Computational Morphology: trends, finite-states and open-source

(Evolución de la morfología computacional: nuevas posibilidades)

Mans Hulden
(University of Helsinki)
Iñaki Alegria
(University of The Basque Country)

Simp. de Comunicación Social, 2011. Santiago de Cuba

Grand outline

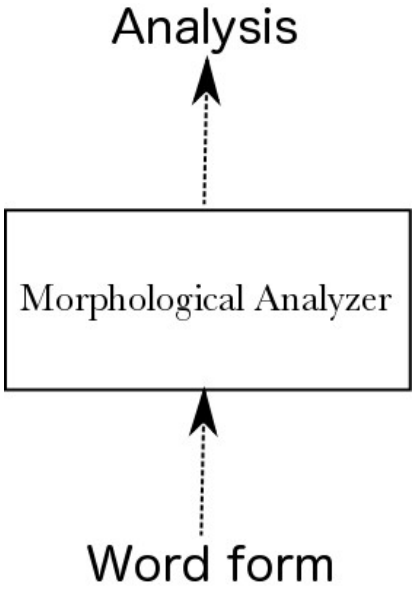
- Trends in computational morphology
- Morphological analysis with finite-state technology
- Tools for compiling automata and transducers
- Specifying the lexicon descriptions (*lexc*)
- Compiling grammars with *lexc* & rewrite rules
- Exceptions and advanced morphotactics
- Applications I: spell checking, spelling correction, etc.
- Applications II: surface syntactic parsing

Trends in computational morphology

- Solved problem using **finite-state technology**
  - ("*...is a solved problem*" [LK, 2003])
- But new research on:
  - Semitic languages
  - Derivation (and composition)
  - **Open source**
  - **Less-resourced languages**
- Linguistic approach, better than data-driven approach
- Data-driven morphology
  - Inference of morphology from corpus
  - Inferences of variants form standard and corpus
    - Digital libraries, dialects, less-resourced lang.

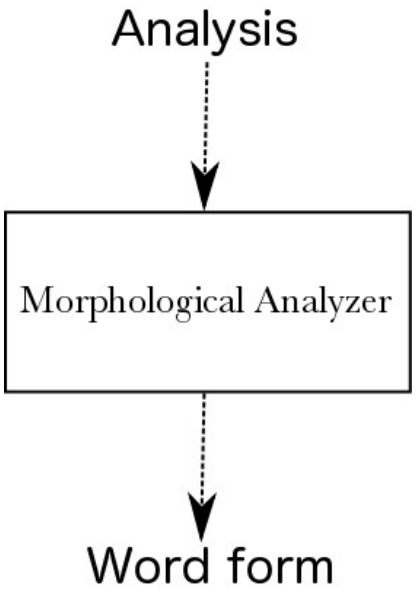
Morphological analysis...

String-to-string translation of word forms to analyses...



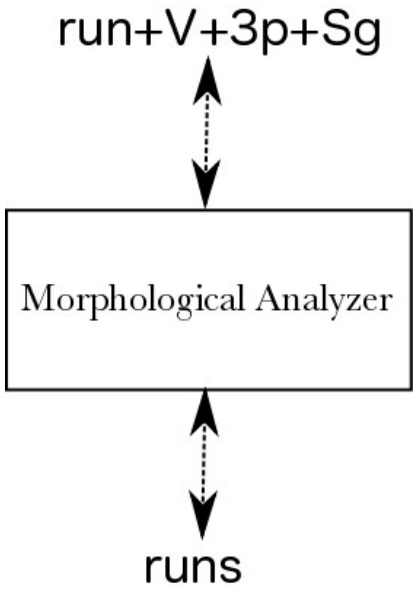
...and generation

String-to-string translation of analyses to word forms...



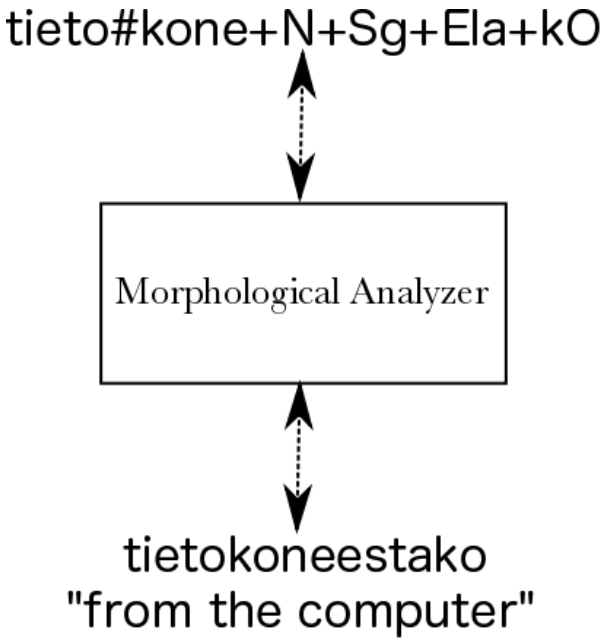
Morphological analysis

English example (simple)



Morphological analysis

Finnish example...
“tietokoneestako”
compound noun tieto + kone
singular
elative case
question particle



Finite-state technology ...

Solves the problem ("...is a solved problem" [LK, 2003])
Research in FST morphology since the early 1980s
Some different formalisms around
- Two-level rules (two-level morphology)
- Composed rewrite rules (generative SPE rules)

A variety of tools and applications around:
- Xerox xfst (commercial, for composed rewrite rules)
- Xerox twolc (commercial, for two-level rules)
- foma (GPL license, for composed rewrite rules)
- hfst-twolc (GPL license, for two-level rules)

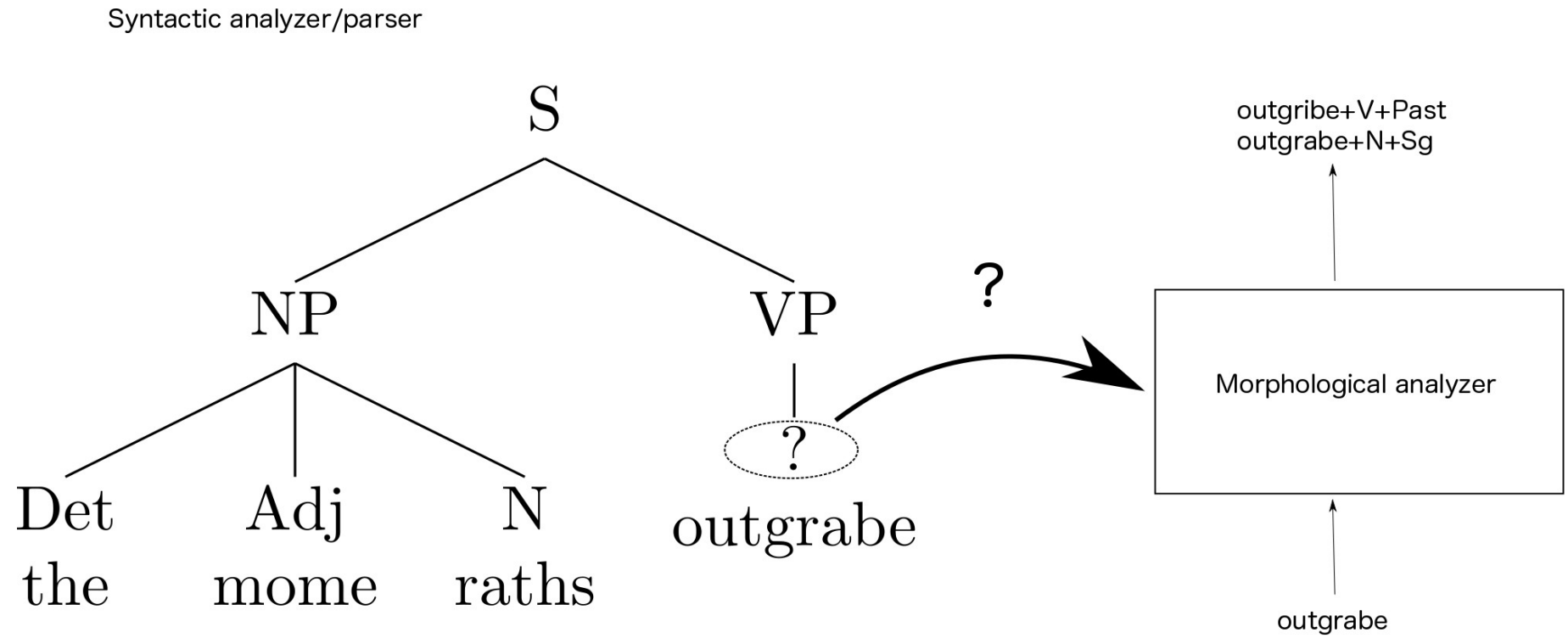
Freely available advanced tools fairly recent
- foma (since 2009, GPL license)
- HFST (since 2009, GPL license)

Applications that require morphological processing

POS Tagger
Shallow parser (chunker)
Syntactic parser
Information extraction
Text-to-speech
Machine translation

Example: syntactic parsing

Generally consults a separate morphological analyzer



Direct derivatives

With a finite-state morphological analyzer for a language, one can with very little effort produce a:

spell checker
spelling corrector (for various types of errors)
lemmatizer
verb conjugator
CALL tools
electronic dictionary tools

Practical goals

Give an overview of **finite-state technology**

Focus on **morphological analysis**

Learn how to write a morphological analyzer/generator with **freely available tools**

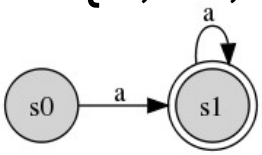
Learn how to create **derivative tools** once a morphological grammar is built: spell checker, spelling correctors, and more

Recognize other potential targets for finite-state technology: linguistic research (**phonology** and morphology), **syntactic parsing**

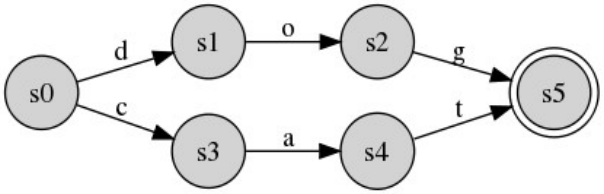
Computational Morphology: ***Compiling automata and transducers***

Recap: finite automata

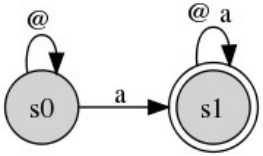
"one or more as": {a,aa,...}:



the words "cat" and "dog":



any word that contains at least an a:

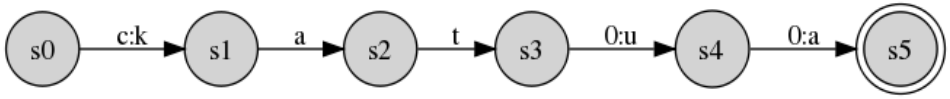


@ = any symbol outside the defined alphabet

Recap: finite transducers

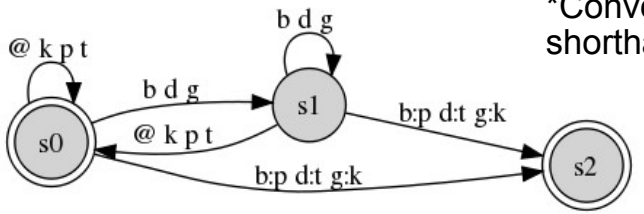
Translates all a-symbols to b and vice versa

Translates “cat” to “katua”



Devoice end-of-word stops:

xleb → xlep, rad → rat, etc.



*Convention: a single symbol on an arc (a) is shorthand for an identity pair (a:a)

Birds-eye view

Generative phonology/morphology tends to model word-formation processes and allomorphy by **minimizing different lexical forms of morphemes**

Eg.:

cat → cats	casa → casas
fox → foxes	pez → peces

The plural morpheme s can be held to be invariant, while surface-variation is introduced by phonological rules

Birds-eye view

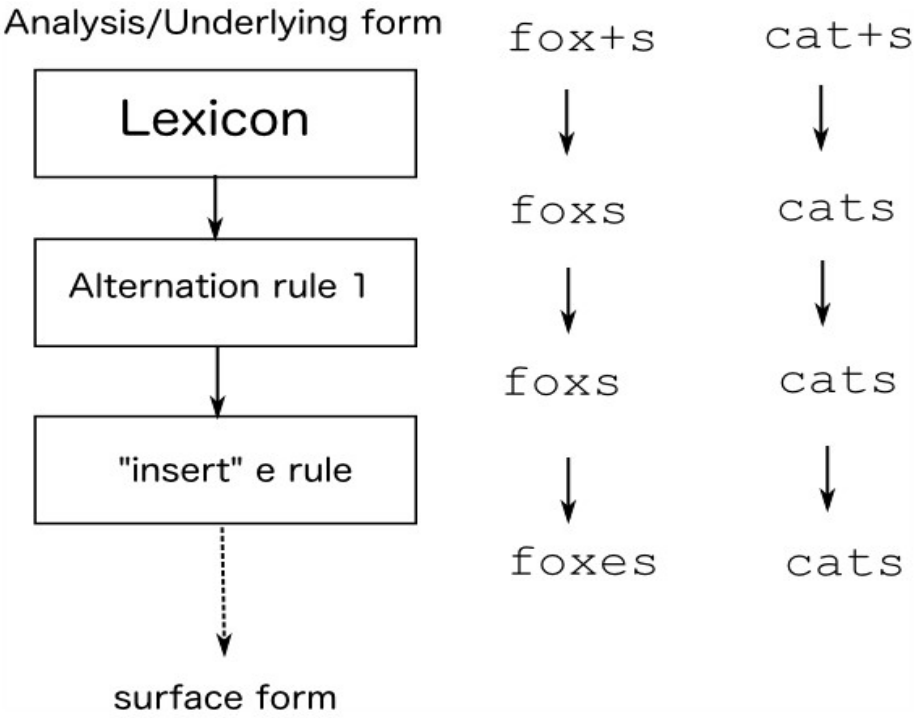
The theoretical mechanisms for such word-formation processes **include a *lexicon component*** (that guarantees proper morpheme ordering) **and a *phonological component*** (usually a set of ordered alternation rules)

Words are “derived” by:

- Constructing a morphotactically correct “underlying” form
- Subjecting this underlying form to various rewriting rules

Birds-eye view

Two different derivations



Birds-eye view

The different **stages of derivation** are modeled **through transducers**

The transducers are j**oined together by composing,** yielding a monolithic transducer with only a relation between the surface and underlying forms

Transducers are built by a special type of **regular expressions**...

Introduction to *foma*

A general-purpose tool for constructing and manipulating automata and transducers

Contains a **regular expression compiler** to convert expressions (including “rewrite rules”) to automata and transducers

Contains a *lexc*-parser to construct **transducers from lexicon descriptions**

Interface and regular expression formalism somewhat **compatible** with the commercial *xfst* and *lexc* tools by Xerox

Available at *http://foma.sf.net*

API available (in C) for integration with other programs [source & binaries for Linux, Mac, and Windows]

Introduction to *foma*

Unix-style command-line tool with interface

Installation & starting

Download appropriate files from http://foma.sf.net

Standard fare: place “foma” in your /usr/local/bin or /usr/bin (Linux and Mac), etc.

- Experimental support for FSM visualization (Linux and Mac)
- Linux: visualization requires “GraphViz” and “gqview”
  - Ubuntu example:
  - sudo apt-get install graphviz
  - sudo apt-get install gqview
- Mac:
  - Visualization requires GraphViz for OSX from http://www.pixelglow.net

foma: hands-on

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 || _ .#.;
```

[demo]

foma: hands-on

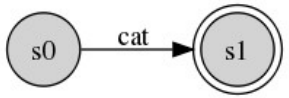
(space)	concatenation
|	union
*	Kleene star
&	Intersection
~	Complement

foma: ordinary symbols

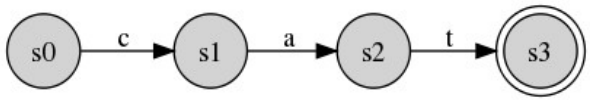
Single-character symbols:
a, b, c, Ω, ب , β, etc.

Multi-character symbols:
[Noun], +3pSg, @a_symbol@, cat, dog

foma[0]: regex cat;
168 bytes. 2 states, 1 arcs, 1 path.



foma[1]: regex c a t;
257 bytes. 4 states, 3 arcs, 1 path.



foma: special symbols

0 the empty string (epsilon)

? “any” symbol (similar to . in grep/perl/awk/sed-regexes, or Σ in “formal language” regexes)

foma: contd.

testing automata against words:

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

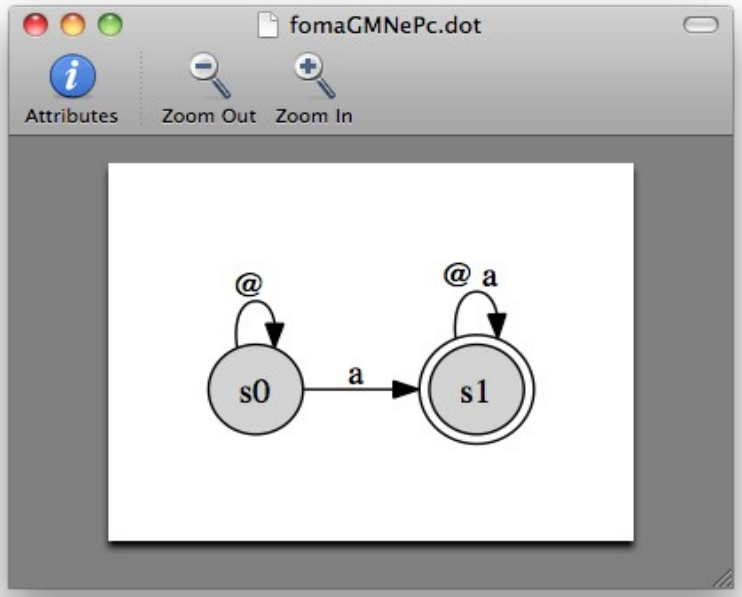
foma: contd.

running transducers:

```
foma[0]: regex [c a t]:[k a t u a];
317 bytes. 6 states, 5 arcs, 1 path.
foma[1]: down
apply down> cat
katua
apply down> dog
???

foma[1]: up
apply up> katua
cat
```

Examining FSMs visually

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



More about foma

Labeling FSMs: the define command

```
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]:
```

Define contd.

```
foma[0]: define V [a|e|i|o|u];
redefined V: 317 bytes. 2 states, 5 arcs, 5 paths.
foma[0]: define C [b|d|g|k|m|n|p|s|t|v|z];
defined C: 497 bytes. 2 states, 11 arcs, 11 paths.
foma[0]: define Syllable [C* V+ C*];
defined Syllable: 1.0 kB. 3 states, 43 arcs, Cyclic.

foma[0]: define PhonologicalWord Syllable+;
defined PhonologicalWord: 887 bytes. 2 states, 32 arcs, Cyclic.

foma[0]: print defined
V                   317 bytes. 2 states, 5 arcs, 5 paths.
StartsWithVowel     429 bytes. 2 states, 11 arcs, Cyclic.
C                   497 bytes. 2 states, 11 arcs, 11 paths.
Syllable            1.0 kB. 3 states, 43 arcs, Cyclic.
PhonologicalWord    887 bytes. 2 states, 32 arcs, Cyclic.
```

Transducer operations

Composition (operator: .o.)

```
foma[0]: define EngBasque [c a t]:[k a t u a];
defined EngBasque: 317 bytes. 6 states, 5 arcs, 1 path.
foma[0]: define BasqueFinn [k a t u a]:[k i s s a];
defined BasqueFinn: 331 bytes. 6 states, 5 arcs, 1 path.
foma[0]: regex EngBasque .o. BasqueFinn;
345 bytes. 6 states, 5 arcs, 1 path.
foma[1]: down
apply down> cat
kissa
apply down>
```

Replacement rules

Simple replacement:

```
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>
```

Replacement rules

Conditional replacement

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

Replacement rules

Conditional replacement w/ 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>
```

Replacement rules

“Parallel” rules, the .#.-symbol

Example: devoice some word-final stops

```
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
dat
```

Replacement rules & composition

We can define **multiple different rules and compose** them into one single transducer:

```
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
```

Review of basic *foma commands*

Compile regex:

```
regex regular-expression;
```

Name a FST/FSM using a regex:

```
define name regular-expression;
```

View (visually) a compiled regex:

`view` or `view net`

View (in text form) a compiled regex:

`net` or `print net`

Run a word through a transducer:

`down <word>` or `apply down <word>`

In the inverse direction:

`up <word>` or `apply up <word>`

Print all the words an automaton accepts:

`words` or `print words`

Only lower side words (for a transducer):

`lower-words` or `print lower-words`

Only upper-side words (for a transducer):

`upper-words` or `print upper-words`

Review of basic *foma regexes*

Special symbols 0 (epsilon) and ? (the “any” symbol)
[and] are grouping symbols
_ is a context separator (don't use in definitions)
.#. is a special symbol indicating left or right word boundary in replacement rules
Reserved symbols (operators) need to be quoted if used as symbols: eg. a “&” b;

space	concatenation
|	union
*	Kleene star
+	Kleene plus
&	Intersection
~	Complement
(A)	Optionality (identical to A | 0)

Transducer-related:

:	Cross-product
A -> B	Replacement rules
A -> B || C _ D	Context-conditioned replacement rules
.o.	Composition

Computational Morphology:
The lexc formalism

Simp. de Comunicación Social, 2011. Santiago de Cuba

lexc: Overview

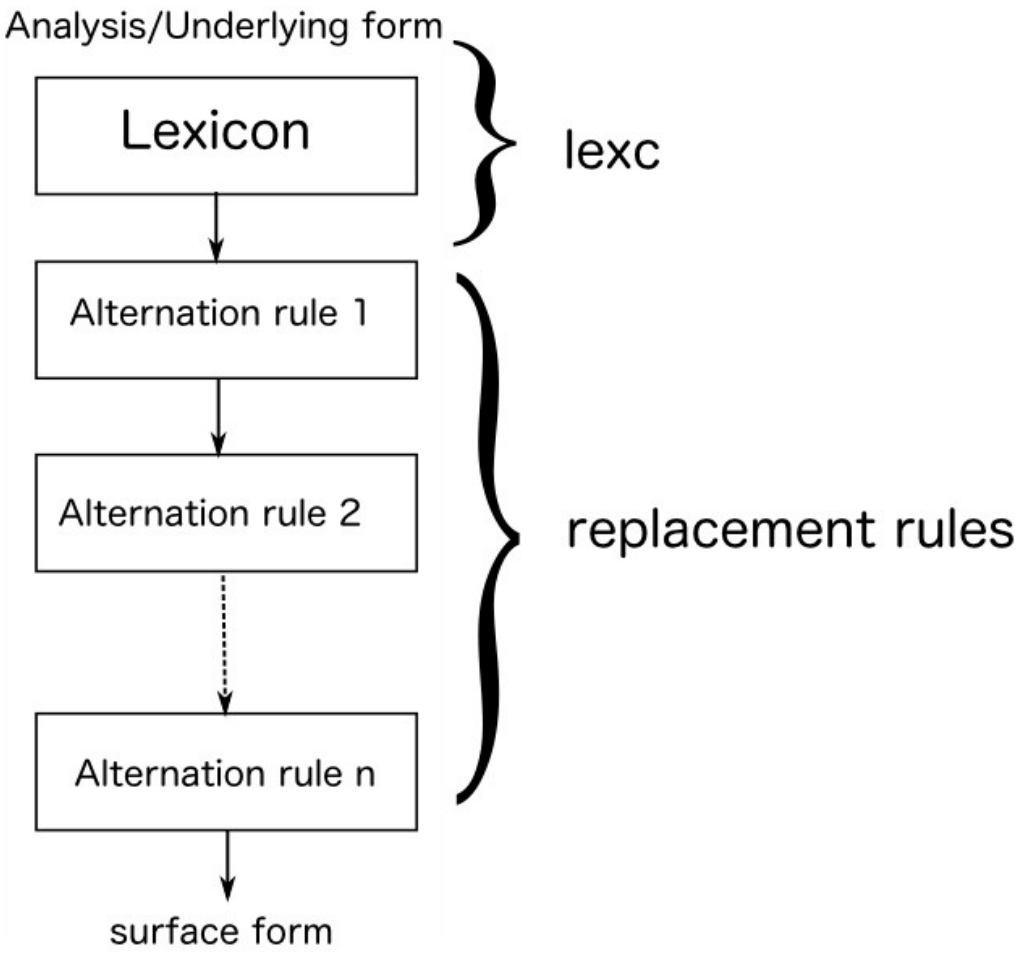
lexc is a somewhat standard formalism for specifying the “topmost” lexical level in a morphology

Compiles into a transducer with *foma*

Suited for concatenative morphologies

Can be adapted to non-concatenative phenomena through different maneuvers (discussed later...)

The role of lexc



A very simple *lexc* example

```
LEXICON Root

cat Suff;
dog Suff;
mouse Suff;
horse Suff;

LEXICON Suff
s #;
  #;
```

Compiling lexc files

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

The lexc “lexicons”

Each *lexc* file consists of arbitrarily named sublexicons

Words are constructed by consulting LEXICONs, selecting a morpheme, and continuing to the next specified lexicon:

```
LEXICON Root
cat Suff;
...
```

The *Root* LEXICON contains the morpheme “cat” which, if chosen, leads to the LEXICON named “Suff”

The **Root** LEXICON is the start LEXICON

The **#** LEXICON is where word construction ends

More lexc...

"Morpheme" entries can be empty:

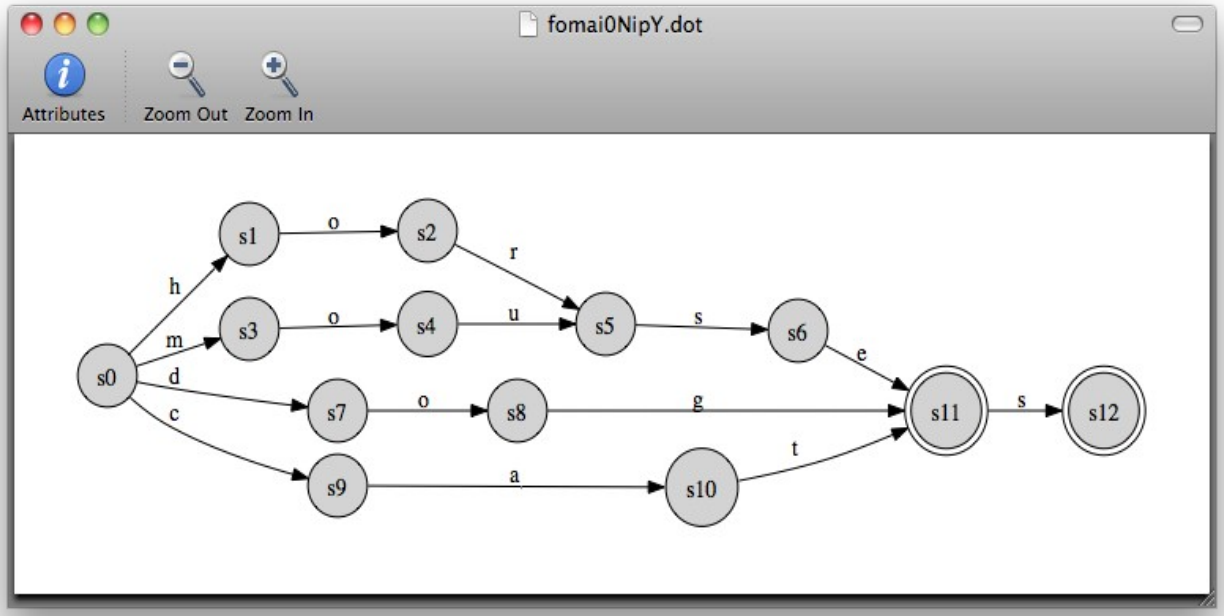
```
LEXICON Suff
s  #;
   #;
```

From LEXICON *Suff*, we can choose either "s" and go to end-of-word, or the "empty string" and go to end-of-word

This makes the suffix (optional), and we can construct both "cat" and "cats"

lexc vs. regular expressions

```
LEXICON Root

cat Suff;
dog Suff;
mouse Suff;
horse Suff;

LEXICON Suff
s #;
  #;

Or:

define Lexicon [c a t|d o g|m o u s e|h o r s e] (s);
```



lexc vs. regular expressions

```
foma[0]: read lexc simplelexc.lexc
Root...4, Suff...2
Building lexicon...Determinizing...Minimizing...Done!
575 bytes. 13 states, 15 arcs, 8 paths.
foma[1]: regex [c a t|d o g|m o u s e|h o r s e] (s);
575 bytes. 13 states, 15 arcs, 8 paths.
foma[2]: test equivalent
1 (1 = TRUE, 0 = FALSE)
```

lexc vs. regular expressions

lexc enforces a “cleaner” design for concatenative morphologies

Compilation time is vastly shorter for large lexicons with *lexc*

The morphotactic combinatorics are more legible

Allows for choice of tools on the level of phonological alternations (*lexc*+two level rules or *lexc*+sequential rewrite rules or ...)

A Spanish lexc-grammar

- As a running example, let's look at a simple Spanish noun grammar with a lexc-part, and a replacement rule part
- We'll only do pluralization of nouns
- Nouns: singular (gato) and plural (gatos)
- There are very few orthographic rules that deal with pluralization, and we'll simplify them, somewhat

Preview of Spanish grammar

Our end goal is to construct a transducer that behaves as follows for analysis/generation:

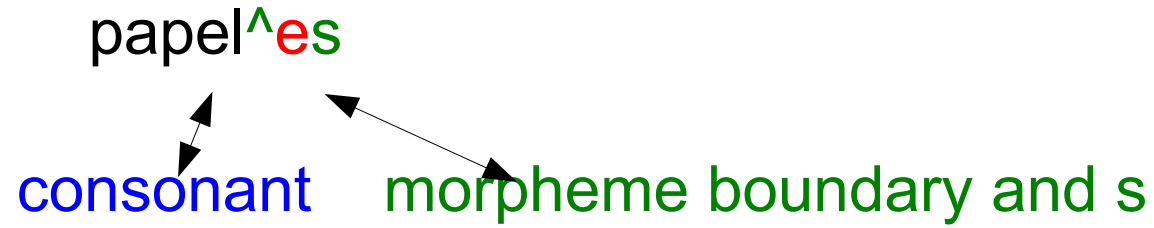
```
foma[1]: up
apply up> amigos
amigo+N+Pl
apply up> peces
pez+N+Pl
apply up>

apply down> nación+N+Pl
naciones
apply down> bebé+N+Pl
bebés
apply down>
```

Facts to be modeled

- Spanish plurals are formed simply by adding -s to the noun stem: (gato → gatos)
- But we need to add e before -s if the stem (a) ends in a consonant (papel → papeles) (b) y (rey → reyes) (c) or a stressed vowel other than é (tisú → tisúes)
- We also need to account for the orthographic change from z to c (vez → veces)
- Stressed vowels in the final stem of the syllable also need to be destressed if pluralized (camión → camiones), but otherwise not (lápiz → lápices) [note the interaction of two generalizations in this example]

Facts to be modeled

- Subsequently, we have a replacement rule that inserts an e in the appropriate environment:



- Preview: we define a rewriting transducer:

```
define AddE [..] -> e || [C | y | [Vacc - é] ] %^ _ s ;
```

The lexc-level

p a p e l +N +Pl (lexc input)
p a p e l ^ s (lexc output)

The Spanish lexc-file

```
Multichar_Symbols +N +Sg +Pl

LEXICON Root
alumno      NINFL;
amigo       NINFL;
bebé        NINFL;
chica       NINFL;
...


LEXICON NINFL
+N+Sg:0  #;
+N+Pl:^s #;
```

The Spanish lexc-file

Points to observe:

Multicharacter symbols must be declared in the beginning:

```
Multichar_Symbols +N +Sg +Pl
```

The Spanish lexc-file

Points to observe:

We have used string pairs in the lexicons:

```
+N+Pl:^s   #;
```

We want the lexc-transducer to translate:

```
pez+N+Pl
pez^s
```

(Here ^ is an abstract symbol that represents a morpheme boundary)

Using lexc-files in foma

- As we saw, we can compile a lexc-file with the command: read lexc <filename>

```
foma[0]: read lexc spanish.lexc
Root...16, NINFL...2
Building lexicon...Determinizing...Minimizing...Done!
2.1 kB. 62 states, 77 arcs, 32 paths.
foma[1]:
```

- The compiled FST is now on top of the stack, and we can name it and use it in regular expressions:

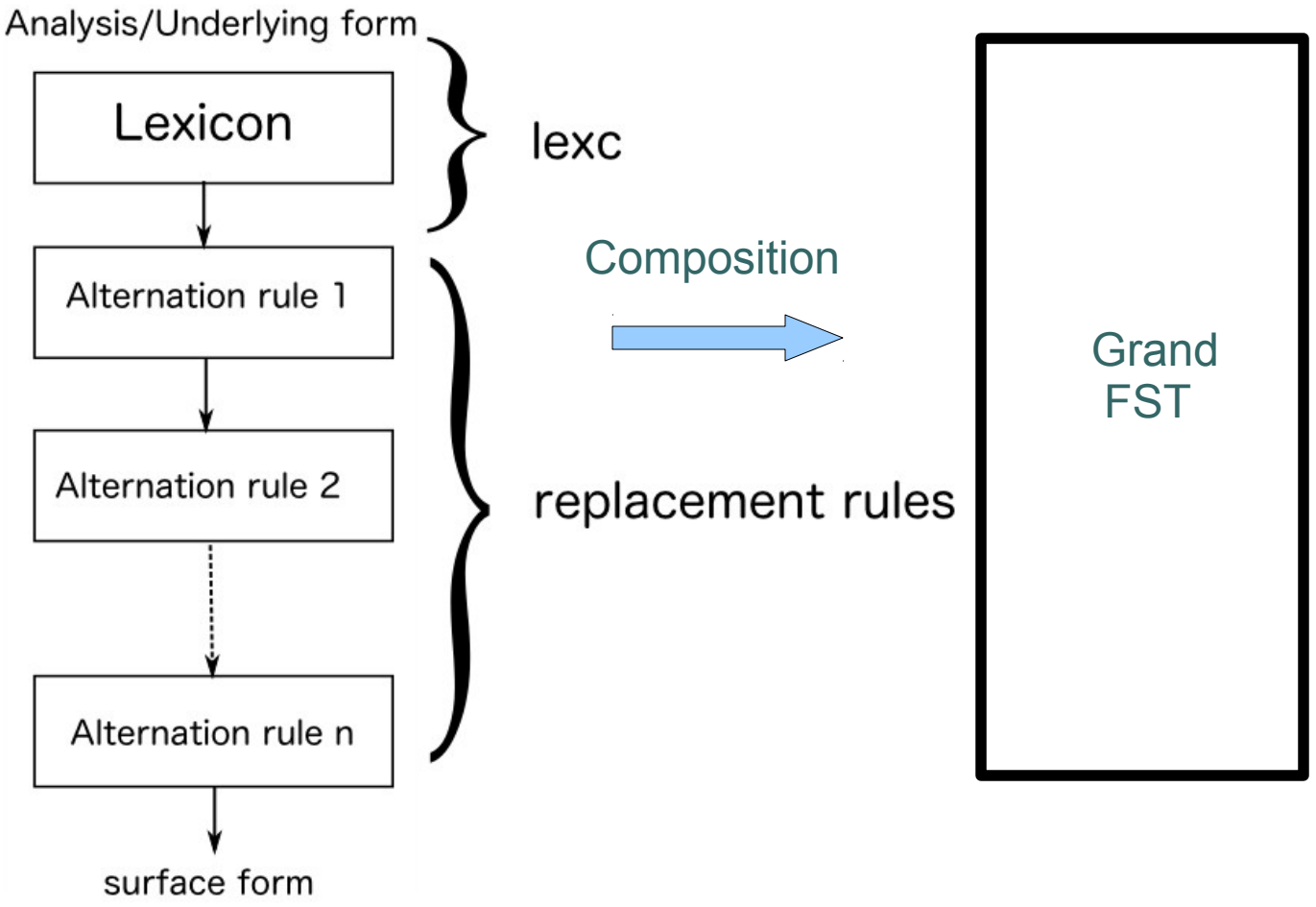
```
foma[1]: define Lexicon;
defined Lexicon: 2.1 kB. 62 states, 77 arcs, 32 paths.
foma[0]: [demo]
```

Computational Morphology: ***Rules & putting it all together***

Rules: Overview

Designing a rewrite-grammar
Composing the lexicon with the rules
Compiling & testing a complete grammar
A few examples

The Big Picture (again)



Running Spanish example

- We created a lexc-grammar that takes us from analyses to intermediate forms:

```
l á p i z +N +Pl
l á p i z ^   s
```

- The task now is to create the replacement rule transducers to be composed with the lexc-transducer, yielding correct surface forms:

```
l á p i z    +N  +Pl   (lexc upper)
l á p i z    ^   s     (lexc lower)
l á p i z    ^ e s     (after “add e” rule)
...
l á p i c      e s     (after last rule)
```

The facts to be modeled II

(1) Z-Rule: change z to c before plural ^ s

```
p e z +N +Pl    (lexc upper)
p e z ^  s      (lexc lower)
...
p e c ^  s      (after z-rule)
...
```

The rule can be defined as:

```
define ZRule z -> c || _ %^ s;
```

The facts to be modeled II

(2) e-insertion: stems ending in consonants, y, and stressed vowels (but not é), add e

```
p a p e l   +N   +Pl   (lexc upper)
p a p e l   ^    s     (lexc lower)
...
p a p e l   ^  e s     (after e-insertion)
...
```

The rule can be defined as:

```
define AddE [..] -> e || [C|y|[Vacc - é]] %^ _ s ;
```

The facts to be modeled II

(3) Remove stem-last-syllable accents if plural is -es:

```
n a c i ó n   +N   +Pl      (lexc upper)
n a c i ó n   ^    s        (lexc lower)
n a c i ó n   ^  e s        (after e-insertion)
n a c i o n   ^  e s        (after accent rem.)
...
```

The rule can be defined as:

```
define RemoveAccent
        á -> a , é -> e , í -> i, ó -> o, ú -> u
                        || _ V* C+ %^ e s;
```

The facts to be modeled II

(4) After we're done with the alternations, we remove the boundary markers:

```
p e z       +N  +Pl         (lexc upper)
p e z       ^    s          (lexc lower)
...
p e c          e s          (after Cleanup)
```

The rule can be defined as:

```
define Cleanup %^ -> 0;
```

Putting the grammar together

```
...

read lexc spanish.lexc
define Lexicon;

define Rules ZRule .o. AddE .o. RemoveAccent .o.
   Cleanup;

regex Lexicon .o. Rules;
```

Compiling

```
foma[0]: source spanish.script
Opening file 'spanish.script'.
defined V: 705 bytes. 2 states, 12 arcs, 12 paths.
defined Vacc: 410 bytes. 2 states, 5 arcs, 5 paths.
defined C: 1.0 kB. 4 states, 22 arcs, 22 paths.
defined ZRule: 674 bytes. 5 states, 19 arcs, Cyclic.
defined AddE: 2.3 kB. 4 states, 85 arcs, Cyclic.
defined RemoveAccent: 4.9 kB. 9 states, 244 arcs, Cyclic.
defined Cleanup: 324 bytes. 1 states, 2 arcs, Cyclic.
Root...16, NINFL...2
Building lexicon...Determinizing...Minimizing...Done!
2.1 kB. 62 states, 77 arcs, 32 paths.
defined Lexicon: 2.1 kB. 62 states, 77 arcs, 32 paths.
defined Rules: 7.8 kB. 17 states, 428 arcs, Cyclic.
2.6 kB. 70 states, 89 arcs, 32 paths.
```

Let's test the grammar!

Testing...debugging...

```
tisú
tisúes
tapiz
tapices
rey
reyes
presidente
presidentes
papel
papeles
pez
peces
nación
naciones
lápiz
lápices
...
```

Points...

Rule order

Simple Spanish (pluralization) rules

```
# examples: papel+s:papeles; pez+s:peces
# sequential (ordered)
(1)     z -> c || _ "+" s .#. ;
(2) [..] -> e || Cons "+" _ s .#. ;
```

Compare:

```
pez+s (1) → pec+s   (2) → pec+es
pez+s (2) → pez+es  (1) → *pez+es
```

Computational Morphology: ***Irregular forms and advanced morphotactics***

Simp. de Comunicación Social, 2011. Santiago de Cuba

Irregular forms: overview

Irregular forms
Parallel “irregular” and regular forms
Long-distance dependencies in lexc
Agreement between separated morphemes

Errors in running example

```
foma[0]: source english.foma
...
foma[1]: down
apply down> make+V+PastPart
maked
apply down>
```

Such irregularities are of course rampant, and we need to handle them

Almost every language has suppletive forms for high-frequency verbs (be, was, were), adjectives (good, better, best), etc. and these need to be handled systematically

Adding exceptions in *lexc*

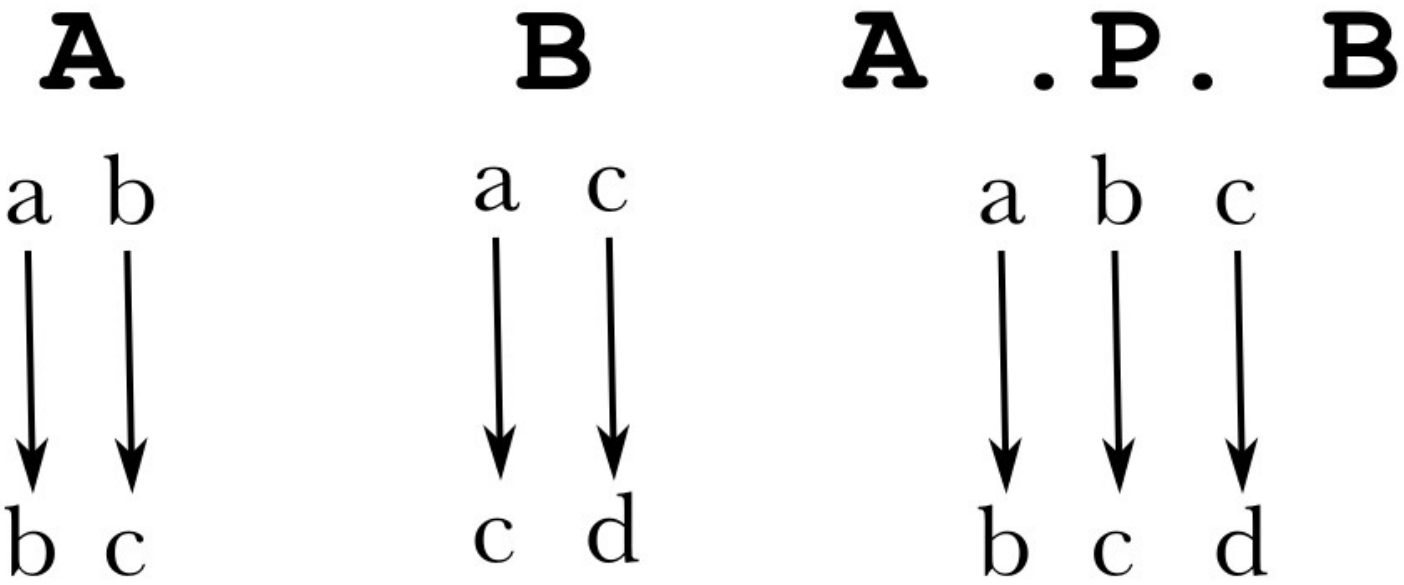
```
LEXICON Verb

fox   Vinf;
beg   Vinf;
make+V+PastPart:made  #;  !Bypass Vinf
make+V                #;
...
watch Vinf;
try   Vinf;
panic Vinf;
```

A separate “exceptions” grammar

```
foma[0]: define Exceptions [m a k e "+V" "+PastPart"]:[m a d e];
defined Exceptions: 370 bytes. 7 states, 6 arcs, 1 path.
foma[0]: regex Exceptions;
370 bytes. 7 states, 6 arcs, 1 path.
foma[1]: down
apply down> make+V+PastPart
made
apply down>
foma[1]:
```

Combining with priority union

Priority union operation for overriding regular with irregular forms



In this case, we need to calculate the transducer [Exceptions .P. Grammar]

Combining with priority union

```
foma[1]: define Grammar;
defined Grammar: 1.8 kB. 47 states, 70 arcs, 42 paths.
foma[0]: define Exceptions [m a k e "+V" "+PastPart"]:[m a d e];
defined Exceptions: 370 bytes. 7 states, 6 arcs, 1 path.
foma[0]: regex [Exceptions .P. Grammar];
1.9 kB. 52 states, 77 arcs, 42 paths.
foma[1]: down
apply down> make+V+PastPart
made
apply down>
foma[1]:
```

Alternate forms

Alternate forms are also possible

English: cactus+N+Pl → cactuses, cacti

In this case we can create an alternate forms grammar, and calculate the union with the regular grammar

```
foma[0]: define ParallelForms [c a c t u s "+N" "+Pl"]:[c a c t i];
redefined ParallelForms: 410 bytes. 9 states, 8 arcs, 1 path.
foma[1]: regex ParallelForms | Grammar ;
...
```

Long-distance dependencies

Non-sequential morphotactics:

i.e. *en+joy+able*

Overgeneration in the lexicon

Constraining the co-occurrence of morphemes

Strategy: compose a filter before or after lexical level

lexical information (tag sequence)

morphemes (morpheme sequence)

Usually of the format ~$[PATTERN];

“The language that does not contain PATTERN”

Example from Basque: no double causatives (many agreement patterns work in similar manner)

Long-distance dependencies

Example of rule (for Basque)

```
# avoiding overgeneration from the lexicon
# causative prefix and suffix, but not both
#     RIGHT: bait+du, du+Elako
#     WRONG: bait+du+Elako
# LEXICAL LEVEL: [Caus]+edun[V][P][3P]+[Caus]
# INTERM. LEVEL:  bait+  du           +Elako
# SURFACE LEVEL:  bait   du            elako
# morphological inf. level
define NOTWO  ~$[ "[Caus]" ?+ "[Caus]" ];
define MORPHOFIL NOTWO .o. LEX .o. RULES ;
# intermediate level
define NOTWO2  ~$[ b a i t "+" ?+ "+" E l a k o];
define MORPHOFIL2 LEX .o. NOTWO2 .o. RULES;
```

Long-distance dependencies

```
foma[1]: regex MORPHO;
10.5 kB. 390 states, 589 arcs, Cyclic.
foma[2]: up baituelako
[Kaus]+edun[ADL][A1][P3]+[Kaus]
foma[2]: regex MORPHOFIL2;
11.6 kB. 450 states, 661 arcs, Cyclic.
foma[3]: up baituelako
???
foma[3]: up baitu
[Kaus]+edun[ADL][A1][P3]
foma[3]: up duelako
edun[ADL][A1][P3]+[Kaus]
```

Computational Morphology: ***Simple applications***

Applications: overview

- Spell checking
- Spelling correction
  - Competence errors
  - OCR errors
- Morphological guessers

Spell checking

A morphological analyzer transducer contains on its lower side, a grammar for the legimate word-forms of the language

We can extract this part with the .l-operator (creating an automaton that only accepts English words):

```
...
defined Grammar: 1.8 kB. 47 states, 70 arcs, 42 paths.
foma[0]: regex Grammar.l;
1.5 kB. 37 states, 52 arcs, 28 paths.
foma[1]: random-words
cat
watch
watching
making
```

Spelling correction

We can re-use the word automaton for creating a rudimentary spelling corrector

An example from a larger English grammar:

Extract the set of words

Compose this set with a transducers that makes a limited number of changes

Run the resulting transducer in the upward direction

A simple corrector

```
foma[0]: load defined engwords.foma
Loading definitions from engwords.foma.
foma[0]: print defined
Words 528.1 kB. 16151 states, 33767 arcs, 42404 paths.
foma[0]: define C1 [?* [?:0|0:?|?:?-?] ?*];
defined C1: 294 bytes. 2 states, 5 arcs, Cyclic.
foma[0]: regex Words .o. C1;
21.6 MB. 32302 states, 1415320 arcs, Cyclic.
```

Testing the corrector

```
foma[1]: up
apply up> caxt
cast
cart
cat
apply up> gdog
dog
apply up> twinx
twins
twine
twin
apply up>
```

MED built in foma

```
foma[0]: regex Words;
528.1 kB. 16151 states, 33767 arcs, 42404 paths.
foma[1]: med

apply med> caxt
Calculating heuristic [h]
Using Levenshtein distance.
cart
caxt
Cost[f]: 1
ca*t
caxt
Cost[f]: 1
cast
caxt
Cost[f]: 1
```

Competence errors

MED for Basque

Phonologically similar segments are interchanged at lower cost (e.g. h/0 x/s, ...)

typo.matrix

```
Insert 2
Substitute 2
Delete 2
Cost 1
:h h: x:j j:x t: :t p:b b:p t:d d:t p:f f:p g:j j:g u:o o:u i:l
l:i c:k k:c z:c c:z m:n n:m b:v v:b s:z z:s x:z z:x s:x x:s
```

script_med_eu

```
regex MORPHO.l ;              #extract lower side
read cmatrix typo.matrix      #attach matrix
med ettxea                    #test words...
med lehioa
```

Competence errors

```
Reading confusion matrix from
file 'typo.matrix'
Calculating heuristic [h]
Using confusion matrix.
```

```
e*txea
ettxea
Cost[f]: 1
```

```
et*xea
ettxea
Cost[f]: 1
```

```
e*txean
ettxea*
Cost[f]: 3
```

```
le*ihoa
lehi*oa
Cost[f]: 2
```

```
le*ihoak
lehi*oa*
Cost[f]: 4
```

```
le*iho*
lehi*oa
Cost[f]: 4
```

OCR errors

```
# test with parallel and sequential rules
define PARAL0 [ m (->) r n , n (->) r i ] ;
define SEQ0 [ m (->) r n .o. n (->) r i ] ;

# commons OCR errors
# parallel rules: only one error in each word

define PARAL  [ c (->) e , e (->) c , c (->) o , o (->) c
   , l (->) i , i (->) l , l (->) 1 , 1 (->) l , r n (->)
   m , m (->) r n , r i (->) n , n (->) r i , c l (->)
   d , d (->) c l , o (->) "0" , "0" (->) o , r t (->)
   n , r t (->) i t , r m (->) n n , r r i (->) m , m (-
   >) r r i , n i (->) m , l i (->) h , u (->) i r ] ;
```

Morphological guessers

For many applications we need to be quite flexible in parsing OOV items (word forms)

Simple strategy: reuse rule grammar, use an “open” lexicon with *any* phonologically possible word as a stem

The more accurate the set of phonologically possible stems, the better (cf. run+V→running, eat+V→eating, *sibl+V→sibling)

We can simply keep the closed-lexicon morphology and guesser separate, or combine the two

An English guesser

```
[englishguesser.foma]
define Stem [C^<3 V C^<3]+;
define GuessLexicon Stem [
                    ["+N" "+Pl"]:["^" s]          |
                    ["+N" "+Sg"]:0                |
                    ["+V"]:0                      |
                    ["+V" "+3P" "+Sg"]:["^" s]    |
                    ["+V" "+PastPart"]:["^" e d ] |
                    ["+V" "+PresPart"]:["^" i n g ]]  ;

foma[1]: up
apply up> sibling
sibling+V
sibling+N+Sg
sible+V+PresPart
sibl+V+PresPart
```

Guessers with priority union

```
foma[1]: regex [Grammar.i .P. Guesser.i].i;
23.8 kB. 82 states, 1475 arcs, Cyclic.
foma[2]: up
apply up> cats
cat+N+Pl
apply up> catting
catt+V+PresPart
catte+V+PresPart
catting+V
catting+N+Sg
apply up> blarks
blark+V+3P+Sg
blark+N+Pl
apply up>
```

Guessers with priority union

```
foma[2]: regex [Grammar.i .P. Guesser.i].i;
23.8 kB. 82 states, 1475 arcs, Cyclic.
foma[3]: up
apply up> cats
cat+N+Pl
apply up> catting
catt+V+PresPart
catte+V+PresPart
catting+V
catting+N+Sg
apply up> blarks
blark+V+3P+Sg
blark+N+Pl
apply up>
```

Computational Morphology: ***Syntax, etc***

Simp. de Comunicación Social, 2011. Santiago de Cuba

Surface syntax. Applications

Identification of entities

Dates, numbers, named entities

Surface syntax (after POS tagging)

Noun phrases, verb phrases...

Translation of dates, numbers...

...

foma

New operators: Longest matching and insertion

@->	substitution of longest matched
...	matched string

```
OriginalString @-> TagBegin ... TagEnd ;
```

Identifying entities

```
define Char [a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z] ;
define Capital [A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z] ;
define NumberElem ["0"|1|2|3|4|5|6|7|8|9] ;
define NumberSymbol ["."|","] ;
define EntiElem Capital [ Capital | Char ]* ;

define EntiStr EntiElem [(" ") EntiElem]* ;
define TagEnti [EntiStr @-> "<ENTI>" ... "</ENTI>"] ;

define NumberStr NumberElem [(NumberSymbol) NumberElem]*;
define TagNumber [NumberStr @-> "<NUMB>" ... "</NUMB>"];

regex TagEnti .o. TagNumber;
```

Identifying entities

Result from foma

```
...
5.1 kB. 9 states, 253 arcs, Cyclic.
April 14, 2010 - Nelson Mandela was honoured
<ENTI>April</ENTI> <NUMB>14</NUMB>, <NUMB>2010</NUMB> -
   <ENTI>Nelson Mandela</ENTI> was honoured
```

English dates

```
...
define Day [(1|2) Number | 3 "0" | 3 1];
define Year Number (Number) (Number) (Number);
define RegDates [WeekDay | Month " " Day (", " Year)];
define DateParser [RegDates @-> "<DATE>" ... "</DATE>"];

defined DateParser: 4.5 kB. 17 states, 238 arcs, Cyclic.
4.5 kB. 17 states, 238 arcs, Cyclic.
April 14, 2010 - Nelson Mandela was honoured
<DATE>April 14, 2010</DATE> - Nelson Mandela was honoured

http://www.stanford.edu/~laurik/fsmbook/examples/DateParser.html
```

Translating numbers to word sequences

[transl_numbers (slightly simplified)]

```
# INPUT: 2,001,634
define Tag1 "," -> M || _ Number^3 .#. ;
define Tag2 "," -> M M || _ Number^3 M ;
define Tag3 Number -> ... C || _ Number^2 [M|.#.] ;
define Tag4 Number -> ... X || _ Number [M|.#.] ;
#2MM0C0X1M6C3X4

define NtoW1 1 X "0"->"ten",1 X 1->"eleven" || _ [M|.#.];
define NtoW2 2->"twenty",3->"thirty"    || _ X ;
define NtoW3 1->"one", 2->"two", 3->"three" || _ [C|M|.#.];

define End1 M M -> " million " ;
define End2 M -> " thousand " ;
define End3 C -> " hundred " ;
define End4 X -> 0 ;
```

Testing the translation rules

```
...
3.6 kB. 27 states, 206 arcs, Cyclic.
2,001,634
2MM0C0X1M6C3X4
34.7 kB. 178 states, 2157 arcs, Cyclic.
two million one thousand six hundred and thirty-four
...
```

Eskerrik Asko! Kiitos!
Muchas Gracias!

Foma can help you, and you can help *foma*:

http://foma.sf.net

We can help you with *foma* (and your language):

Mans Hulden: mans.hulden@helsinki.fi
Iñaki Alegria: i.alegria@ehu.es