Computational Morphology: Morphological operations

Yulia Zinova

09 April 2014 – 16 July 2014

Root-and-Pattern Morphology

- Best-known example of root-and-pattern morphology: derivational morphology of the verbal system of Arabic;
- the first formal generative treatment – McCarthy (1979);
- Semitic languages derive verb stems - actual verbs with specific meanings - from consonantal roots;
- the overall prosodic “shape” of the derivative is given by a prosodic template (in McCarthys original analysis a CV template)
- the particular vowels chosen depend upon the intended aspect (perfect or imperfect) and voice (active or passive).

Examples

- Active forms with the root ktb "notion of writing"

Pattern	Template	Verb Stem	Gloss
I	$C_1aC_2aC_3$	katab	"wrote"
II	$C_1aC_2C_2aC_3$	kattab	"caused to write"
III	$C_1aaC_2aC_3$	kaatab	"corresponded"
IV	$aC_1C_2aC_3$	aktab	"caused to write"
VI	$taC_1aaC_2aC_3$	takaatab	"wrote to each other"
VII	$nC_1aC_2aC_3$	nkatab	"subscribed"
VII	$C_1aC_2aC_3$	katab	"copied"
X	$staC_1C_2aC_3$	katab	"caused to write"

General Architecture

- We will assume that we are combining two elements, the root and the vocalized stem;
- we define the root P as follows:
  $P = ktb$
- we assume that the templates are represented more or less as in the standard analyses;
- exception: the additional affixes that one finds in some of the patterns the *n*- and *sta*- prefixes in VII and X or the *-t* infix in VIII will be lexically specified as being inserted;
- This serves the dual purpose:
  - making the linking transducer simpler to formulate;
  - underscoring the fact that these devices look like additional affixes to the core CV templates (and presumably historically were).

Transducers

$\tau_I = CaCaC$
$\tau_{II} = CaCCaC$
$\tau_{III} = CaaCaC$
$\tau_{IV} = [\epsilon : a]CCaC$
$\tau_{VI} = [\epsilon : ta]CaaCaC$
$\tau_{VII} = [\epsilon : n]CaCaC$
$\tau_{VIII} = C[\epsilon : t]aCaC$
$\tau_X = [\epsilon : sta]CaCaC$

$$\tau = \bigcup_{p \in patterns} \tau_p$$

Last transducer

- Now we need a transducer to link the root to the templates;
- It must do two things:
    - it must allow for optional vowels between the three consonants of the root;
    - it must allow for doubling of the center consonant to match the doubled consonant slot in pattern II.
- The first part can be accomplished by the following transducer: $\lambda_1 = C[\epsilon : V]^* C[\epsilon : V]^* C$
- The second portion the consonant doubling requires rewrite rules (Kaplan and Kay, 1994; Mohri and Sproat, 1996) of the general form: $\lambda_2 = C_i \rightarrow C_i C_i$
- Then the full linking transducer λ can be constructed as: $\lambda = \lambda_1 \circ \lambda_2$

Getting everything together

- The whole set of templates for *ktb* can then be constructed as follows:
  $\Gamma = P \circ \lambda \circ \tau$

Other approaches

- Most large-scale working systems for Arabic such as Buckwalter (2002), sidestep the issue of constructing verb stems and effectively compile out the various forms that verbs take.
- This is reasonable, given that the particular forms that are associated with a verbal root are lexically specified for that root, and the semantics of the derived forms are not entirely predictable.
- Another approach taken is that of Beesley and Karttunen (2000) who propose new mechanisms for handling non-concatenative morphology including an operation called *compile-replace*.
- The basic idea behind this operation is to represent a regular expression as part of the finite-state network, and then to compile this regular expression on demand.

Compile-replace: example

- Consider a case of total reduplication such as that found in Malay: a form like *bagi* “bag” becomes *bagibagi* “bags”.
- In Beesley and Karttunens implementation, a lexical-level form *bagi+Noun+Plural* would map to an intermediate surface form *bagiˆ2*.
- This itself is a regular expression indicating the duplication of the string *bagi*, which when compiled out will yield the actual surface form *bagi-bagi*.
- Thus for any input string *w*, the reduplication operation transforms it into the intermediate surface form *wˆ2*, which compile-replace then compiles out and replaces with the actual surface form.