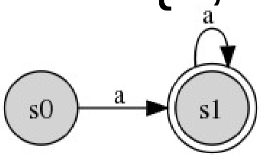
Normalization of dialects and variants using FST technology
Rules in foma

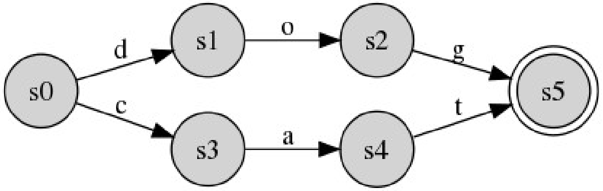
Iñaki Alegria
(University of The Basque Country)

Languages: finite automata

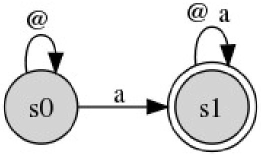
“one or more as”: {a,aa,...}:



the words “cat” and “dog”:



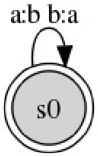
any word that contains at least an a:



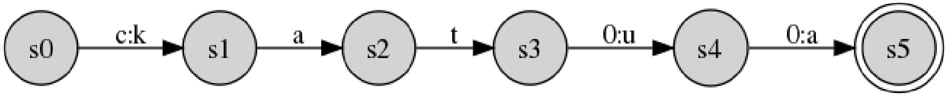
@ = any symbol outside the defined alphabet

Finite transducers

Translates all a-symbols to b and vice versa



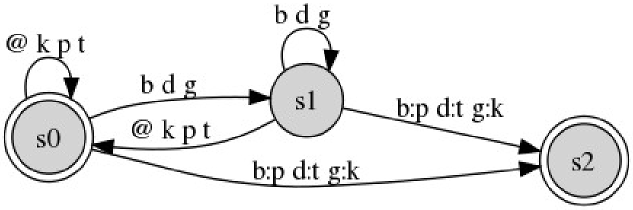
Translates “cat” to “katua”



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

Rule-based normalization tends to model changes from variants to standard/pivot/canonical forms writing rules, which are based on regular expressions and compiled into transducers

variants → standard forms

With this aim, basically phonological changes will be described (but lemmas, affixes, stems and other morphological elements can be interesting)

phoneme → phoneme

morph-element → morph-element

Each change/rule is compiled into a transducer and the transducers can be composed all together

Using *foma*

- Unix-like commands
- A general-purpose tool for constructing and manipulating automata and transducers
- Contains a regular expression compiler to convert expressions (including “rewrite rules”) to transducers
- Contains a lexc-parser to construct transducers from lexicon descriptions (no for today)
- API available (in C) for integration with other programs
- [source & binaries for Linux, Mac, and Windows]

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

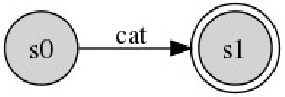
(space)	concatenation
|	union
*	Kleene star
&	Intersection
~	Complement

foma: ordinary symbols

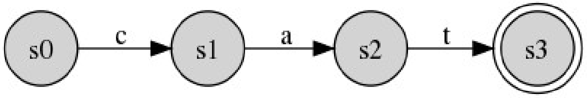
Single-character symbols:
a, b, c, Ω, ب , β, etc.

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:

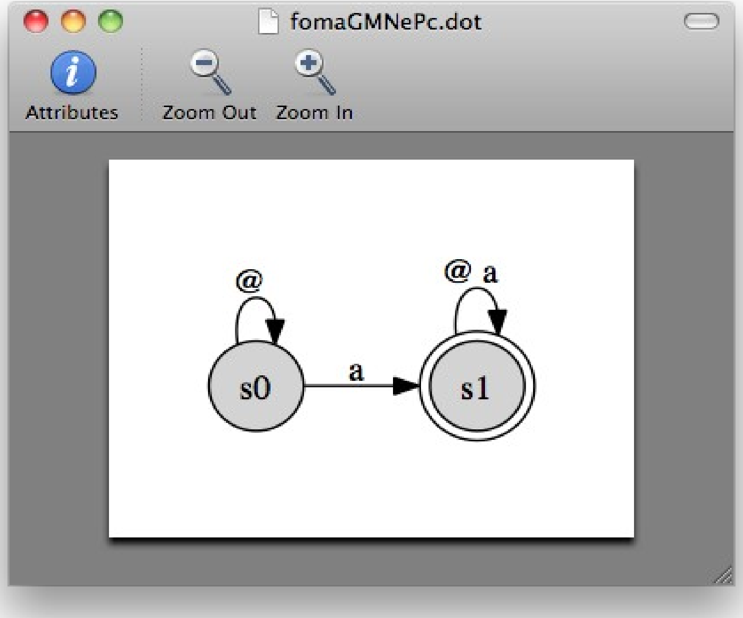
```
foma[0]: regex [c a t]:[k a t u a];
317 bytes. 6 states, 5 arcs, 1 path.
foma[1]: down
apply down> cat
katua
apply down> dog
???

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
```

Optional replacement

Simple optional replacement:

```
foma[0]: regex a (->) b ;
290 bytes. 1 states, 3 arcs, Cyclic.
foma[1]: down
apply down> a
a
b
apply down> axa
axa
axb
bxa
bxb
```

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	**:**	Cross-product
|	union	A **->** B	Replacement rules
*	Kleene star	A **(->)** B	Opt. replacement
+	Kleene plus	A -> B **||** C _ D	Context-conditioned
&	Intersection	A (->) B || C _ D	
~	Complement	**.o.**	Composition
(A**)**	Optionality (identical to A | 0)		

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`

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/upper side words (for a transducer):

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

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

Load word-lists:

`read txt name.txt`

Save binary (for including or for using flookup in test-scripts):

`save stack name.fst`

foma and flookup

First steps using REs:

```
foma
```

Writing, compiling and testing grammars:

```
gedit grammar.txt &
foma -l grammar.txt
```

Scripting (test):

```
flookup grammar.fst # equivalent to apply up
flookup -i          # equivalent to apply down
flookup -h          # list options
flookup -ibx        # inverse, no buffer, no echo
```