Advanced finite-state techniques tutorial

CLT, Göteborgs Universitet, Oct. 1, 2013

Mans Hulden
University of Helsinki

UNIVERSITY OF HELSINKI

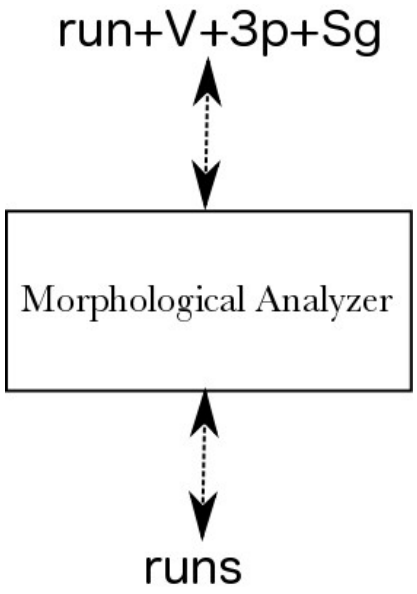
Overview

- Two part-tutorial:
    - First part: unweighted techniques
    - Second part: weighted techniques
- Primary tool in this part: foma finite-state compiler
  http://foma.googlecode.com
- Primary tool for weighted FSMs: *Kleene*
  http://kleene-lang.org
- *(See also OpenFST [http://www.openfst.org] but has less of an interface: Kleene uses OpenFST internally)*

(1) Recap morphological analyzer construction
(2) Advanced techniques in morphologies
(3) Generic tricks (spell checkers from morphologies, etc.)
(4) Weighted automata/transducers

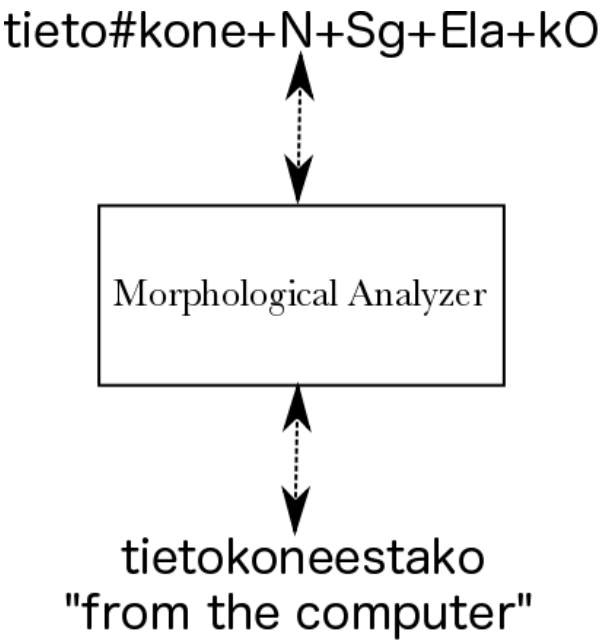
Old hat: Morphological analysis

- English example (simple)



Morphological analysis

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



Real-life example: Basque analyzer

```
foma[0]: load basque-whole-MI.fst
47.5 MB. 2915595 states, 3109378 arcs, Cyclic.
foma[1]: up etxeak
etxe[[Sarrera_etxe--0][KAT_IZE][AZP_ARR][BIZ_-]]+ak[[Sarrera_ak--1][KAT_DEK]
[KAS_ABS][NUM_P][MUG_M][FSL_[FS1_@OBJ][FS2_@PRED][FS3_@SUBJ]]]
etxe[[Sarrera_etxe--0][KAT_IZE][AZP_ARR][BIZ_-]]+ak[[Sarrera_ak--2][KAT_DEK]
[KAS_ERG][NUM_S][MUG_M][FSL_[FS1_@SUBJ]]]
```

etxe[[Sarrera_etxe--0][KAT_IZE][AZP_ARR][BIZ_-]]
(etxe-house, NOUN, COMMON, NOT ANIMATE)

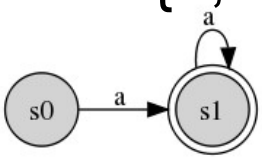
+ak[[Sarrera_ak--1][KAT_DEK][KAS_ABS][NUM_P][MUG_M][FSL_[FS1_@OBJ][FS2_@PRED][FS3_@SUBJ]]]
(+ak (1), DECLEN, ABSOLUT, PLURAL, DETER, SYN_F OBJECT PREDICATE SUBJECT)

etxe[[Sarrera_etxe--0][KAT_IZE][AZP_ARR][BIZ_-]]
(etxe-house, NOUN, COMMON, NOT ANIMATE)

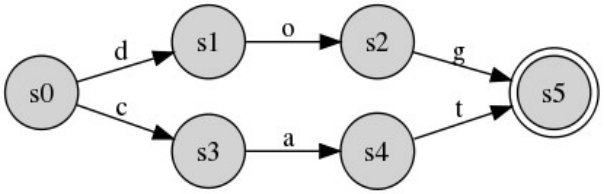
+ak[[Sarrera_ak--2][KAT_DEK][KAS_ERG][NUM_S][MUG_M][FSL_[FS1_@SUBJ]]]
(+ak (2), DECLEN, ERGATIVE, SING, DETERM, SYN_F SUBJECT)

Recap: finite automata

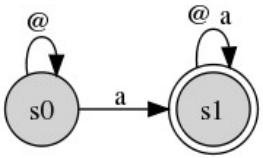
“one or more as”: {a,aa,...}:



the words “cat” and “dog”:



any word that contains at least an a:

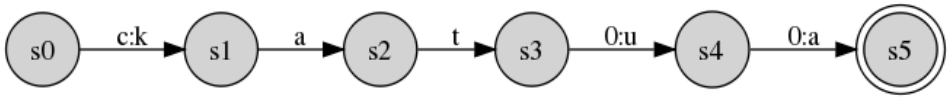


@ = any symbol outside the defined alphabet

Recap: finite transducers

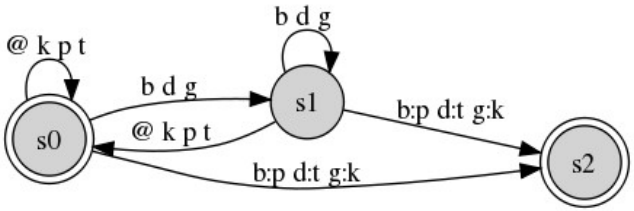
Translates all a-symbols to b and vice versa

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
fox → foxes

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)

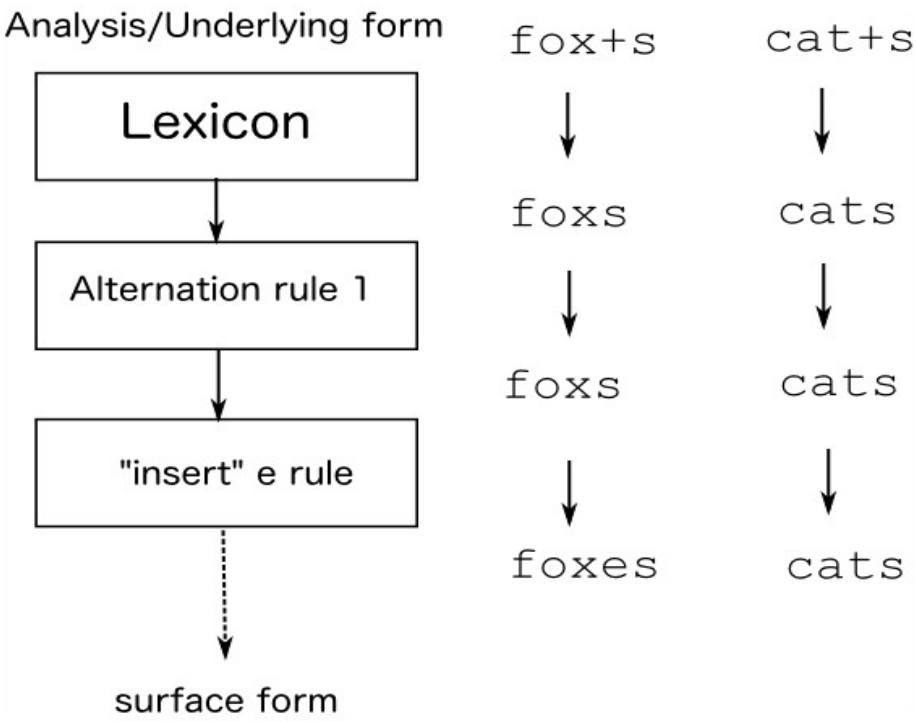
Words are “derived” by

(1) Constructing a morphotactically correct “underlying” form

(2) 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 joined 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.googlecode.com

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

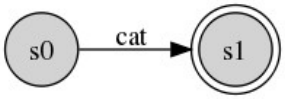
(space)	concatenation
|	union
*	Kleene star
&	Intersection
~	Complement

foma: ordinary symbols

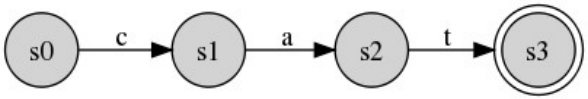
Single-character symbols:
a, b, c, Ω, ب , β, etc.

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

```
foma[0]: regex ?* a ?*;
261 bytes. 2 states, 4 arcs, Cyclic.
foma[1]: net
Sigma: @ a
Size: 1.
Net: 41A7
Flags: deterministic pruned minimized epsilon_free
Arity: 1
Ss0:    @ -> s0, a -> fs1.
fs1: @ -> fs1, a -> fs1.
foma[1]:
```

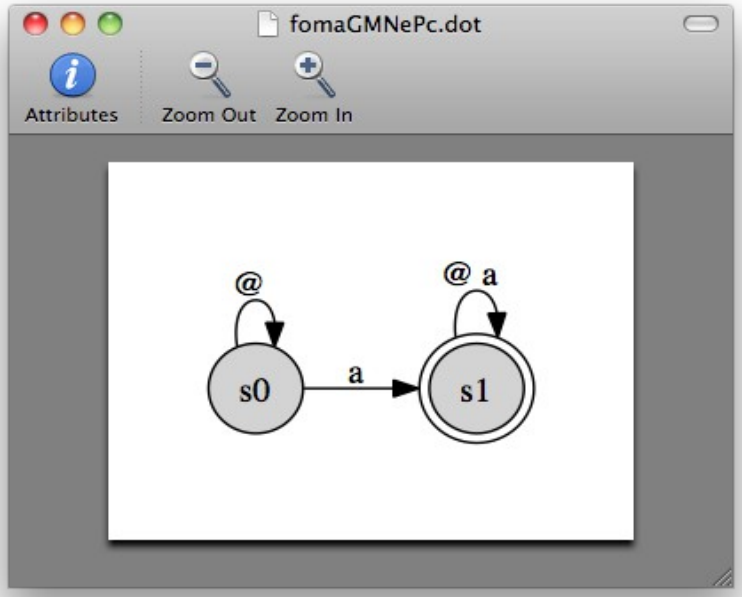
Examining FSMs visually

foma[0]: regex ?* a ?*;

261 bytes. 2 states, 4 arcs, Cyclic.

foma[1]: view

foma[1]:



More about foma

Labeling FSMs: the define command

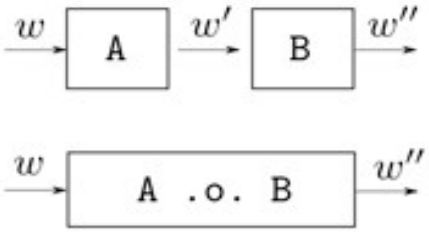
foma[0]: define V [a|e|i|o|u];
defined V: 317 bytes. 2 states, 5 arcs, 5 paths.

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)

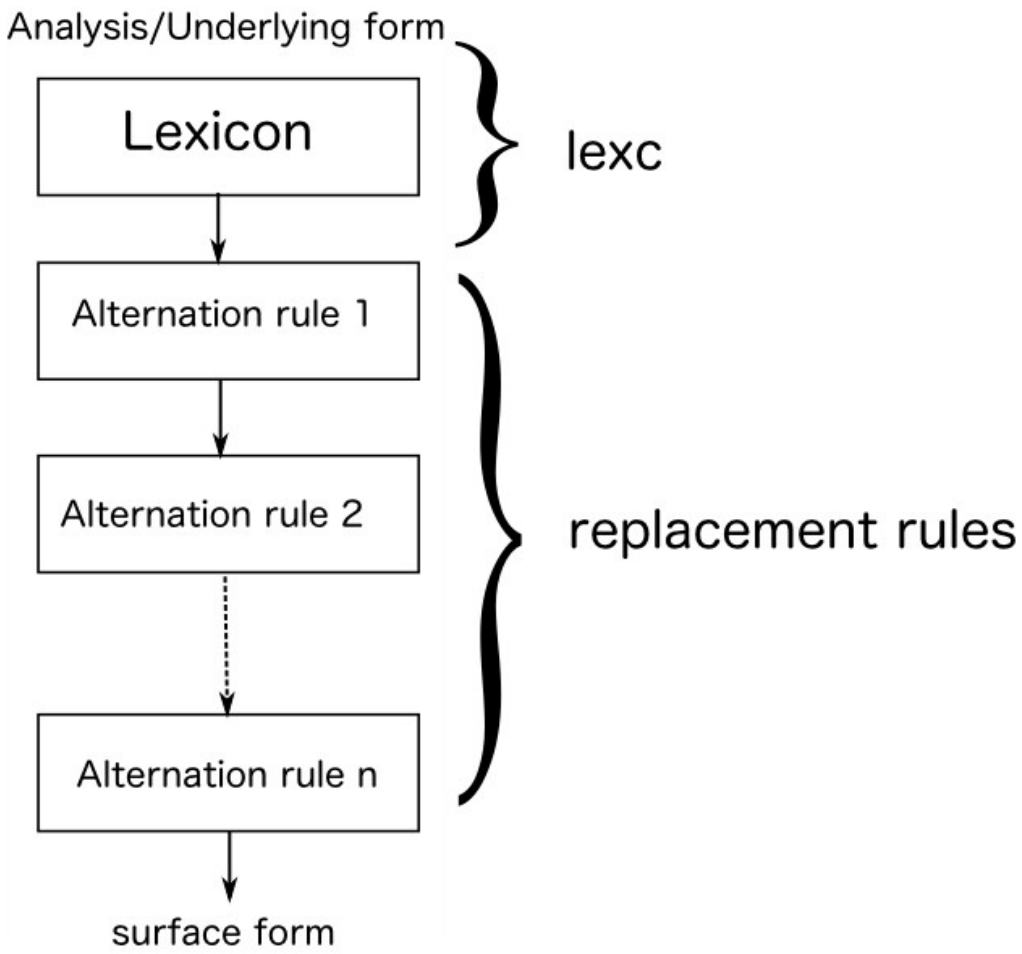
<u>Transducer-related:</u>

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

Overview of lexc

- 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
foma[1]:
```

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...

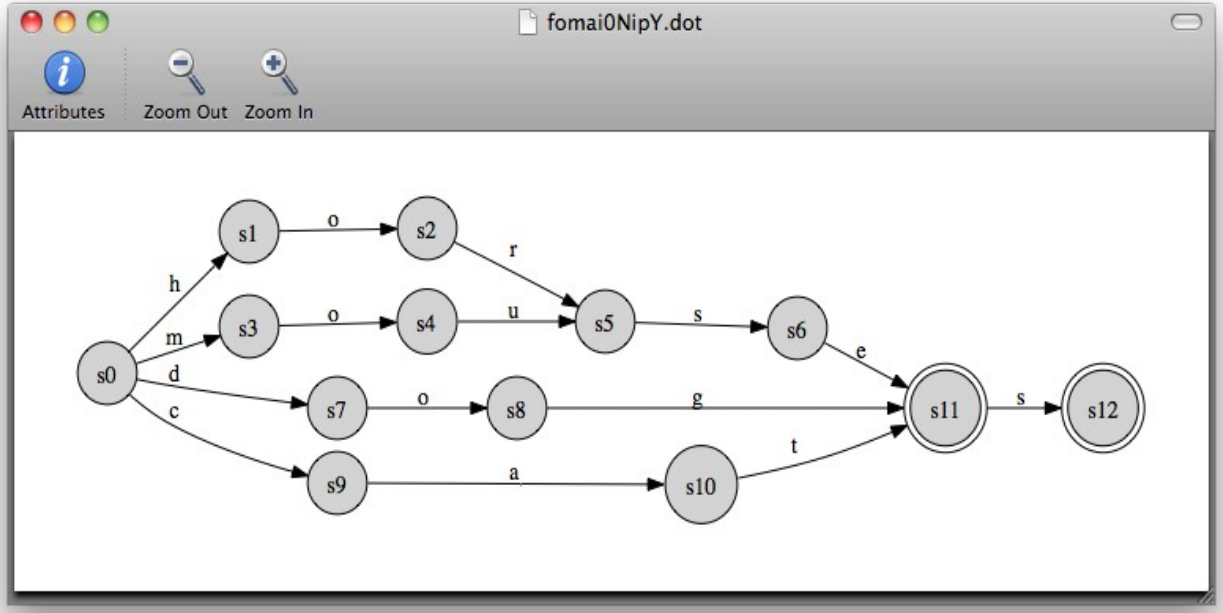
“Morpheme” entries can be empty:

```
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 ...)

An English lexc-grammar

- As a running example, let's look at a simple English grammar with a lexc-part, and a replacement rule part
- We'll focus on some nouns and verbs together with alternation rules
- Nouns: singular (cat) and plural (cats)
- Verbs: infinitive (watch), 3rd person singular (watches), past tense (watched), past participle (watched), and present participle (watching)

Preview of English grammar

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

```
foma[1]: up
apply up> cats
cat+N+Pl
apply up> watches
watch+V+3P+Sg
watch+N+Pl
apply up> trying
try+V+PresPart
apply up>

foma[1]: down
apply down> make+V+PresPart
making
apply down>
```

Facts to be modeled part I

- English plurals are formed simply by adding -s to the noun stem: cat → cats
  - But we have an alternation when the pluralizing morpheme -s is added to stems that end in sibilants (orthographically: sh, zh, z, x, s, ch)

    watch → watches, fox → foxes, ash → ashes
- We also have an alternation y~ie for stems that end in y:

  city → cities

- The standard way to handle such alternations is to choose one form for the general case, and handle the rest through rewrite rules.
- We declare that all plurals are of the form stem^s: cat → cat^s, watch → watch^s

Facts to be modeled part I

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

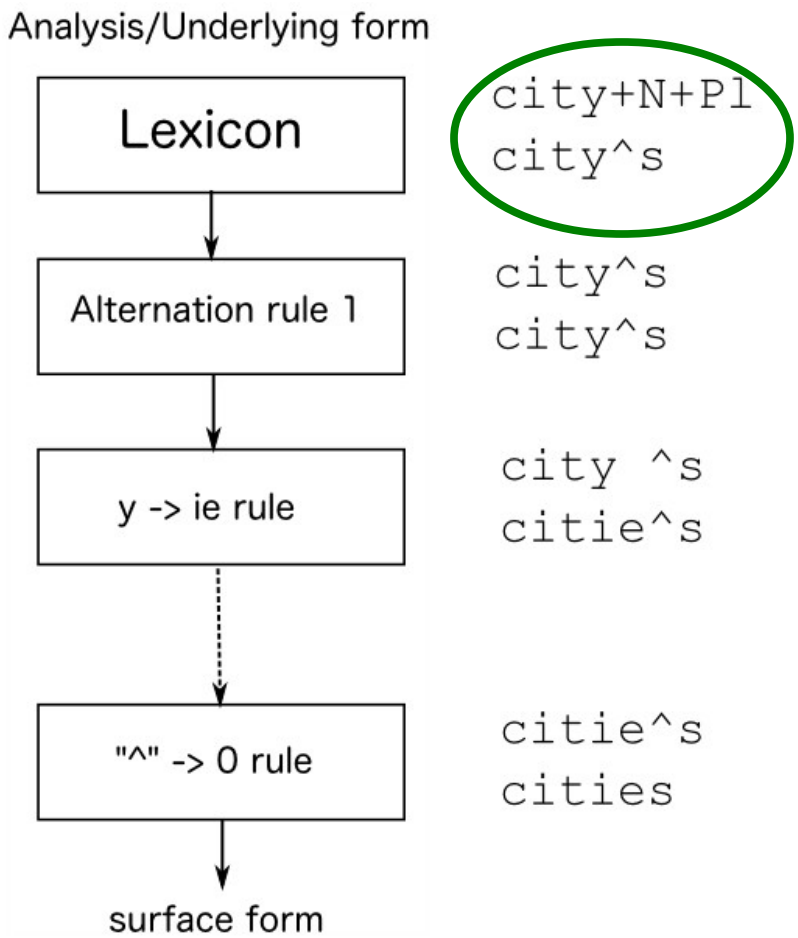
watche^s

sibilant morpheme boundary and s

- Preview: we define a rewriting transducer:

```
define EInsertion [..] -> e || s | z | x | c h | s h _ "^" s ;
```

The lexc-level



English: choosing tags

We'll choose some tags for the analysis strings

Noun: +N
Plural: +Pl
Singular: +Sg

Verb: +V
Third person: +3P
Past tense: +Past
Past participle: +PastPart
Present participle: +PresPart

The English lexc-file

```
Multichar_Symbols +N +V +PastPart +Past +PresPart +3P
+Sg +Pl

LEXICON Root

Noun ;
Verb ;

LEXICON Noun

cat   Ninf;
city  Ninf;
watch Ninf;
try   Ninf;
panic Ninf;
fox   Ninf;
```

```
LEXICON Verb

fox   Vinf;
beg   Vinf;
make  Vinf;
watch Vinf;
try   Vinf;
panic Vinf;
...
```

The English lexc-file

Points to observe:

Multicharacter symbols must be declared in the beginning:

```
Multichar_Symbols +N +V +PastPart +Past +PresPart +3P +Sg
+Pl
```

We have an empty “Root”-lexicon that simply jumps to the Noun lexicon or Verb lexicon with no morphemes:

```
LEXICON Root

Noun ;
Verb ;
```

The English lexc-file part II

```
LEXICON Ninf

+N+Sg:0    #;
+N+Pl:^s   #;   ! ^ is our morpheme boundary

LEXICON Vinf

+V:0                #;
+V+3P+Sg:^s         #;
+V+Past:^ed         #;
+V+PastPart:^ed     #;
+V+PresPart:^ing    #;
```

The English lexc-file

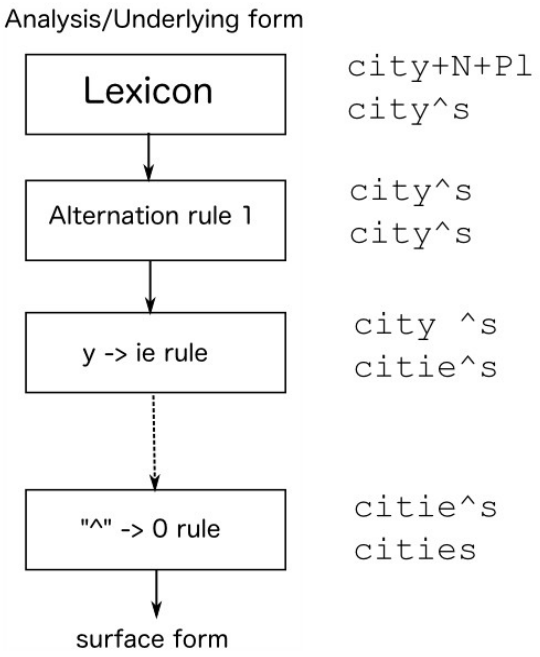
Points to observe:

We have used string pairs in the lexicons:

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

We want the lexc-transducer to translate:

```
cat+N+Pl
cat^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 english.lexc
Root...2, Noun...6, Verb...6, Ninf...2, Vinf...5
Building lexicon...Determinizing...Minimizing...Done!
1.3 kB. 32 states, 46 arcs, 42 paths.
foma[1]:
```

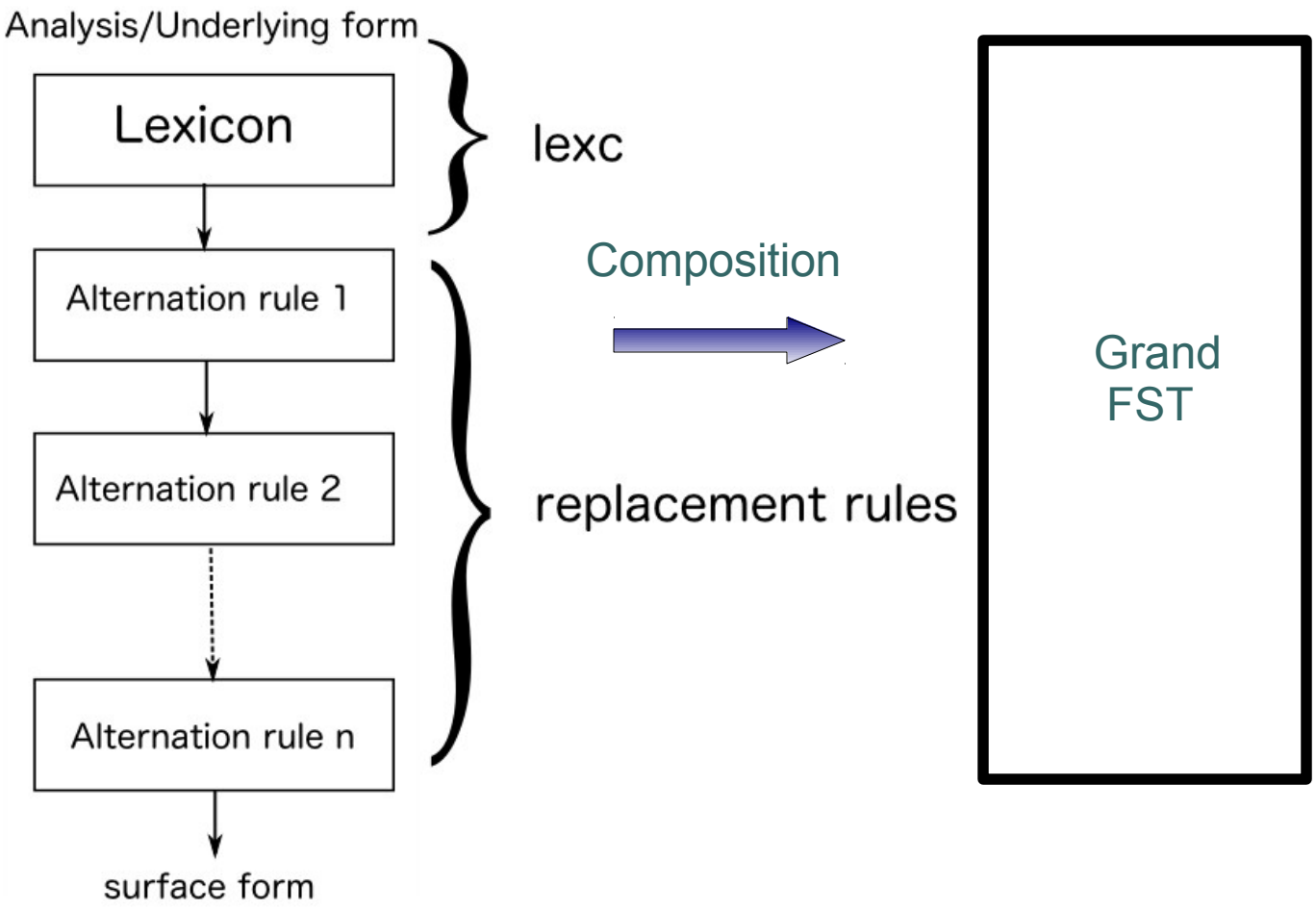
- The compiled FST is now on top of the stack, and we can name it and use it in regular expressions:

```
foma[1]: define Lexicon;
defined Lexicon: 1.3 kB. 32 states, 46 arcs, 42
paths.
foma[0]: [demo]
```

Overview

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

The Big Picture (again)



Running English example

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

```
c i t y +N +Pl
c i t y ^   s
```

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

```
c i t y   +N  +Pl   (lexc upper)
c i t y   ^     s   (lexc lower)
c i t i e ^     s   (after y -> i e rule)
...
c i t i e s         (after nth rule)
```

The facts to be modeled II

(1) E-deletion: silent e dropped before -ing and -ed (make/making)

```
m a k e +V +PresPart    (lexc upper)
m a k e ^ i n g         (lexc lower)
...
m a k 0 ^ i n g         (after E-deletion)
...
```

The rule can be defined as:

```
define EDeletion e -> 0 || _ "^" [ i n g | e d ]
;
```

Let's test the rule *separately [in foma]*:

The facts to be modeled II

(2) K-insertion: verbs ending with vowel-c add -k at end of stem when succeeded by -ed/-ing

```
p a n i c   +V +PresPart   (lexc upper)
p a n i c   ^ i n g        (lexc lower)
...
p a n i c k ^ i n g        (after K-insertion)
...
```

The rule can be defined as:

```
define V [a | e | i | o | u ];
define KInsertion [..] -> k ||  V c _ "^" [e d|i n
g];
```

The facts to be modeled II

(3) E-insertion:

```
f o x   +N +Pl      (lexc upper)
f o x   ^  s        (lexc lower)
...
f o x e ^ s         (after E-insertion)
...
```

The rule can be defined as:*

```
define EInsertion [..] -> e || [s|z|x|c h|s h] _ "^"
s ;
```

*This is not foolproof: consider arch → arches vs. monarch → monarchs

The facts to be modeled II

(4) Consonant doubling: 1-letter consonant doubled before -ing/-ed

```
b e g   +V  +PresPart (lexc upper)
b e g   ^   i n g     (lexc lower)
...
b e g g ^   i n g     (after C-doubling)
...
```

The rule can be defined (for g) as:

```
define V [a | e | i | o | u ];
define ConsonantDoubling g -> g g || V _ "^" i n g ;
```

The facts to be modeled II

(5) Y-replacement: y changes to ie before -s, and i before -ed

```
t r y       +N  +Pl        (lexc upper)
t r y       ^    s         (lexc lower)
...
t r i e     ^    s         (after Y-replacement)
...
```

The rule can be defined as:

```
define YReplacement y -> i e || _ "^" s    ,,
                    y -> i   || _ "^" e d ;
```

The facts to be modeled II

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

```
t r y       +N  +Pl        (lexc upper)
t r y       ^    s         (lexc lower)
...
t r i e          s         (after Cleanup)
...
```

The rule can be defined as:

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

Putting the grammar together

```
...

read lexc english.lexc
define Lexicon;

regex Lexicon .o.
      ConsonantDoubling .o.
      EDeletion .o.
      EInsertion .o.
      YReplacement .o.
      KInsertion .o.
      Cleanup;
```

Compiling

```
foma[0]: source english.foma
Opening file 'english.foma'.
defined V: 317 bytes. 2 states, 5 arcs, 5 paths.
Root...2, Noun...6, Verb...6, Ninf...2, Vinf...5
Building lexicon...Determinizing...Minimizing...Done!
1.3 kB. 32 states, 46 arcs, 42 paths.
defined Lexicon: 1.3 kB. 32 states, 46 arcs, 42 paths.
defined ConsonantDoubling: 1.0 kB. 11 states, 47 arcs,
Cyclic.
defined EDeletion: 1.1 kB. 11 states, 52 arcs, Cyclic.
defined EInsertion: 1000 bytes. 7 states, 43 arcs, Cyclic.
defined YReplacement: 874 bytes. 9 states, 36 arcs, Cyclic.
defined KInsertion: 1.2 kB. 11 states, 59 arcs, Cyclic.
defined Cleanup: 260 bytes. 1 states, 2 arcs, Cyclic.
1.8 kB. 47 states, 70 arcs, 42 paths.
foma[1]:
```

Let's test the grammar!

Testing...debugging...

```
foma[1]: lower-words
cat
cats
city
cities
panic
panics
panic
panics
panicking
panicked
panicked
...
```

Review of lexc+rules

General strategy:

- Create lexc-grammar, load in foma, define:

```
read lexc english.lexc
define Lexicon;
```

- Replacement rules in foma:

```
define Rule1 x -> y ...
```

- Combine with composition:

```
define Grammar Lexicon .o. Rule1 .o. ... .o. RuleN;
regex Grammar;
```

Morphological guessers

Morphological Guesser = a system that provides analyses to words outside the lexicon. Preferably retaining phonological/morphotactic plausibility.

Currently we get:

foma[1]: up
apply up> blarg
???

Maybe we'd want to see something like:

apply up> blarg
GUESS+blarg+N+Sg

Morphological guessers

We can add a fake entry to the lexicon file:

```
!!! english.lexc !!!
Multichar_Symbols +N +V +PastPart +Past +PresPart +3P
+Sg +Pl ^NOUNGUESS

LEXICON Noun

^NOUNGUESS Ninf;  ! Placeholder for unknown nouns
cat   Ninf;
city  Ninf;
fox   Ninf;
```

Morphological guessers

We can modify the grammar to “expand” our guesses to actual words (before phonological/orthographic rules apply)

```
# english.foma #
...
define OrthWord [C* V+ C*]+;
read lexc english-guess.lexc
substitute defined OrthWord for "^NOUNGUESS"
```

Replaces all instances of ^NOUNGUESS with the *language* OrthWord

Morphological guessers

Maybe we'd also like to see a word “tagged” that it's a guess:

```
apply up> blarg
GUESS+blarg+N+Sg
```

```
!!! english.lexc !!!
...
LEXICON Noun

GUESS+^NOUNGUESS:0^NOUNGUESS Ninf; ! Tagged guess
cat   Ninf;
city  Ninf;
fox   Ninf;
```

Morphological guessers

So now we get:

```
apply up> blarg
GUESS+blarg+N+Sg
```

but also:

```
apply up> city
city+N+Sg
GUESS+city+N+Sg
```

Morphological guessers

We can do some grammar trickery:

```
def PriorityUnionOutput(A,B) [A | [B .o. ~A.l]];
# same as A .p. B
```

and now:

```
regex PriorityUnionOutput(
       ~$["GUESS+"] .o. Grammar,
        $["GUESS+"] .o. Grammar
);
```

Morphological guessers

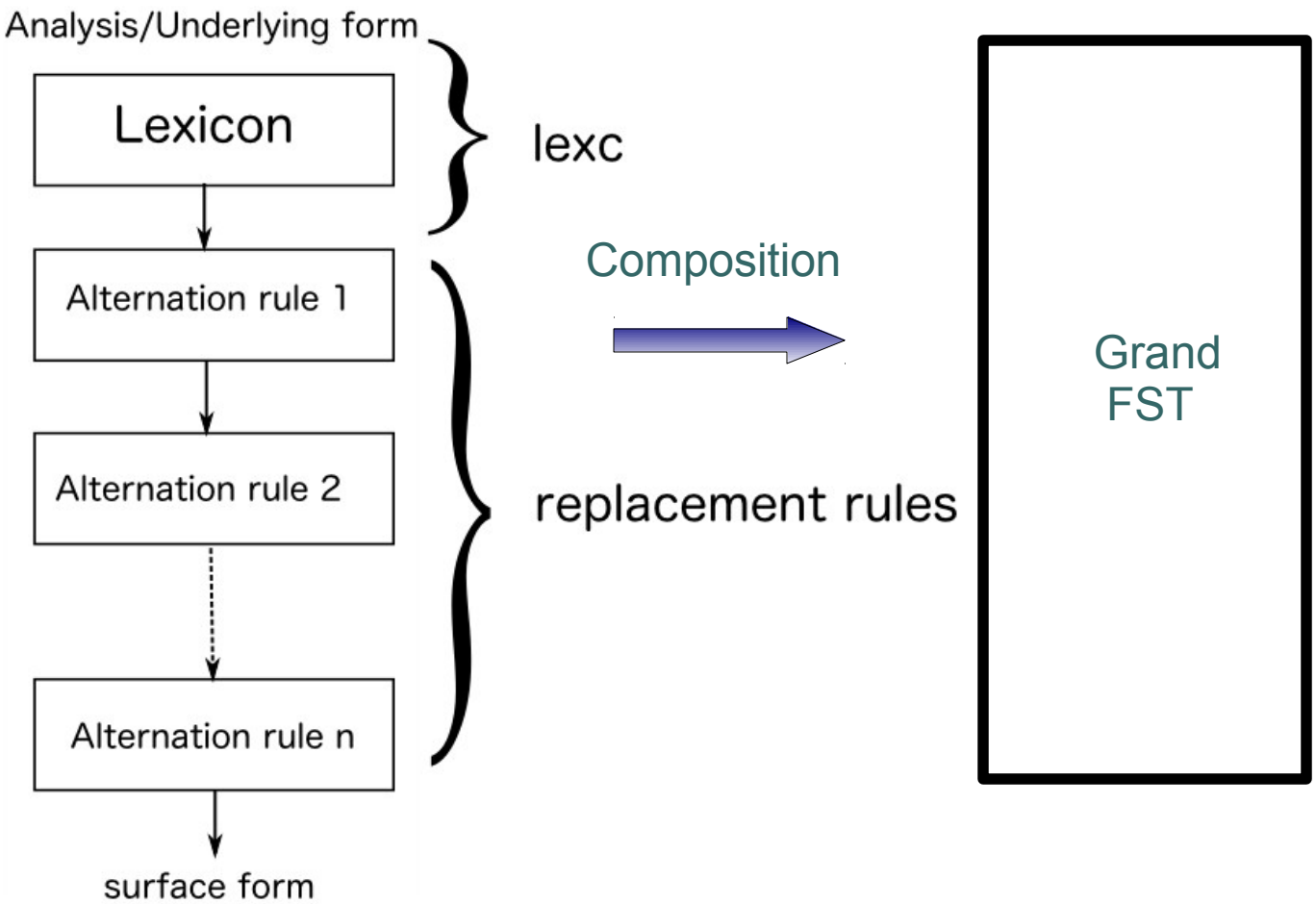
So now we get:

```
apply up> blarg
GUESS+blarg+N+Sg
```

and only:

```
apply up> city
city+N+Sg
```

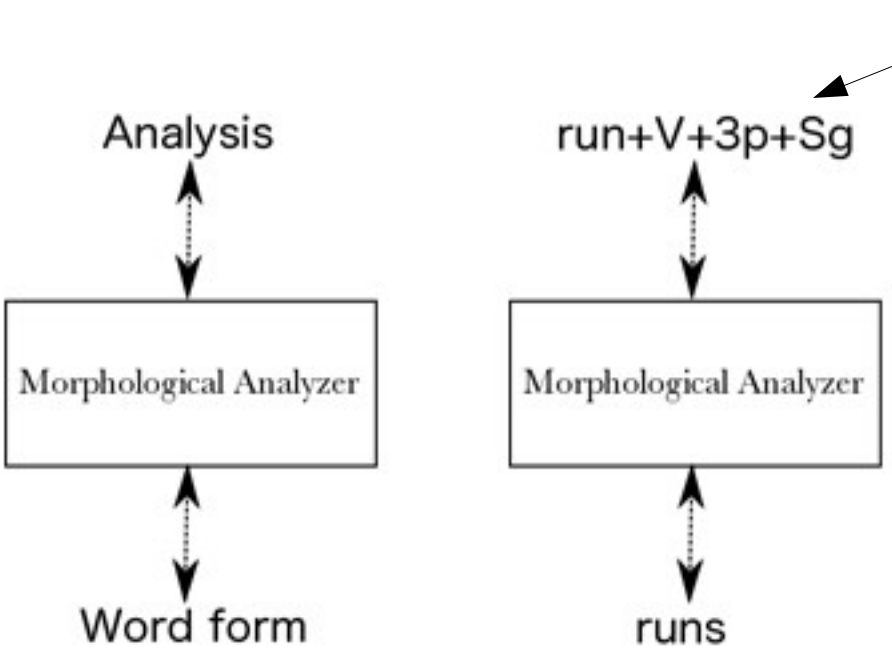
Applications: language model from morphology



For more in-depth explanation, see
https://code.google.com/p/foma/wiki/MorphologicalAnalysisTutorial

Language model from morphology

`Morphology.u` extracts this part



`Morphology.l` extracts this part

Spell checking

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

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

```
$ foma -l english.foma
defined Grammar: 2.2 kB. 47 states, 72 arcs, 42
paths.
foma[0]: regex Grammar.l;
1.5 kB. 37 states, 52 arcs, 28 paths.
foma[1]: random-words

[1] begs
[1] talk
[1] panicking
```

Our toy grammar.

Spelling correction

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

An example from a larger English grammar:

(1) Extract the set of words

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

(3) Run the resulting transducer in the upward direction

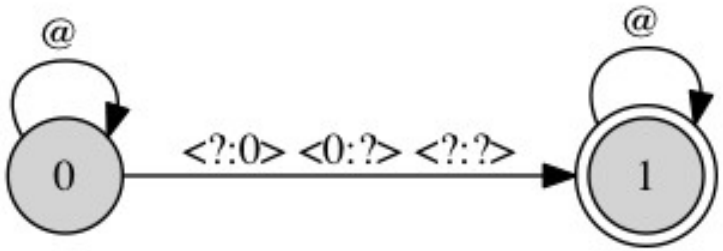
- We can also simply use a word list
  - Example list and compilation into automaton:
  - define W @txt”engwords.txt”;

More detail

- Regular expression trick: define a transducer C1 that makes one change to input words (a deletion, an insertion, or change)
- `define C1 [?* [?:0|0:?|?:?-?] ?*];`

anything (repeat) deletion insertion change anything

equivalent FST:



Simple spelling correction

- Idea: compose this transducer with a lexicon (W):

catx (input word)

C1

`C1 .o. W`

cax, atx, cat, atx, ctx, datx, catc,... (one change)

W

cart, cast, cat,... (one change away + exists in lexicon)

Simple spelling correction

Testing:

```
foma[0]: regex C1 .o. W;
21.6 MB. 32302 states, 1415320 arcs, Cyclic.
foma[1]: down
apply down> caxt
cart
cast
cat
apply down> dogx
dogs
dog
```

Simple spelling correction contd.

- What about more edits?
- MED <= 2:

```
define C2 [?* [?:0|0:?|?:?-?] ?*]^<3;
```

```
foma[4]: regex C2 .o. W;
42.7 MB. 48453 states, 2796873 arcs, Cyclic.
```

Original lexicon size: 528.4kB:
the size of the precomposed corrector grows very quickly...

More spelling correction

- Longer edit distances can be lazily evaluated for each word, at some cost of execution speed.
- Idea:

```
foma[1]: regex {catx} .o. C2 .o. W;
2.6 kB. 50 states, 109 arcs, 93 paths.

foma[2]: words
chat
chats
cot
cots
coat
coats
coax
...
```

Spelling correction

- Or, if we're using foma, we can run minimum-edit distance searches directly against an automaton (with the med/apply command):

```
foma[0]: regex W;
528.4 kB. 16151 states, 33767 arcs, 42404 paths.
foma[1]: med
apply med> grblxal
gradual                 gril--l                 orbital
grblxal                 grblxal                 grblxal
Cost[f]: 3              Cost[f]: 3              Cost[f]: 3
```

Competence errors

- We can also build a more sophisticated error model by specifying weights for different substitutions with *med/apply med*
- 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: s:z z:s x:z z:x s:x x:s
```

script_med_eu

```
regex MORPHO.l ;            # extract lower side of morphology
read cmatrix typo.matrix     # attach matrix
```

Competence errors

```
apply med> leioa
leion
leioa
Cost[f]: 1

leiok
leioa
Cost[f]: 1
```

without confusion matrix

```
apply med> leioa
leihoa
lei-oa
Cost[f]: 1

leion
leioa
Cost[f]: 2
```

with confusion matrix

Manual rules for correction

- We can also specify the “error model” using arbitrary rewrite rules, perhaps in conjunction with edit distance.

```
#transceive, receive, conceive, etc. + teh -> the
define CommonErrors [ c i e v e -> c e i v e ,,
                      t e h -> t h e  || .#. _ .#. ];

define Corrector [CommonErrors .o. W] .P. [C1 .o. W] ;
```

“priority union”: “if CommonPatterns don't produce an output with W, accept `[C1 .o. W]`'s output. The left hand side has “priority” in the union.

Combining manual rules and MED

```
apply down>
recieve
relieve
```

```
apply down>
recieve
receive
relieve
```

```
apply down>
recieve
receive
```

Model: C1 .o. W

(MED 1 search)

[CommonErrors|C1] .o. W

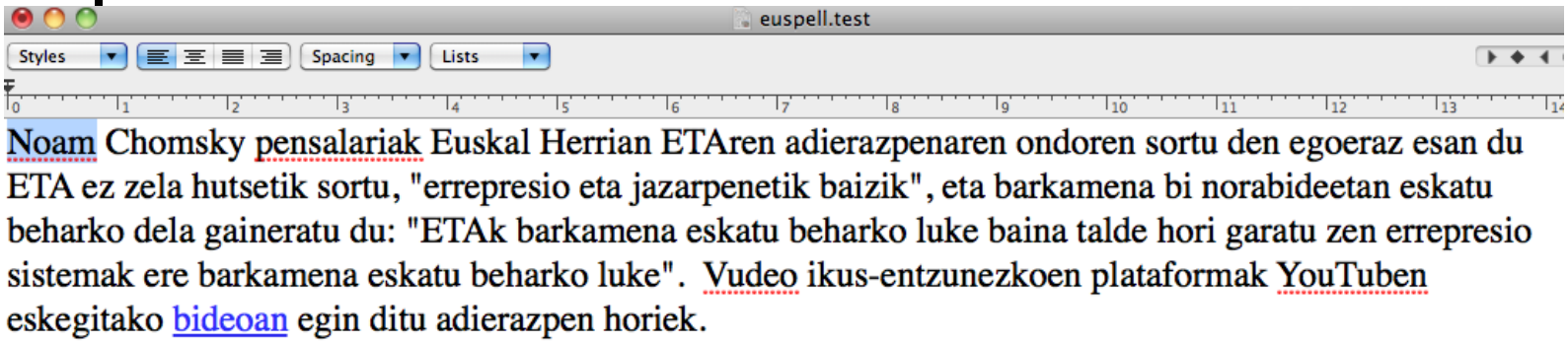
(MED 1 *or* common errors)

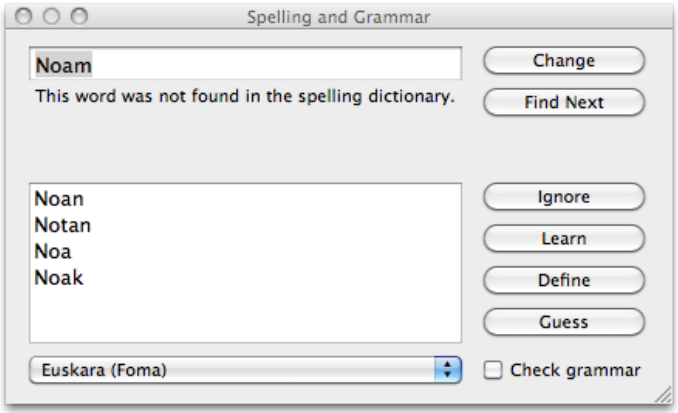
[CommonErrors .o. W] .P. [C1 .o. W]

(Common errors have priority over MED 1)

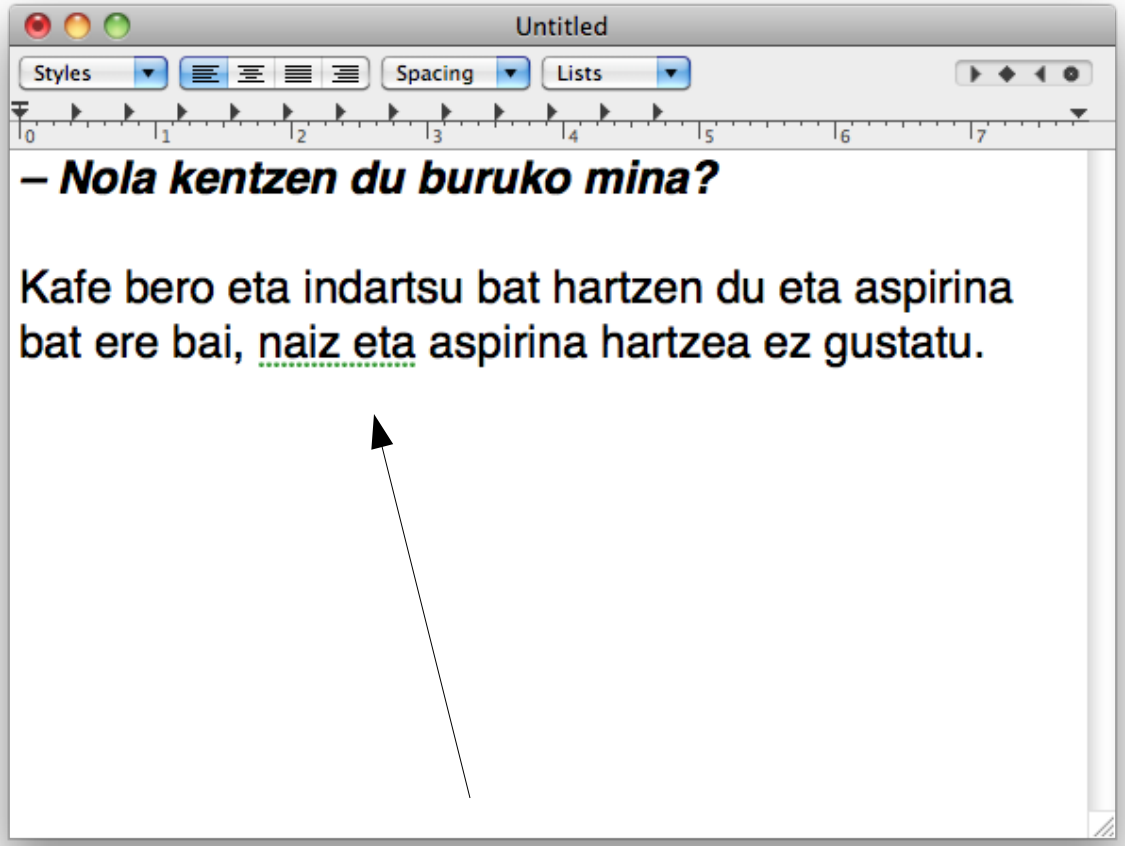
(in engcorr4.script)

Integration with system spell checkers





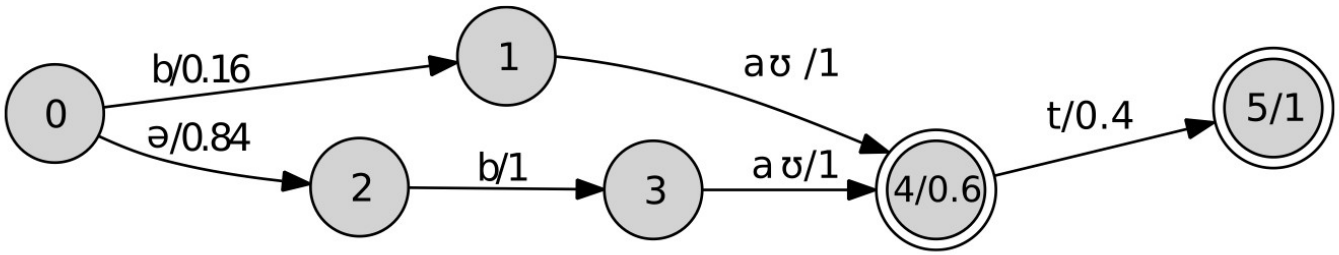
Integration with system grammar checkers



common competence error: should be “nahiz eta,” but “naiz” is also a real word, and so we give a different warning...

PART II - Weighted automata

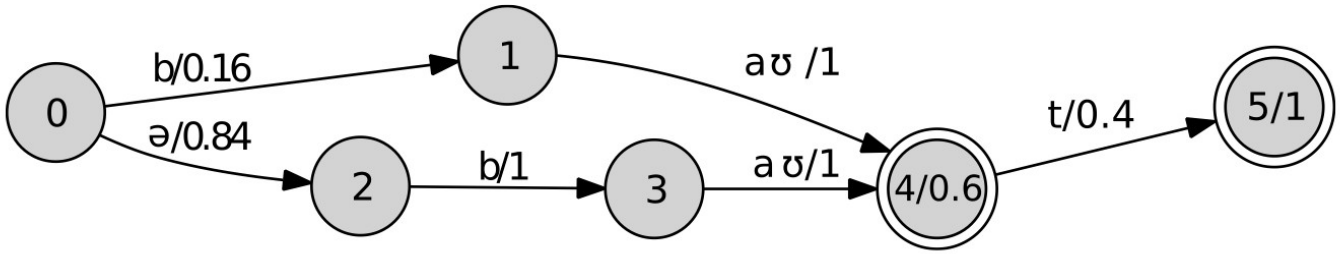
Probabilistic automaton (distribution over strings):



$$p([\text{əbaʊt}]) = 0.336\ (0.84 \times 1 \times 1 \times 0.4 \times 1)$$
$$p([\text{əbaʊ}]) = 0.504\ (0.84 \times 1 \times 1 \times 0.6)$$
$$p([\text{baʊt}]) = 0.064\ (0.16 \times 1 \times 0.4 \times 1)$$
$$p([\text{baʊ}]) = 0.096\ (0.16 \times 1 \times 0.6)$$

Weighted automata

Probabilistic automaton:



As always, we would prefer using(negative) logprobs, since this makes calculations easier:

-log(0.16) ≈ 1.8326

-log(0.84) ≈ 0.1744

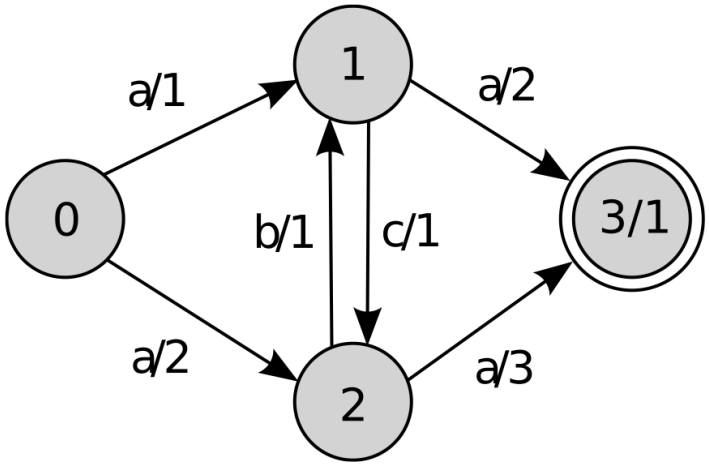
-log(1) = 0

-log(0) = ∞

Since the more probable is now numerically smaller, we call them weights

Weighted automata

Different “costs” for string aa under various semirings:

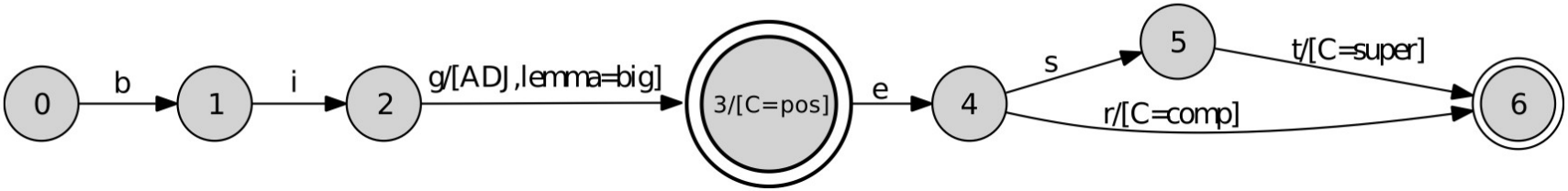


Semiring	**Calculation**
Probability (real)	$1 \times 2 \times 1 + 2 \times 3 \times 1 = 10$
Log	$(1 + 2 + 1) \oplus_{log} (2 + 3 + 1) \approx 3.873$
Tropical	$\min(1 + 2 + 1, 2 + 3 + 1) = 4$

Weighted automata

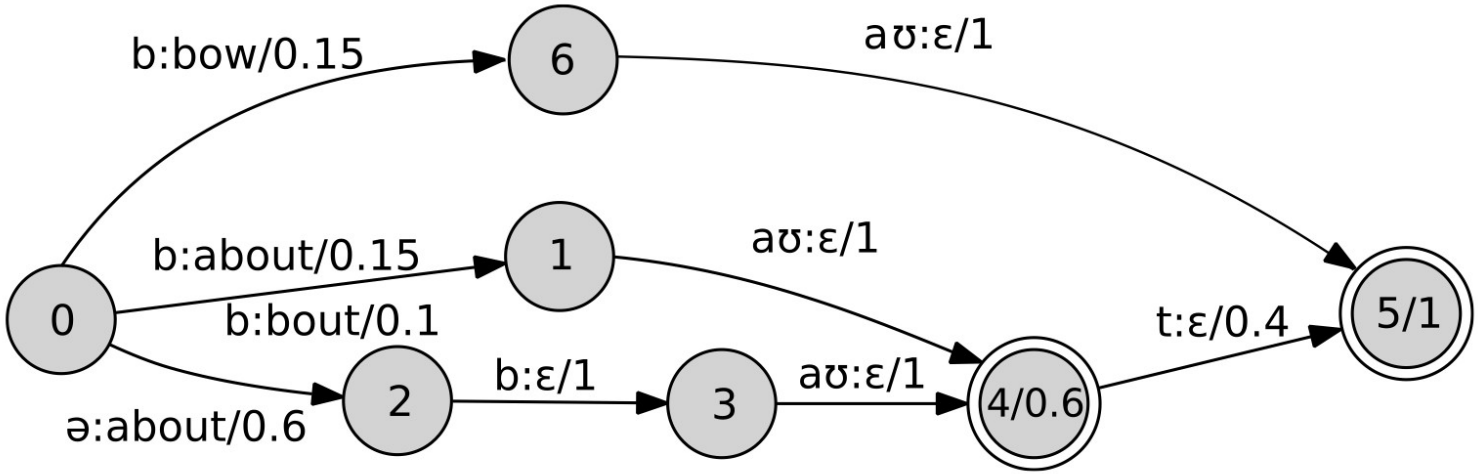
- But “weights” can be defined in almost any structure as long as it obeys some rules

- Example weight structure with feature-value pairs:



Weighted transducers

- All of the above generalizations apply to transducers as well
- Example “pronunciation” transducer:

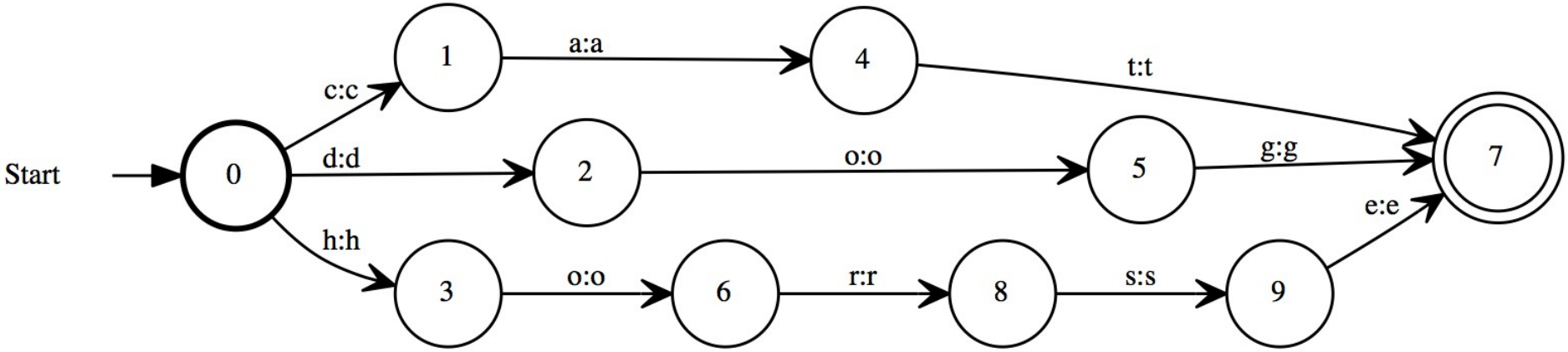


Kleene intro.

- Language for manipulating weighted FSMs
- Very similar to other utilities
- Uses OpenFST, so can be thought of as an interface to OpenFST
- Download at *http://kleene-lang.org*
- [Examples]

Example: weighted edit distance

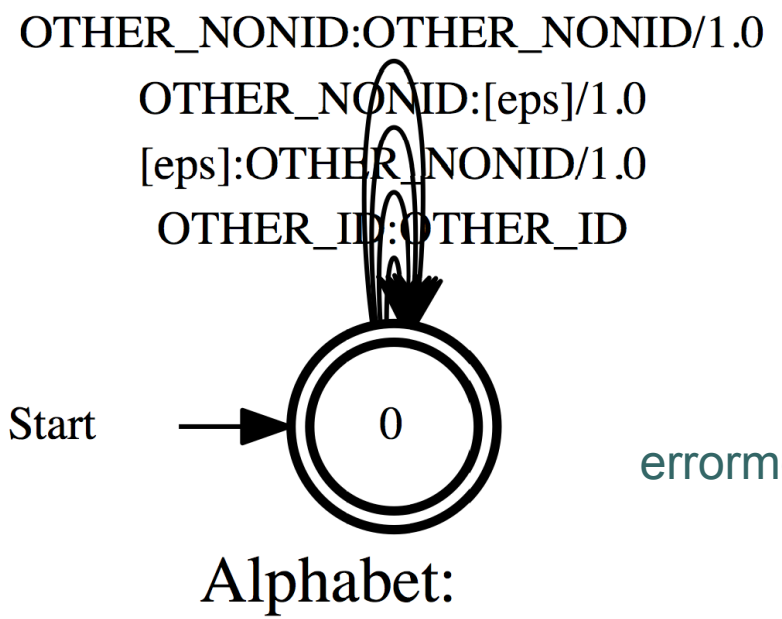
$LM = (cat|dog|horse) ;



Alphabet: g, d, t, e, s, r, c, a, o, h

Example: weighted edit distance

```
$rep = . ; $ins = "":.; $del = .:""; $chg = .:.-.;
$EM = ( $rep<0.0> | $ins<1.0> | $del<1.0> |
$chg<1.0> )*;
```

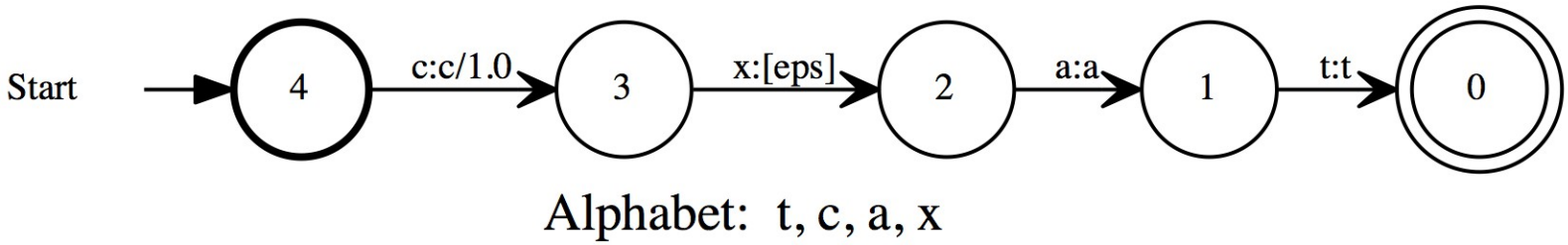


errormodel.kl

Example: weighted edit distance

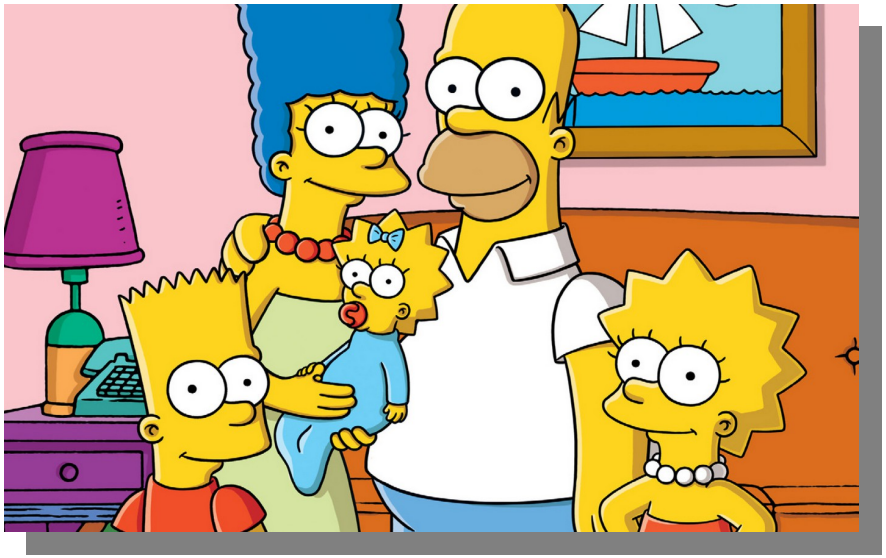
```
$corr = (cxat) _o_$EM _o_ $LM;
// FstType: vector, Semiring: standard, 50 states, 139 arcs, 938 paths, Transducer, Weighted, Closed Sigma
```

```
$corr = $^shortestPath( (cxat) _o_$EM _o_ $LM );
```



More complex LM (still unigram)

$LM = (the<3.3123733563043>|
you<3.40834334278697>
i<3.47764362842074>|
a<3.62151061674717>|
to<3.74035111367985>|
and<4.12455498051775>
of<4.2521768299548>|
...



Unigram model from The Simpsons word frequency list
(http://pastebin.com/anKcMdvk)

Noisy channel modeling

$p(w|s) \propto p(s|w)\ p(w)$

$correctioncost = (word) _o_ $EM _o_ $LM

(we want to the the maximum *w*)

s = scrambled word
w = intended word

Kleene functions

$p(w|s) \propto p(s|w)\ p(w)$

$correction = (word) _o_ $EM _o_ $LM

We can wrap this into a function:

```
$^correct($word) {
    return $^shortestPath($^lowerside($word _o_ $EM _o_ $LM));
}
```

```
print $^correct(hoxse);
horse : 1.0
```

Real-world task: tweet normalization

“Normalize” tweets in Spanish (bakeoff at SEPLN 2013):

Variation

El cafe que quiero en mis mañanas es el de tus ojos.

Variation Correct

Si es que tenia yo razon pasando x la Isla estaba el desvio para Zaragoza jajajaja

@[redacted] jajajaja ok, ok!

NoES OOV OOV OOV OOV

Es que estamos siempre muy atentos a todo XD aajajajajajajajaj aiiiii las mejores visitas las de mi inflaaa joeee!!

Common strategy among participants:

(1) “Learn” an error model from aligned real/normalized tweets ($EM)

```
es q e      sepaaas      to     ...
es_que      sepa  s      toda   ...
```

(2) Get a language model from somewhere, and encode it as weighted automaton ($LM)

(3) Calculate:

```
$^shortestPath($^lowerside($sentence _o_ $EM _o_ $LM));
```

Morphology with Kleene: right-linear grammars

Right-linear grammar (to replace lexc):

```
S -> a X
X -> b | b X
```

in Kleene:

```
$S> = a $>X;
$>X = b | b $>X;
$Grammar = $^start($>S);
```

Morphology with Kleene

Kleene

```
// LEXICON

$>Root = ($>Noun | $>Verb) ;
$>Verb = (beg|fox|talk|panic|try|watch) ($>Vinf) ;
$>Noun = (cat|city|fox|panic|try|watch) ($>Ninf) ;
$>Ninf = (\+N\+Sg):"" | (\+N\+Pl):(\^s);
$>Vinf = (\+V):"" | (\+V\+\3P\+Sg):(\^s) |
         (\+V\+Past):(\^ed) | (\+V\+PastPart):(\^ed) |
         (\+V\+PresPart):(\^ing) ;
$Lexicon = $^start($>Root) ;
```

LEXC

```
LEXICON Root

Noun ;
Verb ;

LEXICON Noun

cat   Ninf;
...
```

Kleene phon. rules

Kleene

```
// RULES

$V = [aeiou] ;

$ConsonantDoubling = g -> gg / _ \^ (ing|ed) ;
$EDeletion = e -> "" / _ \^ (ing|ed) ;
...
```

foma

```
# Vowels
define V [ a | e | i | o | u ];

define ConsonantDoubling g -> g g || _ "^" [i n g | e d ];
define EDeletion e -> 0 || _ "^" [ i n g | e d ] ;
...
```

Testing grammars

source “english.kl”;
test $Grammar;