6.863J Natural Language Processing Lecture 2: Automata, Two-level phonology, & PC-Kimmo (the Hamlet lecture)

Instructor: Robert C. Berwick
berwick@ai.mit.edu

The Menu Bar

- Administrivia
  web page: www.ai.mit.edu/courses/6.863/ now with Lecture 1, Lab components I (background), II (Lab 1a) – due date is February 17 (holiday the 16th)
- *What* and *How:* word processing, or computational morphology
- What's in a word: morphology
- Modeling morphophonology by finite-state devices
- Finite-state automata vs. finite state transducers
- Some examples from English
- PC-Kimmo & Laboratory 1a :how-to

Levels of language

- Phonetics/phonology/morphology: what words (or subwords) are we dealing with?
- Syntax: What phrases are we dealing with? Which words modify one another?
- Semantics: What's the literal meaning?
- Pragmatics: What should you conclude from the fact that I said something? How should you react?

Start with words: they illustrate all the problems (and solutions) in NLP

- Parsing words
  Cats → CAT + N(oun) + PL(ural)
- Used in:
  - Traditional NLP applications
  - Finding word boundaries (e.g., Latin, Chinese)
  - Text to speech (*boathouse*)
  - Document retrieval (example next slide)
- In particular, the problems of *parsing*, *ambiguity,* and *computational efficiency* (as well as the problems of *how people do it*)

Terminology

- Morpheme, etc..
- See webpage
  http://pandora.cii.wwu.edu/vajda/ling201/test1materials/Morphologyoverhead.htm

Two parts to the "what"

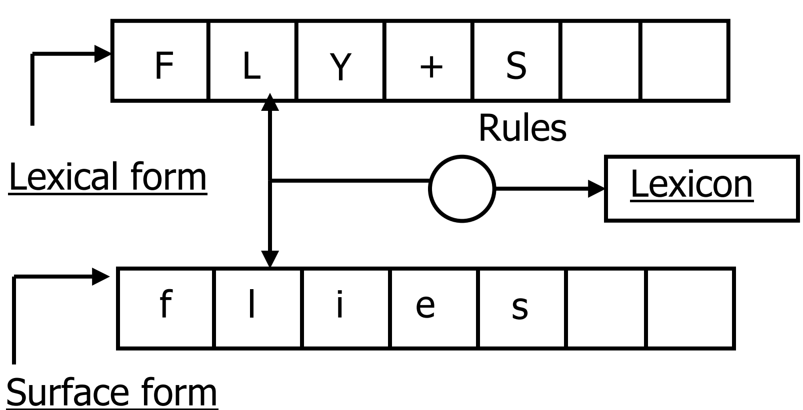
1. Which units can glue to which others (roots and affixes) (or stems and affixes), eg,
2. What 'spelling changes' (orthographic changes) occur – like dropping the *e* in 'chase + ed'

OK, let's tackle these one at a time

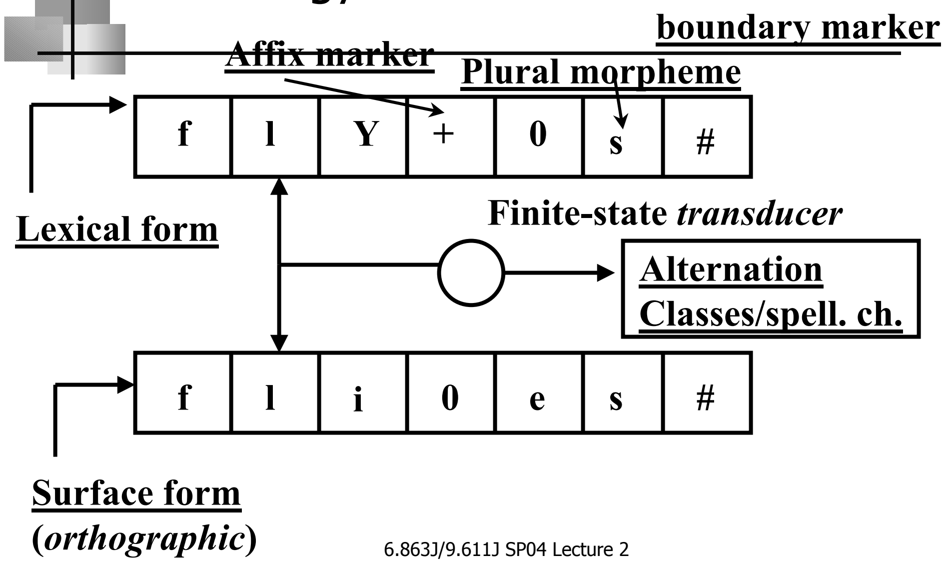
The Knowledge

- ✓ Which units can glue to which others (roots and affixes) (or stems and affixes)
- ✓ What 'spelling changes' (orthographic changes) occur – like dropping the *e* in 'chase + ed'

Our goal: PC-Kimmo



Terminology



What is the Knowledge?

- Morpheme classes
- Like beads on a string
- Computational model:
  - Finite-state automata

Two <u>parallel</u> finite-state machines

- Machine 1: order of morphemes
- Machine 2: spelling changes

Why finite-state machines?

- Minimal model
- Fast
- What makes up a finite-state automaton?
- Linear concatenation of <u>equivalence classses</u>

Definition of finite-state automaton (fsa)

- A *finite-state automaton* (FSA) is a quintuple (Q, Σ, δ, q_0, F) where
  - *Q* is a *finite* set of states
  - Σ is a finite set of terminal symbols, the alphabet
  - $q_0 \in Q$ is the initial state
  - $F \subseteq Q$, the set of final states
  - δ is a mapping from $Q \times \Sigma \rightarrow 2^Q$, the transition mapping

Application to morphology

- Purely concatenative morphemes like a rooted tree (but what about prefixes?)

English morphology: what states do we need for the fsa?

- As an example, consider adjectives

  Big, bigger, biggest

  Cool, cooler, coolest, coolly

  Red, redder, reddest

  Clear, clearer, clearest, clearly, unclear, unclearly

  Happy, happier, happiest, happily

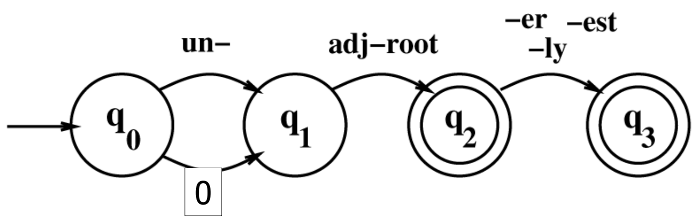
  Unhappy, unhappier, unhappiest, unhappily

  Real, unreal, silly

FSA states

- Equivalence classes under the notion of 'substitution': elements that 'behave alike'
- Consider: ___er What goes in the space?
  - Cool, big, happy (Adjectives that can be comparative)
- If we need to make more refinements, we need a new class

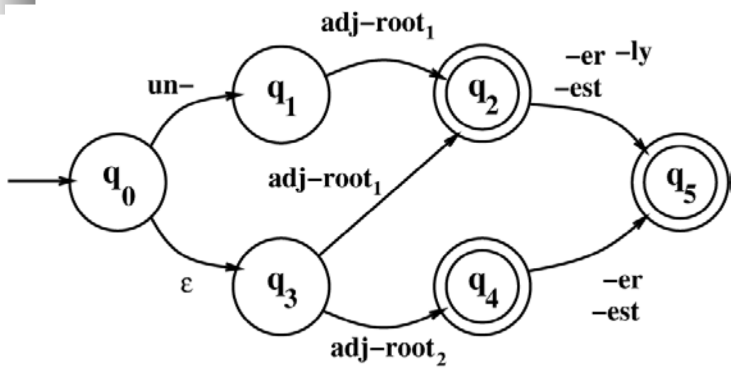
Will this fsa work?



Ans: no!

- Accepts all adjectives above, but
- Also accepts *unbig, readly, realest*
- Common problem: <u>*overgeneration*</u>
- Solution?

Revised picture



Extension to politics: an alien language

- Bush could win the election
- Bush will win the election
- Bush did win the election
- Bush could have won the election
- Bush will have won the election

…

Structural linguistics circa 1940s

"[Words] are assigned to classes on the basis of the environments in which they occur. Each environment determines one and only one class, namely the class of all [words] occurring in that environment… A word *A* belongs to the class determined by the environment ____X if X is either an utterance or occurs as part of some utterance." Wells, 1947, pp. 81-82.

Finite transition networks (FTNs)

- 1-1 picture of fsa-ftn
- Easy to see, easy to prove certain properties: closure under concatenation, intersection
- Conversion of nondeterministic to deterministic fsa – always possible

Limits of FTNs

- Finite # of 'memory states'
- Can distinguish only a finite # of classes (bins) i.e., the states
- This sets limits on the patterns fsa's can recognize
- FSA's <u>cannot</u> even <u>describe</u> all possible human words…

Then can be *indescribable* words (for an fst)

- Can we even do all natural languages?
- Example: Bambarra (African language in Mali)
- Words in form *Noun+o+Noun,* as in *wuluowulo* ='whichever dog'
- Also have repeated endings (like anti-anti…)
  wulu+nyini+la ='dog searcher'
  wulunyinina+ nyini+la ='one who searches for dog searchers'
- Fatal bite: combine with word *o word* formation:
  wulunyininanyinila o wulunyininanyinila (arbitrarily long!)

Why is this not describable by an fsa?

- Intuition: w^n o w^n language
- Need arbitrary # of bins to keep track of w^n string to match it up with w^n after the ‘o’
- Must be able to count to arbitrary *n* to keep track of # of copies…
- But, only a finite # of bins…so….

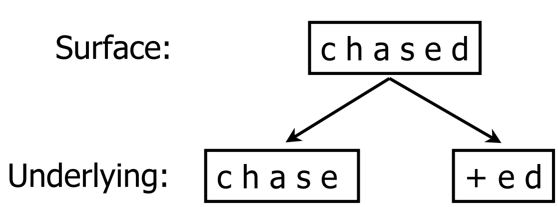
What does fsa machine for English look like?

Next: what about the spelling changes? That's harder!

- ✓ Which units can glue to which others (roots and affixes) (or stems and affixes)

2. What 'spelling changes' (orthographic changes) occur – like dropping the *e* in 'chase + ed'

Mapping between surface form & underlying form



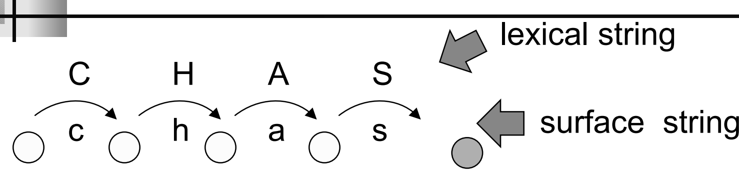
But clearly this can go either way – given the underlying form, we can *generate* the surface form – so we really have a *relation* betw. surface & underlying form, viz.:

Conventional notation

Lexical (underlying) form: `c h a s e + e d`
Surface form: `c h a s 0 0 e d`

The 0's "line up" the lexical & surface strings
This immediately suggests a finite-state automaton 'solution' : an extension known as a
finite-state transducer

Finite-state transducers: a *pairing* between lexical/surface strings



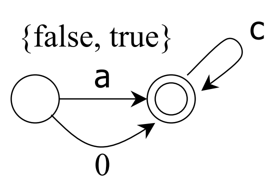
- Or more carefully

Definition of finite-state transducer

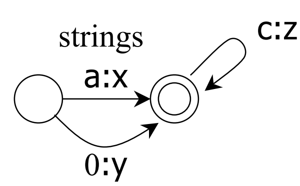
- state set Q
- initial state q_0
- set of final states F
- input alphabet S (also define Σ^*, Σ^+)
- output alphabet D
- transition function $\delta : Q \times \Sigma \rightarrow 2^Q$
- output function $\sigma : Q \times \Sigma \times Q \rightarrow D^*$

The difference between (familiar) fsa's and fst's: functions from…

Acceptors (FSAs)



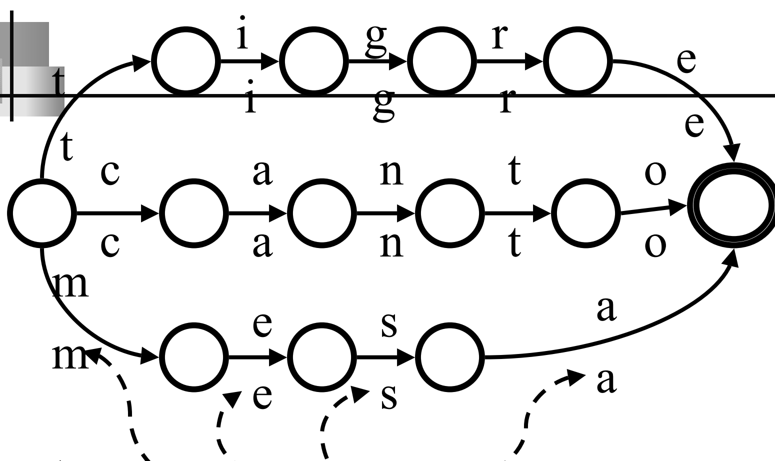
Transducers (FSTs)



Properties of fst's – compare to fsa's

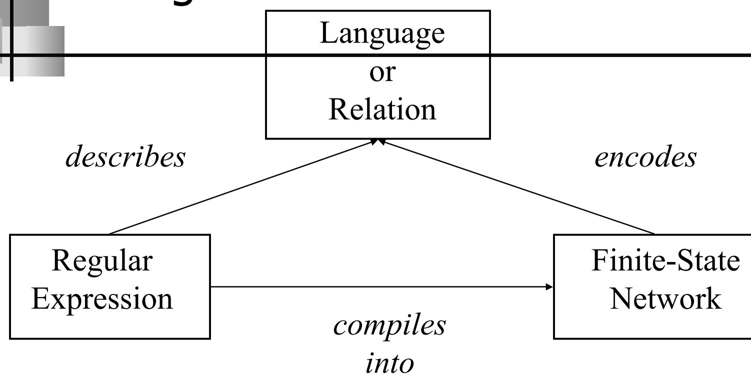
- Closed under concatenation – get another fst if wired together
- NOT closed under intersection
- NOT always able to make deterministic

A Two-Level Transducer



Input: m e s a
Output: m e s a

The Big Picture



Defining an fst for a spelling-change rule

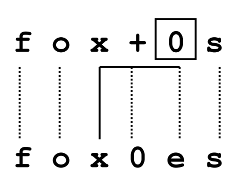
- Suggests all we need to do is build an FST for a spelling-change rule that ‘matches’ lexical and surface strings
- Example: fox+s, foxes; buzz+s, buzzes
- Rule: e <u>before non initial x,s,z</u>
- Instantiation as an FST:

F	**O**	**X**	**+**	**0**	**S**	**#**	**lexical**
f	o	x	0	e	s	#	**surface**

Implication

- 0:e can occur <u>only</u> in this context
- Must write this as a <u>constraint</u>
- Write an FTN that accepts only strings of this form, e.g., dafjakjdx0:es#

FTN imposes a <u>Constraint</u>



0:e correspondence requires a preceding x on the lexical side, s:s following

In this context, all other possible realization of a 0:s are prohibited.

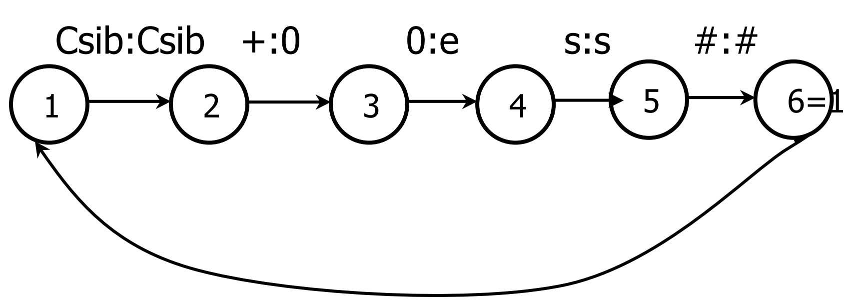
Turning this into an fst

- Write down the left, center, and right context
- In this case:

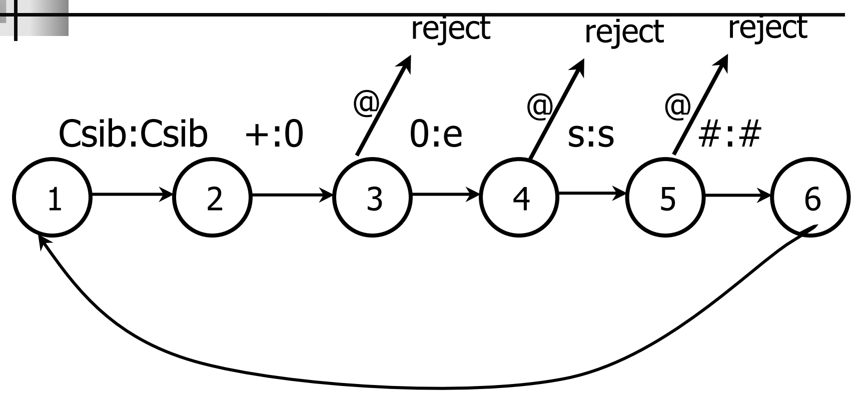
  x:x +:0 0:e s:s #:#

  Csib:Csib
- Pad out with nulls (0’s)
- Write an fsa (ftn) that accepts exactly this string

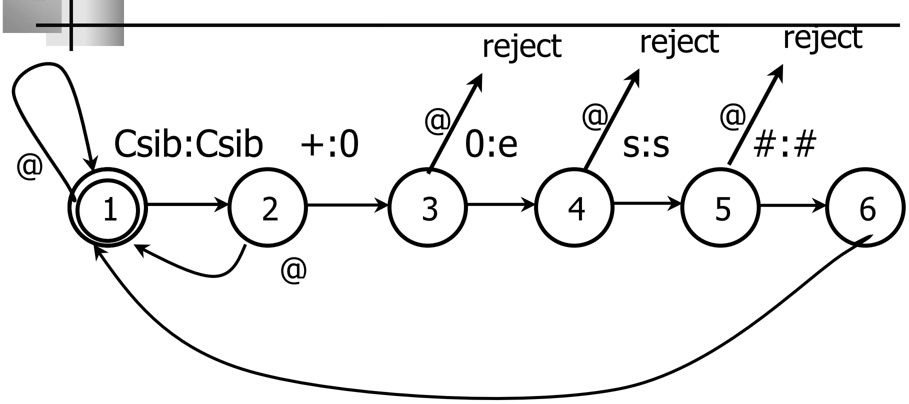
Start with straightline fst

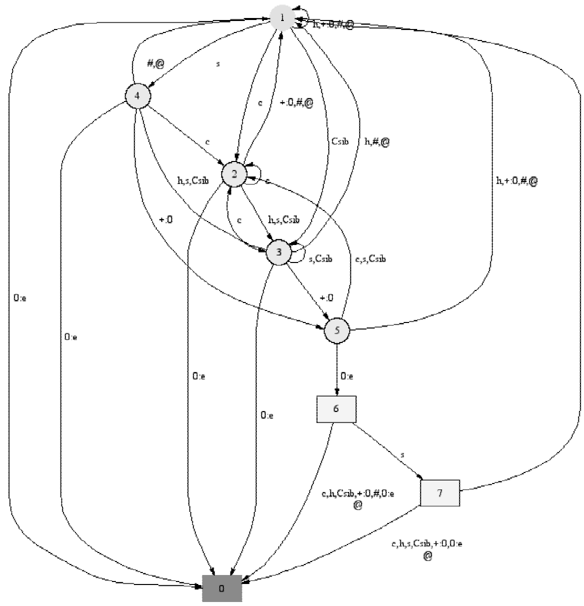


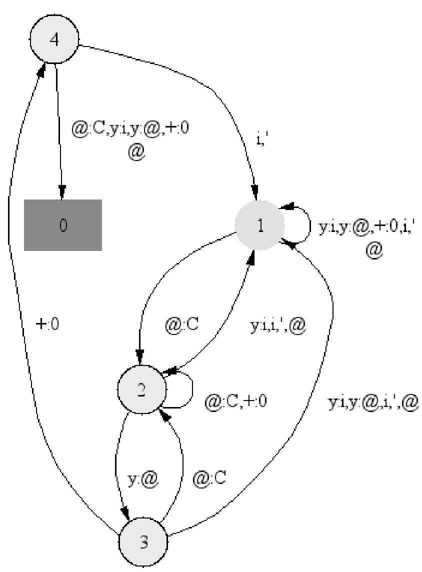
Now add rejection notices…



And acceptance (cook until done)




1
h,+:0,#,@
#,@
s
4
c
+:0,#,@
c
Csib
h,#,@
2
h,s,Csib
c
h,+:0,#,@
c
h,s,Csib
+:0
3
s,Csib
c,s,Csib
#
0:e
+:0
0:e
5
0:e
0:e
6
0:e
s
7
c,h,Csib,+:0,#,0:e
@
c,h,s,Csib,+:0,0:e
@
0


4
@:C,y:i,y:@,+:0
@
i,'
0
1
y:i,y:@,+:0,i,'
@
+:0
@:C
y:i,i,',@
2
@:C,+:0
y:i,y:@,i,',@
y:@
@:C
3

Tabular format

```
RULE "3 Epenthesis, 0:e => [Csib|ch|sh|] +:0___s [#]"
7 8
   c h s Csib + # 0 @
   c h s Csib 0 # e @
1: 2 1 4 3    1 1 0 1
2: 2 3 3 3    1 1 0 1
3: 2 1 3 3    5 1 0 1
4: 2 3 3 3    5 1 0 1
5: 2 1 2 2    1 1 6 1
6. 0 0 7 0    0 0 0 0
7. 0 0 0 0    0 1 0 0
```


And that's (almost) all folks…

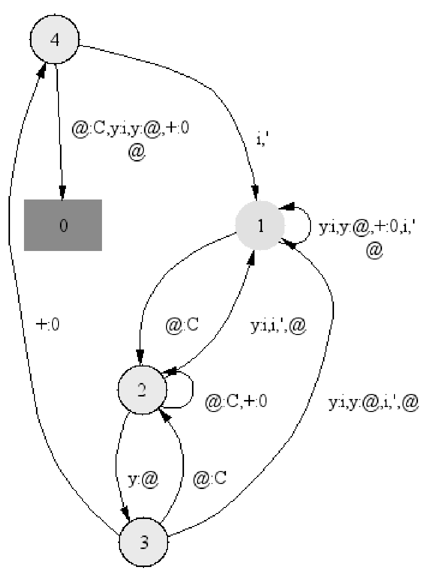
- Except…
- There's more than one rule…

- Spy+s → spies
- Quiz+s → quizzes
- Make+ing → making

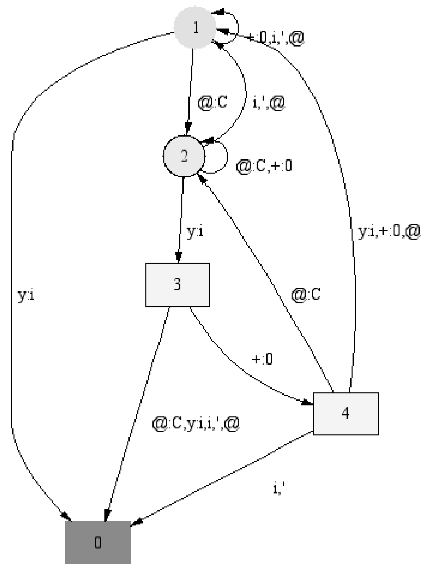
Spelling change rules

Name	Description	Example
Consonant Doubling (gemination, G)	1-letter consonant doubled before -*ing/ed*	beg/begging
E deletion (elision, EL),	Silent *e* dropped before *-ing, -ed*	make/making
E insertion (epenthesis, EP)	*e* added after *-s, -z, -ch, -sh* before *-s*	fox/foxes
Y replacement (Y)	*-y* changes to *-ie* before *-ed*	try/tries
I spelling (I)	*I* goes to *y* before vowel	lie/lying

Set of 5 spelling change FTNs



Another…name that automaton…



So, we're done, right?

- So, not so fast…!!!!
- Sometimes, more than 1 spelling change rule applies. Example: spy+s, spies: y
- *y* goes to *i* before an inserted *e* (compare, "spying"
- *e* inserted at affix +s
- Here's the picture:

Two-Level Constraints 2

y:i <=> _ 0:e

0:e <=> Cons: y: _ +:0 s:s

Another Example from English ("gemination")

underlying: quiz + s

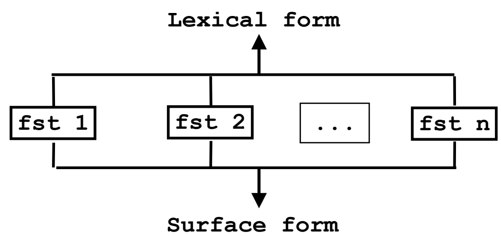
Rule A: *s -> es after z*

intermediate: quiz + es

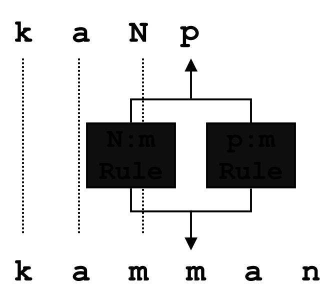
Rule B: *z doubles before Suffix beginning with vowel*

surface: quizzes

Run transducers in parallel?



Parallel application – how?

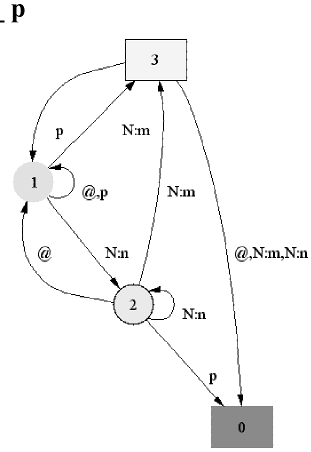


Sequential Application

```
k a N p a n          N -> m / _ p
    ↓
k a m p a n          p -> m / m _
      ↓
k a m m a n
```

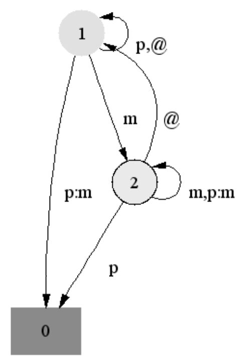

Machine Rule 1 ("N goes to m")

Rule 1: N→m | __ p



Machine Rule 2 ("p goes to m")

Rule 2: p→ m | m___



Sequential Application in Detail

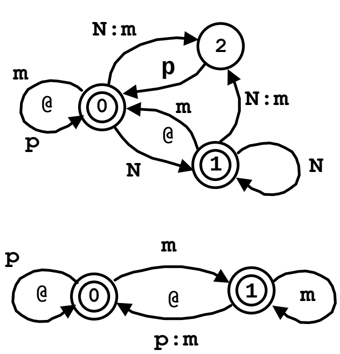
k a N p a n

0 0 0 2 0 0 0

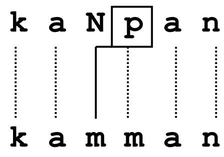
k a m p a n

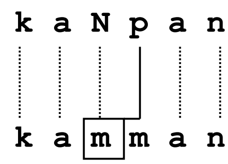
0 0 0 1 0 0 0

k a m m a n



Constraints on both sides





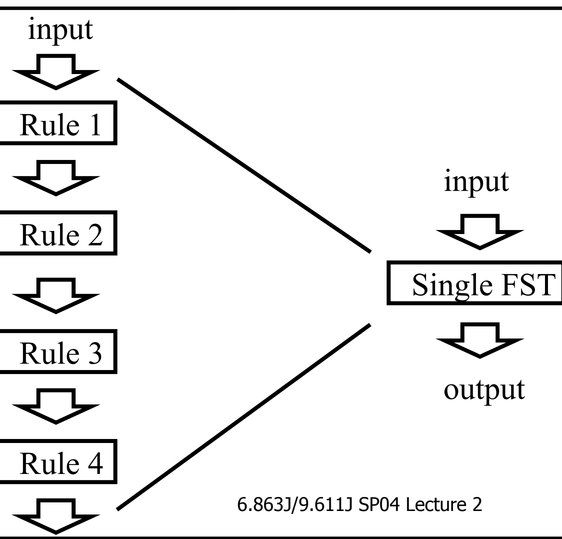
N:m correspondence requires a following p on the lexical side.

In this context, all other possible realization of a lexical N are prohibited.

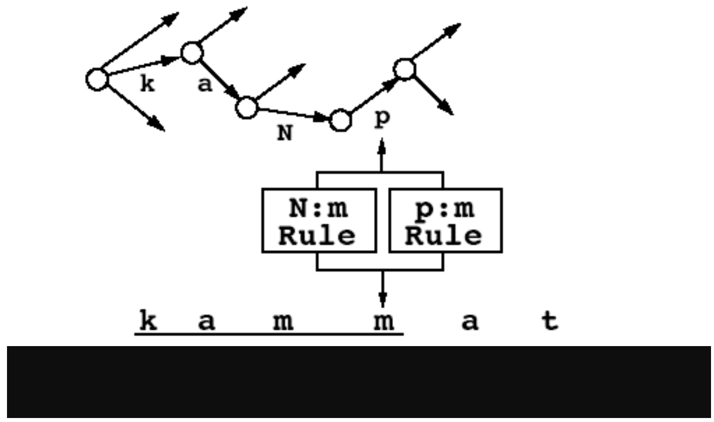
p:m correspondence requires a preceding m on the surface side.

In this context, all other possible realization of a lexical p are prohibited.

When is this possible?



Plus lexicon – lexical forms always constrained by the path we're following through the lexicon tree



We trace through the finite-state devices in tandem

recognizing *'foxes'*
root= **always 1st 'class'**