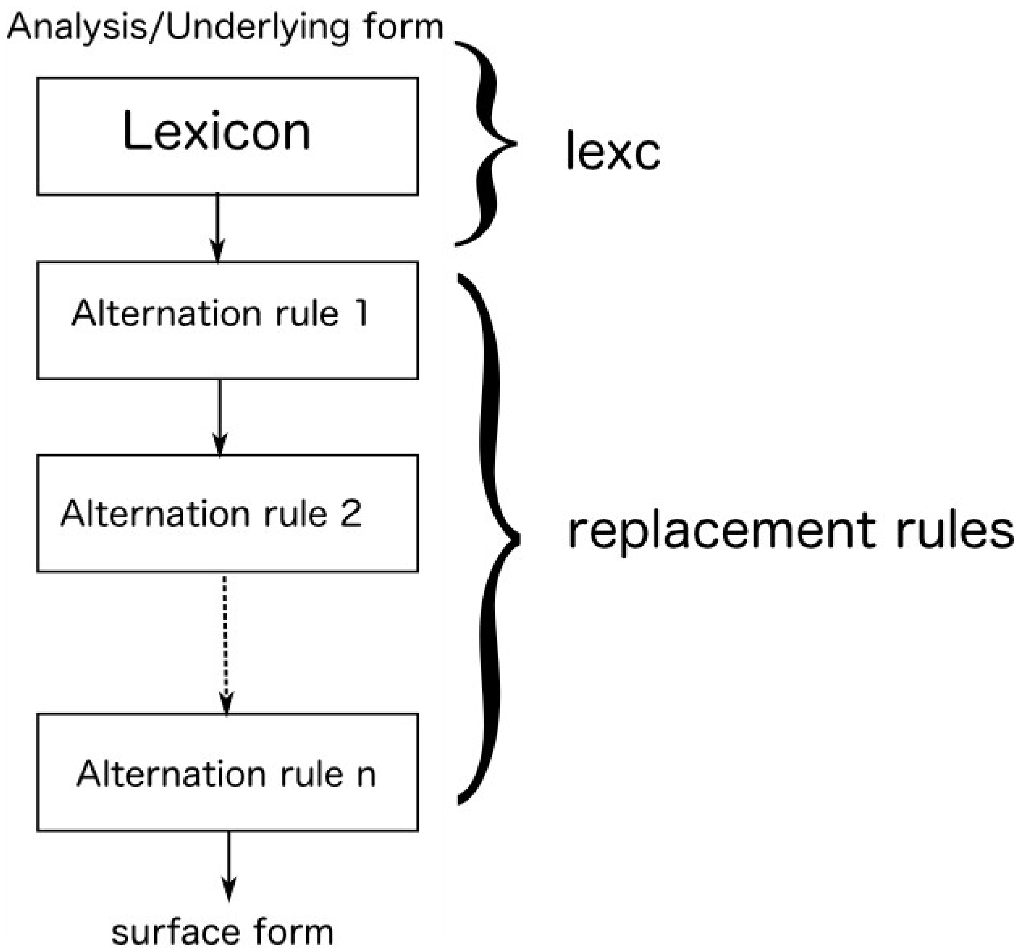
Creating LRs with FSTs
Part III
The lexc formalism

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

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
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 #;
  #;
```

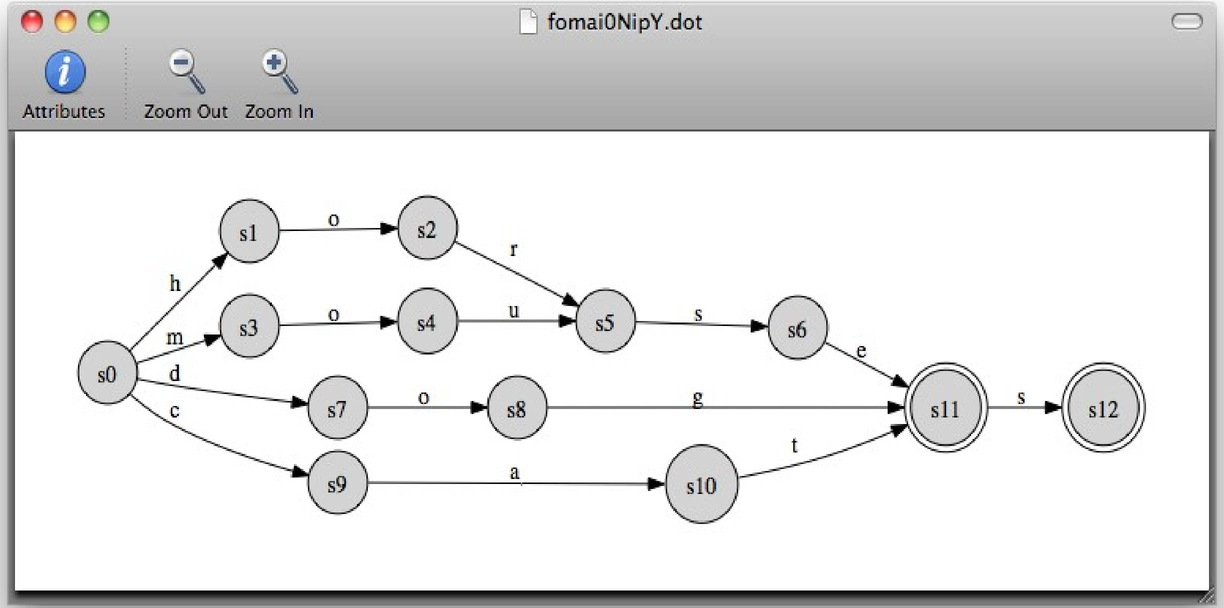
- From LEXICON Suff, we can choose either “s” and go to end-of-word, or the “empty string” and go to end-of-word
- This makes the suffix (optional), and we can construct both “cat” and “cats”

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:

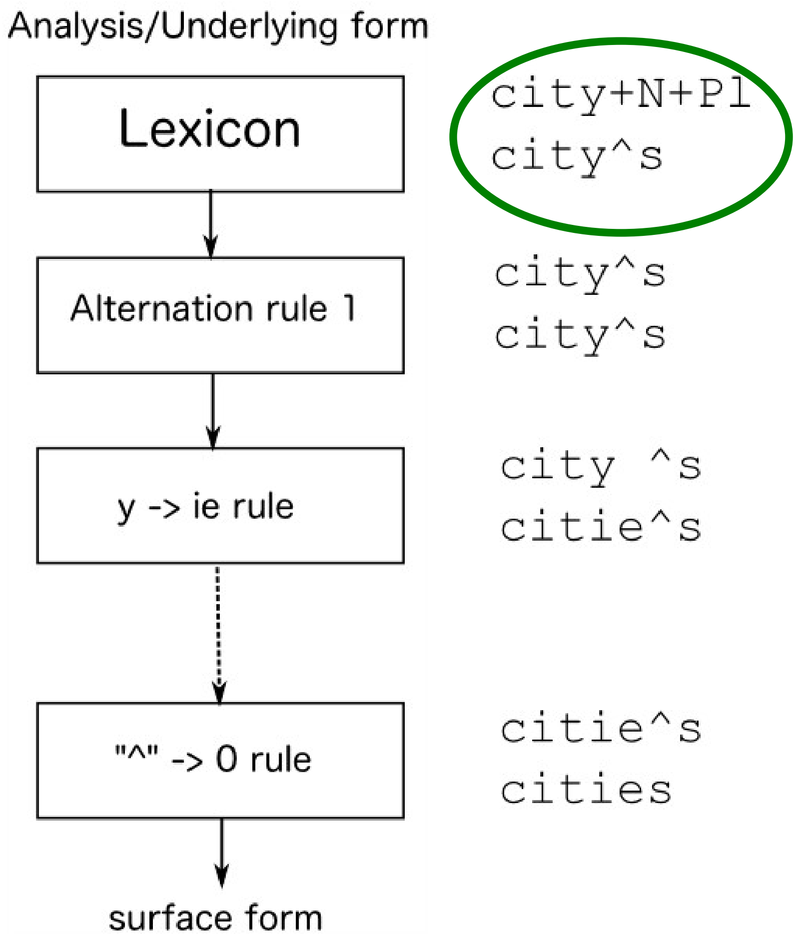
watche^s

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

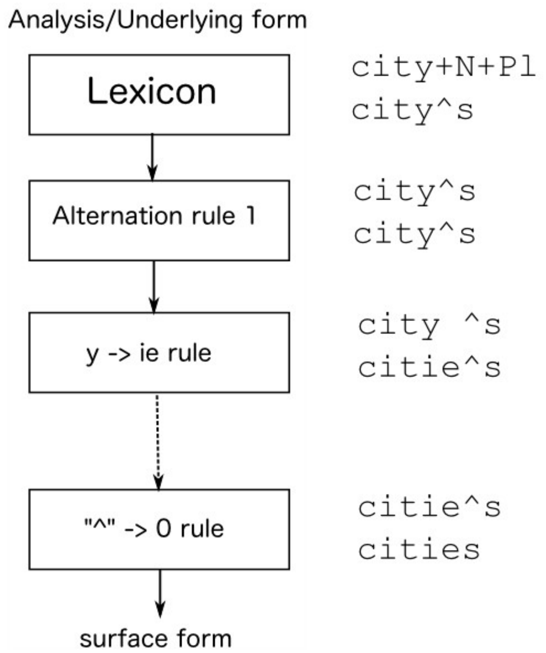
Points to observe:

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]
```