems, prefixes, suffixes.
- Notes (glosses) at the end of every sublexicon.
- Example: (edges labeled same way as nodes they lead to)
Continuation Classes

• Unlike trie the lexicon is not a tree but a DAG (directed acyclic graph).
• The lexicon knows a continuation class (alternation) for each entry.
• Continuation class is the set of sublexicons to which one may transfer from the end of the current sublexicon (after accepting an entry).
• For example, one could traverse from the sublexicon of noun stems to one of the sublexicons of the case-marking suffixes.
• There are as many continuation classes for noun stems as there are noun paradigms (see example in pc-kimmo).
Examples of Lexicons

- English noun stems (typically whole words at the same time): book, bank, car, cat, donut…
- See also pc-kimmo / englex.
- Czech stems (not always a whole lemma): pán, hrad, muž, stroj, (před)sed, soudc, žen, růž, píseň, kost, měst, moř, kuř, staven
- Czech prefixes: do, na, od, po, pře, před, při, se, z, za… odpo, dopři, pona… nej, ne dvoj, troj…
Examples of Lexicons

• Suffixes of Czech nouns
  – 0, a, e, u, ovi, i, o, em, ou, i, ové, y, ů, ům, ech, ích
  – a, e, 0, y, i, u, o, ou, í, ám, ím, em, ách, ích, ech, ami, emi, mi
  – o, e, í, a, ete, u, i, eti, em, etem, ím, ata, 0, at, ům, ím, atům, ech, ích…

• Suffixes of Czech adjectives
  – ý, ého, ému, ým, í, ých, é, ými, á, ou, ém
  – í, ího, ímu, ím, ích, ími
  – (ej+, ěj+) š + í, ího, …

• Suffixes of Czech verbs
  – (n+) u, eš, e, eme, ete, ou
  – ím, íš, í, íme, íte, í
  – ám, áš, á, áme, áte, ají
  – (e+, u+) (j+) 0, me, te
  – l, en, t
  • 0, a, o, i, y, y, a
PC Kimmo Demonstration

- List of Czech nouns
- List of suffixes of inflection class žena
  - t cs
  - r žena
  - r ženy
- Separate entry for each interpretation of +y so we can have different glosses
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!
Buy One Get One Free: Morphology and Phonology

- Integration of morphology and phonology is possible and easy.
- Phonology is what is really “two-level” here.
- Morphology (morphemics): Connected lexicons implemented using finite-state automata (FSA) (just seen).
- Phonology: two-level. Set of rules implemented using finite-state transducers (FST). Example of a rule:

```
baby + 0s
babies
```
Two-Level Rules

- **Upper and lower language**
  - Upper is also called **lexical**.
  - Lower is also called **surface**.

- Two-line notation is encoded using colons:

  \[
  \begin{align*}
  &b:a \quad b:y + 0 \quad s:b \\
  &b:a \quad b:i \quad 0 \quad e:s
  \end{align*}
  \]

- \(b:b\) \(a:a\) \(b:b\) \(y:i\) \(+:0\) \(0:e\) \(s:s\)

- The + character usually denotes morpheme boundary.
- The 0 character usually denotes an empty position (its counterpart has no realization on this level).
- Other special characters of PC-Kimmo: #, @.
Finite-State Transducer

- 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)
- Analysis
  - input: sequence s ∈ S (two-l morphology: surface notation)
  - output: sequence r ∈ R (two-l morphology: lexical notation) + additional information from lexicon
- Generating
  - same as analysis but swapped roles S ↔ R
Automaton vs. Transducer

N: baby

plural

N: book

plural
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.
  \[ y:i \leq _ +:0 \ s:s \]
  - We don’t require the reverse implication this time. It is possible that y is changed to i elsewhere for other reasons.
- At the same time we require that in the same context an e is inserted before s:
  \[ 0:e \leq y:i +:0 _ s:s \]
- Create finite-state transducer that converts the lexical layer to the surface one according to the rules.
  - More precisely: a transducer is an automaton that only checks that we are converting the layers correctly.
Example of Transducer: \textit{baby+s}

\[ y : i \leq _ + : 0 \ s : s \]

\[\begin{array}{c}
\text{N: non-terminal state} \\
\text{F: terminal state} \\
\text{E: error state}
\end{array}\]
How to Get the FST Input

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

\begin{itemize}
  \item \textbf{N}: non-terminal
  \item \textbf{F}: terminal state
  \item \textbf{E}: error state
\end{itemize}

Explicitly add \textit{y:i} to some transducer so the system knows about the possibility.
Example of Transducer:

\textit{baby+s}

\begin{align*}
0: & e \\ y: & i \\ :: & 0 \\ _ & s:s
\end{align*}

N: non-terminal state

F: terminal state

E: error state
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 also assume for all \( x \) the default conversion rule \( x:x \).
Lexicon and Rules Together

Diagram showing the process of forming words from a lexicon and applying rules. Nodes labeled F1, F2, F3, E0 represent different stages of the process. Arrows indicate transitions between these stages, with labels such as 'b', 'a', 'y', 'o', and 's'. The words 'baby', 'book', and 'plural' are depicted at the end of the process, showing the application of rules to form plural forms.
Two-Level Morphological Analysis

1. Initialize set of paths $P = \{\}$.  
2. Read input symbols one-by-one.  
3. For each symbol $x$ generate all lexical symbols that may correspond to the empty symbol ($x:0$).  
4. Extend all paths in $P$ by all corresponding pairs ($x:0$).  
5. Check all new extensions against the phonological transducer and the lexical automaton. Remove disallowed path prefixes (unfinished solutions).
Two-Level Morphological Analysis

6. Repeat 4–5 until the maximum possible number of subsequent zeroes is reached.

7. Generate all possible lexical symbols (of all transducers) for the current symbol. 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 all impossible paths.

10. Repeat since step 3 until input finishes.

11. Collect glosses from the lexicon from all paths that survived.
Algorithm Example

Diagram of a formal language algorithm with states and transitions.
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
- ... $0 : e$ $+ : 0$ ... OK
- ... $0 : e$ $+ : 0$ $+ : 0$ ... error
- ... $+ : 0$ $0 : e$ $+ : 0$ ... error
- ... $0 : e$ $s : s$ ... error
- ... $+ : 0$ $0 : e$ $s : s$ ... OK
- ... $0 : e$ $+ : 0$ $s : s$ ... OK
- ... $+ : 0$ $0 : e$ $s : s$ $+ : 0$ ... error
- ... $0 : e$ $+ : 0$ $s : s$ $+ : 0$ ... error
- One of the hypotheses could be blocked by our FSTs if we designed them better ($\Leftrightarrow$)
Example of Transducer: \textit{baby+s}
Czech Examples

- Joining stem with suffix may for instance bring together d’ and e that normally cannot occur together. \((kád’ = tun)\)
  
  \[
  \begin{align*}
  kád’ + e \\
  kád’ 0 e
  \end{align*}
  \]

- We need a rule for such cases that will ensure the correct conversion d’e → dě.
  
  \[
  \begin{align*}
  kád’ + e \\
  kád 0 ě
  \end{align*}
  \]
Example of Transducer: d’, t’, ň on morpheme boundary

• d’:ď + 0 e:ě is correct, other possibilities are not.
• Assumption: d’e, d’i could only occur on morpheme boundary (other positions are in the lexicon and should be correct).
• We don’t cover d’ě. The character ě can appear in the suffix only because of a phonological change, not otherwise:
  – (brzda brzď’e, žena žeñe, máta máť’e, máma mámňe, bába bábje, matka matće, váha váże, sprcha sprš’e, kůra kůře, mula mule, vosa vose, lůza lůze)
• We further don’t cover d’y (which could arise by application of the inflection paradigm to a noun ending in –ď’a; it is incorrect and should be changed to –di).
Example of Transducer: $d', t', n$ on morpheme boundary

N: non-terminal state
F: terminal state
E: error state
Example of Transducer: d', t', ň on morpheme

Possible conversions:
- d': d
- t': t
- ň: n
- +: 0
- e: ě
- i: i
- í: í

E: error state
Example of Transducer:
\(d', t', \tilde{n}\) on morpheme boundary

Possible conversions:
- \(d':d \ @\)
- \(t':t \ @\)
- \(\tilde{n}:n \ @\)
- \(+:0\)
- \(e:e \ @\)
- \(i:i\)
- \(\tilde{i}:\tilde{i}\)
- \(\@:\@\)
Example of Transducer: d’, t’, ň on morpheme

Add alphabet:
- d’: d d’
- t’: t t’
- ň: n ň
- +: 0
- e: ě e
- i: i
- í: í
- x: x ...
Example of Transducer: d’, t’, ň on morpheme boundary

- N: non-terminal state
- F: terminal state
- E: error state

Transducer Encoding in a Matrix

**RULE**

\[ [\ddot{d}:d \mid \ddot{n}:n \mid \ddot{t}:t] \iff _ +:0 [e:ě \mid i:i \mid í:í] \] 5 12

<table>
<thead>
<tr>
<th>d' ň t' d' ň t' + e i í e @</th>
</tr>
</thead>
<tbody>
<tr>
<td>d n t @ @ @ 0 ě i í @ @</td>
</tr>
</tbody>
</table>

1: 2 2 2 4 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 1 1 1 1 1
5: 2 2 2 4 4 4 1 0 0 0 0 1

6.11.2009

http://ufal.mff.cuni.cz/course/npfl094
The pairs illustrate various stem-final changes in the paradigm žena of Czech feminine nouns. All words are surface strings—nominitive singular on the left, dative singular on the right.

- váha – váze
- sprcha – sprše
- matka – matce
- kůra – kůře
- Olga – Olze
- vláda – vládě
- máta – mátě
- žena – ženě
- bába – bábě
- karafa – karafě
- máma – mámě
- chrpa – chrpě
- jíva – jívě
- Naďa – Nadě
- Jíťa – Jítě
- Áňa – Áně
Palatalization žena – ženě

\[ H:Z = g:ž | h:ž | ch:š | k:c | r:ř \]
\[ B:B = b:b | f:f | m:m | p:p | v:v | w:w | q:q | d:d | t:t | n:n \]
\[ ç:č | ě:ě \]
PC Kimmo Demonstration

• r ženě matce Bláže Nadě…
• Separate paradigms žena and růže using continuation classes
  • r Nadi
  • g Naď’y
Examples of Two-Level Rules in Czech

- Palatalization of stem-final consonants.

  \[\text{matEK} + e\]
  \[\text{mat0c0e}\]

- Epenthesis: inserting or deleting of \(e\).

  \[\text{matEK}\]
  \[\text{matek}\]

- Transitions among present, past and infinitival verbal stems.
  - Palatalization of stem-final consonant in imperative.
PC Kimmo: 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... jarnější, jarnějšího...

- Irregular comparatives: mladý (young) ⇒ mladší (younger); snadný (easy) ⇒ snadnější | snazší (easier)

- Superlative: nej + comparative, e.g. nejmladší (youngest)
PC Kimmo: Czech Adjectives

entries  continuation classes  lexicons  entries

- mlad → AdjTInfl → ADJTINFL → ý
- snadn → AdjTDS → ADJDEG → ejš
- mladš → AdjMInfl
- snazš → AdjMInfl
- jarn → AdjMDS → ADJMINTFL → í
PC Kimmo Demonstration

- Adjectives: irregular comparison
  - r mladý mladší *mladější
  - r snadný snazší snadnější
  - r jarní jarnější
- Palatalization works here, too:
  - jarn + ejš + í = jarnější
  - r nejmladší *nejmladý
Long-Distance Dependencies

• Disadvantage of TLM:
  – Capturing of long-distance dependencies is clumsy!
Example from German

- German umlauts (simplified):
  - $u \leftrightarrow \ddot{u}$ if (not only if) followed by $c h e r$ (Buch $\rightarrow$ Bücher)
  - pravidlo: $u:ü \iff$
  - __c:cc h:hh e:ee r:rr_

FST:

Buch:
F1 F3 F4 F5

Buch: [diagram]
F1 F3 F4 F5 F6 E0

This detour only defines what “$u:@$” means.
Example from German

- **Buch / Bücher, Dach / Dächer, Loch / Löcher**
- Context should also contain `+:0` and perhaps test end of word (`#`)
  - Otherwise *Sucherei* (searching) will be considered wrong!
  - Not only must we 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 umlaut-less *Kocher* an error!
    - *Besucher* (visitor), derived from *Besuch* (visit), same singular and plural, there is no *Besücher*!
- Capturing long-distance dependencies is clumsy.
  - E.g. *Kraut / Kräuter* has different intervening symbols so it looks like a different rule.
  - A transducer could be more general and allow anything until `+er` but would it overgenerate?
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 this right-hand context we *must* change d’ to d
- Notation:
  \[ \text{d’} : \text{d} \leq _+ 0 \text{ e@} \]
- (However, unless there are other rules, by this we have permitted d’:d in other contexts as well.)
Two-Level Grammar

• \( x:y \leq lc \_ rc \)
  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.

• \( x:y \rightleftharpoons lc \_ rc \)
  If and only if \( x \) is found in this context, it surfaces as \( y \).

• \( x:y \not\leq lc \_ rc \)
  \( x \) never surfaces as \( y \) in this context.
Two-Levelness and the Lexicon

• The lexicon contains only lexical (upper) symbols.
  – Their relation to the surface level is expressed solely by the transducers.

• On the other hand there are the *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 
  
- **Generation (synthesis)** is the transition from the lexical to the surface level.
  - Typical input would be glosses rather than morphemes.
  
- book $+\text{plural}$ $\Rightarrow$ book+s $\Rightarrow$ books
Lexicon for Analysis

- Implemented as FSA (trie).
- Compiled from a list of strings and inter-lexicon links.
- Sublexicons for stems, prefixes, suffixes.
- Notes (glosses) at the end of each sublexicon.
Lexicon for Generation

- Swap surface and lexical levels (glosses).
- Again, it can be automatically compiled from the same list as the lexicon for analysis.
- The rest works the same way.
Generation in PC Kimmo Version 2

- Originally only concatenation of morphemes:
  - g žen+e

- Newly in PCK v.2:
  - l synthesis-lexicon cs.lex
  - s N(žena) +SG+LOC
How to Fill the Lexicon?

• See *lexicon acquisition* in earlier presentations.
• E.g. we have *annotated corpus* (lemmas, tags)
  – Created using existing morphological analyzer
  – The analyzer is not available to us and we want to create our own
• Easy:
  – All Czech words under the paradigm *žena*:
    • Extract words tagged **NNFS1.** whose lemma ends in –a
    • Some words have not occurred in singular nominative. If we don’t want to lose them we repeat the procedure with another characteristic form of the paradigm *žena*.  
How to Fill the Lexicon?

- We have only unannotated corpus
  - What inflection class (paradigm) does a new word belong to?
  - Hypothesis: the word města belongs to paradigm žena.
  - Ask PC Kimmo to generate all forms of this word according to the paradigm, then search for them in the corpus.
  - Problem 1: what is stem and what suffix?
    - Gradually try m+ěsta, mě+sta, měs+ta, měst+a, města+λ
  - Problem 2: what about phonological changes? What if I’m investigating the word matce?
    - Gradually try all combinations of symbol changes allowed by the rule file for PC Kimmo
Multi-Level Finite State Rules

Daniel Zeman

http://ufal.mff.cuni.cz/~zeman/
XFST

- Xerox Finite State Toolkit
  - xfst, lexc, tokenize, lookup
  - Binaries and API for multiple operating systems
  - [http://cs.haifa.ac.il/~shuly/teaching/06/nlp/xfst-tutorial.pdf](http://cs.haifa.ac.il/~shuly/teaching/06/nlp/xfst-tutorial.pdf)
  - [http://cs.haifa.ac.il/~shuly/teaching/06/nlp/fst2.pdf](http://cs.haifa.ac.il/~shuly/teaching/06/nlp/fst2.pdf)
- Current version uses UTF8 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
  - 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/
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 a+
- regex cat | dog
- regex ?* a ?*
- regex [a:b | b:a] *
- regex [cat]:[katua]
- 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)
Difference between Colon “:” and Arrow “→”

- Colon “:” affects a specific position or 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.
Testing Automata against Words

\texttt{foma}[0]: \texttt{regex} ?* a ?*;

261 bytes. 2 states, 4 arcs, Cyclic.

\texttt{foma}[1]: \texttt{down}

apply \texttt{down}> \texttt{ab}

\texttt{ab}

apply \texttt{down}> \texttt{bbx}

???

apply \texttt{down}> \texttt{CTRL+D}

\texttt{foma}[1]:

6.11.2009

http://ufal.mff.cuni.cz/course/npfl094
Graphical Visualization in Linux

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

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.
Rewrite Rules

\[
\begin{align*}
foma[0]: & \quad \text{regex } a \rightarrow b; \\
& \quad \text{290 bytes. 1 states, 3 arcs, Cyclic.}
\end{align*}
\]

\[
\begin{align*}
foma[1]: & \quad \text{down} \\
& \quad \text{apply down} > a \\
& \quad b \\
& \quad \text{apply down} > axa \\
& \quad bxb \\
& \quad \text{apply down} > \text{CTRL+D}
\end{align*}
\]

Accepts any input. Changes \(a\) to \(b\).
Conditional Replacement

\texttt{foma[0]}: \texttt{regex a \to b || c \_ d ;}
526 bytes. 4 states, 16 arcs, Cyclic.
\texttt{foma[1]}: \texttt{down cadca}
\texttt{cbdca}
\texttt{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
cbdebfb
apply down> a
a
apply down> CTRL+D
Parallel Rules
End-of-Word Symbol

foma[0]: \texttt{regex b \rightarrow p, g \rightarrow k, d \rightarrow t ||}
_ .#. ;
634 bytes. 3 states, 20 arcs, Cyclic.

foma[1]: down
apply down, cab

apply down, dog
dok

apply down, dad
dat
Composition of Rules

\begin{verbatim}
\texttt{foma[0]: define Rule1 \textit{a} -> \textit{b} || \textit{c} \_ ;}
defined Rule1: 384 bytes. 2 states, 8 arcs, Cyclic.
\texttt{foma[0]: define Rule2 \textit{b} -> \textit{c} || \_ \textit{d} ;}
defined Rule2: 416 bytes. 3 states, 10 arcs, Cyclic.
\texttt{foma[0]: regex Rule1 .o. Rule2;}
574 bytes. 4 states, 19 arcs, Cyclic.
\texttt{foma[1]: down}
apply down\textgreater{} cad
cad
apply down\textgreater{} ca
cb
apply down\textgreater{} ad
ad
\end{verbatim}
Review

- **regex** regular-expression;
  - compile regular expression and put it on the stack

- **define** name regular-expression;
  - name a FST/FSM using regex; do not put it on the stack

- **view** *(view net)*
  - (Linux only) display the compiled regex from stack graphically in a window

- **net** *(print net)*
  - textual net description

- **down** *
  - run a lexical word through a transducer (generation)

- **up** *
  - run a surface word through a transducer (analysis)

- **words** *(print words)*
  - print all the words an automaton accepts

- **lower-words**
  - only lower side of an FST

- **upper-words**
  - only upper side of an FST
Lexicon in lexc Format

- Create the file, then load it to Foma

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

LEXICON Suff
  s  #;
    #;
```
Load Lexicon to Foma

foma[0]: read lexc simple.lexc
Root...3, Suff...2
Building lexicon...Determinizing...Minimizing...Done!
575 bytes. 13 states, 15 arcs, 8 paths.
foma[1]: print words
horse horses dog dogs cat cats
foma[1]: define Lexicon;

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  Ninf;
city Ninf;

LEXICON Ninf
+N+Sg:0  #;
+N+Pl:^s  #;
!^ is our morpheme boundary
Put It All Together

- **Lexical string** = city+N+Pl
- **Lexicon transducer**: city+N+Pl $\rightarrow$ city^s
- $y \rightarrow ie$ rule: city^s $\rightarrow$ citie^s
- **Remove ^**: citie^s $\rightarrow$ cities
- **Surface string** = cities
Put It All Together

\begin{verbatim}
\texttt{foma}[0]: read lexc en2.lexc
\texttt{foma}[1]: define Lexicon;
\texttt{foma}[0]: define YRepl y \rightarrow i e \mid i e \mid _ ^ "^"
\texttt{s};
\texttt{foma}[0]: define Cleanup "^" \rightarrow 0;
\texttt{foma}[0]: regex Lexicon .o. YRepl .o. Cleanup;
\texttt{foma}[1]: lower-words
\texttt{cat cats city cities ...}
\end{verbatim}
Irregular Forms

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

6.11.2009
Priority Union

foma[1]: define Grammar;
foma[0]: define Exceptions [make "+V" "+PastPart":[made];
   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[1]: regex Parallel | Grammar;

...
Long-Distance Dependencies

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

```plaintext
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 feat/val $@R.feature(.value)@$  
  – D … disallow feat/val $@D.feature(.value)@$  
  – C … clear feature $@C.feature@$  
  – E … require equal feat/val $@E.feature.value@$
Flag Diacritics to Control Czech Superlatives

- Multichar_Symbols Sup+ +Pos +Comp
  \textsc{P.Sup.On} \textsc{D.Sup}
- LEXICON AdjSup
  \textsc{P.Sup.On}@Sup+:@P.Sup.On@nej\^{\text{^Adj;}}
- LEXICON AhardDeg
  \textsc{D.Sup}+Pos:@D.Sup@ Ahard;
  +Comp:^ej\v{s} Asoft;
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  #;
  - ...

Czech Rules Example

• # matk + ^0 --> matek
define NFPlGenEInsertion [t k]->[t e k] || _ "^" \;

• # matke -> matce, žene -> žeňe
define NFSgDatPalatalization k->c, n->ň || _ "^" e;

• # de te ňe -> dě tě ně
define DeTeNe [ď "^" e]->[d "^" ě], [ť "^" e]->[t "^" ě
ě], [ň "^" e]->[n "^" ě];

• # Finally erase temporary symbols.
define Surface "^" -> 0, \ -> 0;

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

Foma: Czech Demo

- `foma -l cs.foma`
Unsorted Notes

• Rozdíl mezi dvojtečkou a šipkou?
  – Šipka se implementuje pomocí dvojtečky.
  – Dvojtečka ovlivňuje konkrétní pozici nebo posloupnost pozic.
  – Regexy s šipkou vedou na převodníky, které přijímají libovolný řetězec, ale pokud v něm narazí na hledaný znak, nahradí ho.
  – Dvojtečky se používají v regexechech, které omezují množinu slov patřících do jazyka.
  – Srovnaj tyto převodníky:
    • regex ?* a ?*;
    • regex ?* a:b ?*;
    • regex a -> b;

• Proč označují hranici morfému znakem „^“? Proč mi nefunguje „+“?
Unsorted Notes from Sproat’s Book

- Phonological word (stress) / Syntactic word (clitic) / Lexical word (lexeme; multiword expressions) / Orthographic word (sometimes may not match any of the above)
- Long-distance effects: vowel harmony in Finnish (solved in Koskenniemi’s thesis)
- Morphotactics: what morphemes can occur in what order?
- Traditional generative phonology (Chomsky’s followers): successive application of ordered rules
- Kimmo example: Sanskrit consonant harmony (p. 134, uSnatarānām)
- Kay and Kaplan (p. 139) proposed cascaded FSTs. They had upper and lower tape and several intermediate tapes in between. Analysis problem: non-determinism can cause the number of intermediate tapes to grow exponentially
- DECOMP (p. 184; mid 1960s): a model of English morphotactics and a recursive morph-partitioning algorithm. Free root can appear either alone or with prefixes and suffixes: side, cover, spell. Absolute morph disallows most affixes: the, into, of. Prefix morphs: pre-, dis-, mis-. Left-functional roots must always be followed by a derivational suffix: nomin- (e.g. nominate, nominee). Derivational suffix: -ness, -ment, -y. Regular grammar describes the permitted combinations.
Slovník pro syntézu

Inf|Pas
Pre|Imp
Past

něs|nes|nes
brát|ber|bra
1 → +u
2 → +eš
3 → +e
1 → +eme
2 → +ete
3 → +ou

6.11.2009
http://ufal.mff.cuni.cz/course/npfl094