CMPT 413
Computational Linguistics

Anoop Sarkar
`http://www.cs.sfu.ca/~anoop`

Finite-state transducers

- Many applications in computational linguistics
- Popular applications of FSTs are in:
  - Orthography
  - Morphology
  - Phonology
- Other applications include:
  - Grapheme to phoneme
  - Text normalization
  - Transliteration
  - Edit distance
  - Word segmentation
  - Tokenization
  - Parsing

Orthography and Phonology

- Orthography: written form of the language (affected by morpheme combinations)
  move + ed → moved
  swim + ing → swimming S W IH1 M IH0 NG
- Phonology: change in pronunciation due to morpheme combinations (changes may not be confined to morpheme boundary)
  intent IH2 N T EH1 N T + ion
  → intention IH2 N T EH1 N CH AH0 N

Orthography and Phonology

- Phonological alternations are not reflected in the spelling (orthography):
  - Newton Newtonian
  - maniac maniacal
  - electric electricity
- Orthography can introduce changes that do not have any counterpart in phonology:
  - picnic picnicking
  - happy happiest
  - gooey gooiest

Segmentation and Orthography

- To find entries in the lexicon we need to segment any input into morphemes
- Looks like an easy task in some cases:

  looking → look + ing

  rethink → re + think
- However, just matching an affix does not work:

  **thing* → th + ing

  **read* → re + ad
- We need to store valid stems in our lexicon

  what is the stem in *assassination* (*assassin* and not *nation*)

Porter Stemmer

- A simpler task compared to segmentation is simply stripping out all affixes (a process called **stemming**, or finding the stem)
- Stemming is usually done without reference to a lexicon of valid stems
- The Porter stemming algorithm is a simple composition of FSTs, each of which strips out some affix from the input string
  - input=..*ational*, produces output=..*ate* (*relational* → *relate*)
  - input=..V..*ing*, produces output=ε (*motoring* → *motor*)

Porter Stemmer

- False positives (stemmer gives incorrect stem): *doing* → *doe*, *policy* → *police*
- False negatives (should provide stem but does not): *European* → *Europe*, *matrices* → *matrix*

I'm a rageaholic. I can't live without rageahol.

Homer Simpson, from *The Simpsons*

- Despite being linguistically unmotivated, the Porter stemmer is used widely due to its simplicity (easy to implement) and speed

Segmentation and orthography

- More complex cases involve alterations in spelling

  foxes → fox + s [***e*-insertion**]

  loved → love + ed [***e*-deletion**]

  flies → fly + s [***i* to *y*, *e*-deletion**]

  panicked → panic + ed [**k-insertion**]

  chugging → chug + ing [**consonant doubling**]

  **singging* → sing + ing

  impossible → in + possible [***n* to *m***]
- Called *morphographemic* changes.
- Similar to but not identical to changes in pronunciation due to morpheme combinations

Morphological Parsing with FSTs

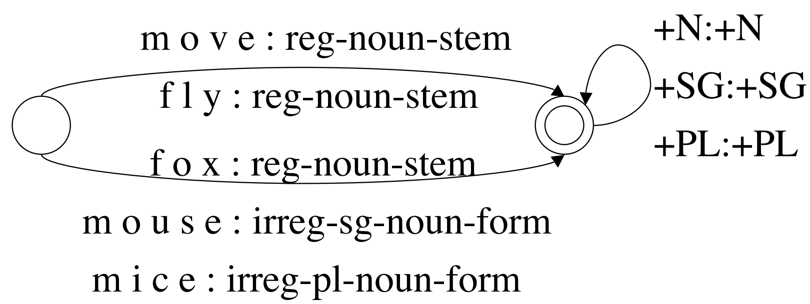
- Think of the process of decomposing a word into its component morphemes in the reverse direction: as *generation* of the word from the component morphemes
- Start with an abstract notion of each morpheme being simply combined with the stem using concatenation
  - Each stem is written with its part of speech, e.g. cat+N
  - Concatenate each stem with some suffix information, e.g. cat+N+PL
  - e.g. cat+N+PL goes through an FST to become *cats* (also works in reverse!)

Morphological Parsing with FSTs

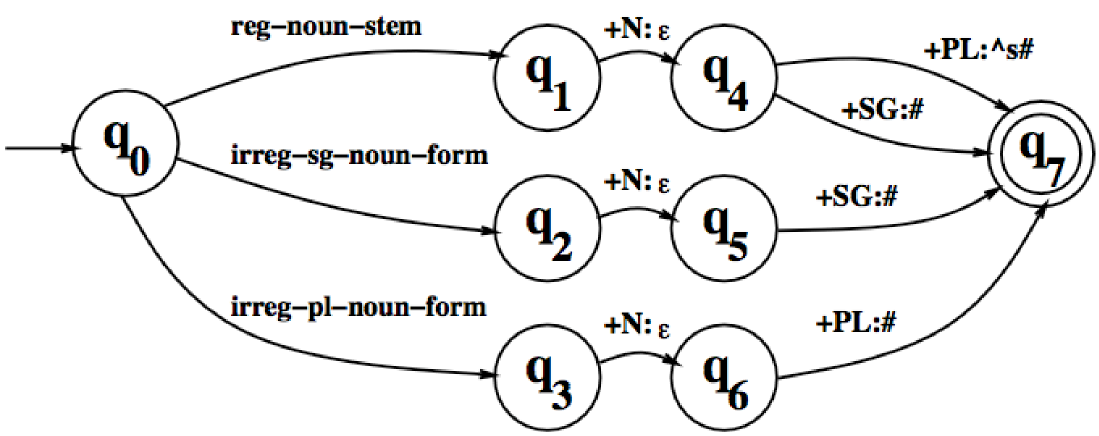
- Retain simple morpheme combinations with the stem by using an intermediate representation:
  - e.g. cat+N+PL becomes *cat^s#*
- Separate rules for the various spelling changes. Each spelling rule is a different FST
- Write down a separate FST for each spelling rule

foxes :: fox^s# [***e*-insertion FST**]
loved :: love^ed# [***e*-deletion FST**]
flies :: fly^s# [***i* to *y*, *e*-insertion FST**]
panicked :: panic^ed# [**k-insertion FST**] (arced::arc^ed#)??
etc.

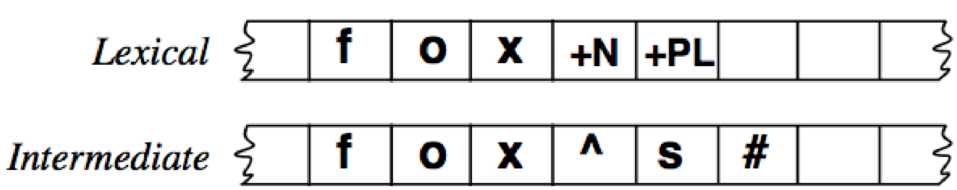
Lexicon FST (stores stems)



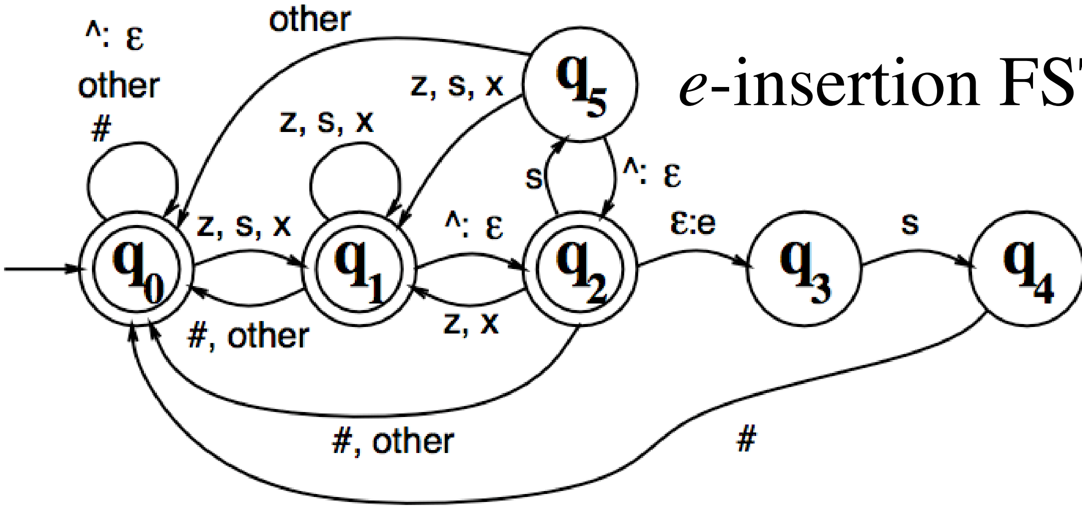
Compose the above lexicon FST with some inflection FST



This machine relates intermediate forms like `fox^s#` to underlying lexical forms like `fox+N+PL`



e-insertion FST



• The label *other* means pairs not use anywhere in the transducer.
• Since # is used in a transition, q_0 has a transition on # to itself
• States q_0 and q_1 accept default pairs like (*cat^s#, cats#*)
• State q_5 rejects incorrect pairs like (*fox^s#, foxs#*)

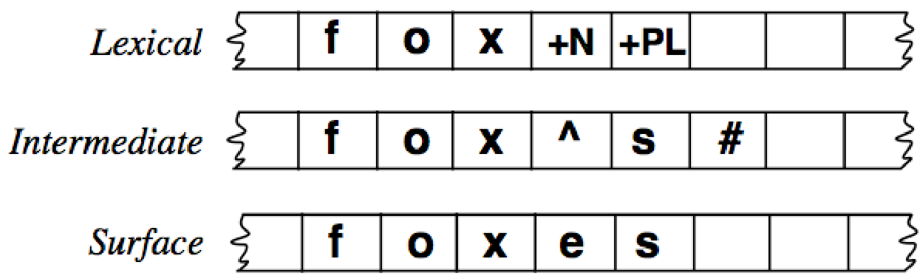
e-insertion FST

- Run the e-insertion FST on the following pairs:

(*fir#, fir#)*	(*fizz^s#, fizzs#*)
(*fir^s#, firs#*)	(*fizz^s#, fizzes#*)
(*fir^s#, fires#*)	(*fizz^ing#, fizzing#*)

- Find the state the FST reaches after attempting to accept each of the above pairs
- Is the state a final state, i.e. does the FST accept the pair or reject it

• We first use an FST to convert the lexicon containing the stems and affixes into an intermediate representation
• We then apply a spelling rule that converts the intermediate form into the surface form
• **Parsing**: takes the surface form and produces the lexical representation
• **Generation**: takes the lexical form and produces the surface form
• But how do we handle multiple spelling rules?



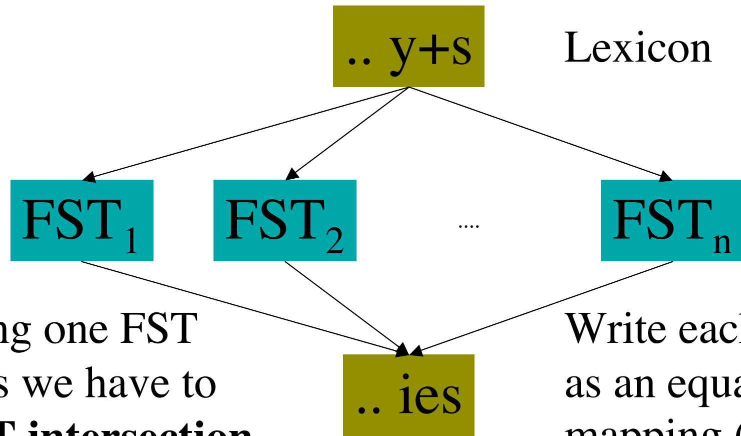
Method 1: Composition

FST composition: creates one FST for all rules

Lexicon

write one FST for each spelling rule: each FST has to provide input to next stage

Method 2: Intersection



Creating one FST implies we have to do **FST intersection** (but there's a catch: *what is it?*)

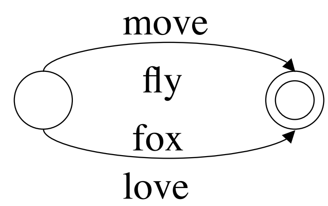
Write each FST as an equal length mapping (ε is taken to be a real symbol)

Intersecting/Composing FSTs

- Implement each spelling rule as a separate FST
- We need slightly different FSTs when using Method 1 (composition) vs. using Method 2 (intersection)
  - In Method 1, each FST implements a spelling rule if it matches, and transfers the remaining affixes to the output (composition can then be used)
  - In Method 2, each FST computes an equal length mapping from input to output (intersection can then be used). Finally compose with lexicon FST and input.
- In practice, composition can create large FSTs

Length Preserving “two-level” FST for *e-deletion*

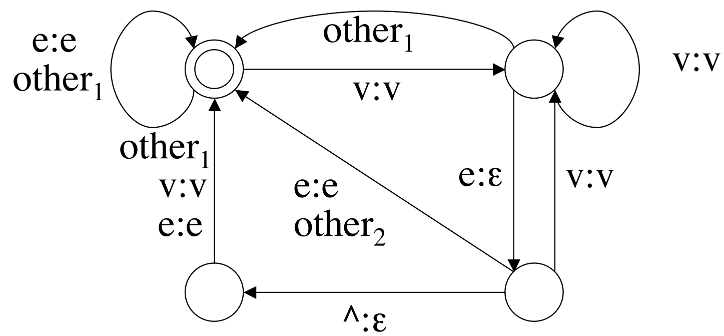
Stems/Lexicon



move ^ ed
movε ε ed

other$_1$ = Σ - {e,v}

other$_2$ = Σ - {e,v,^}



Should also work for leaving :: leave^ing

Rewrite Rules

left context   right context

- Context dependent rewrite rules: $\alpha \rightarrow \beta \,/\, \lambda \,\_\_\, \rho$
  - ($\lambda\ \alpha\ \rho \rightarrow \lambda\ \beta\ \rho$; that is α becomes β in context $\lambda \,\_\_\, \rho$)
  - α, β, λ, ρ are regular expressions, α = input, β = output
- How to apply rewrite rules:
  - Consider rewrite rule: a $\rightarrow$ b / ab __ ba
  - Apply rule on string *abababababa*
  - Three different outcomes are possible:
    - *abbbabbbaba* (left to right, iterative)
    - *ababbbabbba* (right to left, iterative)
    - *abbbbbbbbba* (simultaneous)

Rewrite Rules

$u \rightarrow i \,/\, i\, C^* \,\_\_$

$(u \rightarrow i \,/\, \Sigma^*\, i\, C^* \,\_\_\, \Sigma^*)$

Input: kikukuku

from *(R. Sproat slides)*

Rewrite Rules

$u \rightarrow i \,/\, i\, C^* \,\_\_$

kikukuku
kikukuku
kikikuku
kikikuku
kikikiku
kikikiku
kikikiki

output of one application *feeds* next application

left to right application

Rewrite Rules

$u \rightarrow i \,/\, i\, C^* \_\_$

kikukuku
kikukuku
kikukuku
kikukuku
kikikuku
kikikiku
kikikiki

← *right to left application*

Rewrite Rules

$u \rightarrow i \,/\, i\, C^* \_\_$

kikukuku
kikukuku
kikikuku

simultaneous application
(context rules apply to input string only)

Rewrite Rules

- Example of the e-insertion rule as a rewrite rule:
  $\varepsilon \rightarrow e \,/\, (x \mid s \mid z)\text{^} \_\_ \, s\#$
- Rewrite rules can be optional or obligatory
- Rewrite rules can be ordered wrt each other
- This ensures exactly one output for a set of rules

Rewrite Rules

- Rule 1: iN → im / __ (p | b | m)
- Rule 2: iN → in / __
- Consider input *iNpractical* (N is an abstract nasal phoneme)
- Each rule has to be obligatory or we get two outputs: *impractical* and *inpractical*
- The rules have to be ordered wrt to each other so that we get *impractical* rather than *inpractical* as output
- The order also ensures that *intractable* gets produced correctly

Rewrite Rules

- Under some conditions, these rewrite rules are equivalent to FSTs
- We cannot apply output of a rule as input to the rule itself iteratively:

  $\varepsilon \rightarrow ab$ / a __ b

  If we allow this, the above rewrite rule will produce $a^n b^n$ for $n >= 1$ which is not regular

  Why? Because we rewrite the ε in aεb which was introduced in the previous rule application

  Matching the a__b as left/right context in aεb is ok

Rewrite Rules

- In a rewrite rule: $\alpha \rightarrow \beta$ / λ __ ρ
- Rewrite rules are interpreted so that the **input** α does not match something introduced in the previous rule application
- However, we are free to match the **context** either λ or ρ or both with something introduced in the previous rule application (see previous examples)
- In this case, we can convert them into FSTs

Rewrite rules to FSTs

$u \rightarrow i$ / Σ^* i C^* __ Σ^* (example from R. Sproat's slides)

- Input: kikukupapu (use left-right iterative matching)
- Mark all possible right contexts

  $> k > i > k > u > k > u > p > a > p > u >$
- Mark all possible left contexts

  $> k > i <> k <> u > k > u > p > a > p > u >$
- Change u to i when delimited by <>

  $> k > i <> k <> i > k > u > p > a > p > u >$
- But the next u is not delimited by <> and so cannot be changed even though the rule matches

Rewrite rules to FSTs

$u \rightarrow i$ / Σ^* i C^* __ Σ^*

- Input: kikukupapu
- Mark all possible right contexts

  $> k > i > k > u > k > u > p > a > p > u >$
- Mark all *u* followed by > with $<_1$ and $<_2$

  $k > i > k <_1 > u > k <_1 > u > p > a > p <_1 > u >$

  $<_2$ u $<_2$ u $<_2$ u

  Short-hand for multiple paths in the FST
- Change all *u* to *i* when delimited by $<_1$ >

  $k > i > k <_1 > i > k <_1 > i > p > a > p <_1 > i >$

  $<_2$ u $<_2$ u $<_2$ u

$u \rightarrow i \,/\, \Sigma^* \, i \, C^* \; \_\_ \; \Sigma^*$

Rewrite rules to FSTs

$k > i > k <_1 > i > k <_1 > i > p > a > p <_1 > i >$

$\quad <_2 \quad u \qquad <_2 \; u \qquad\qquad <_2 \quad u$

- Delete >

$k \; i \; k <_1 i \; k <_1 i \; p \; a \; p <_1 i$

$\quad <_2 u \quad <_2 u \qquad <_2 u$

- Only allow *i* where $<_1$ is preceded by iC*, delete $<_1$

k i k i k i p a p

$\quad <_2 u \quad <_2 u \qquad <_2 u$

- Allow only strings where $<_2$ is **not** preceded by iC*, delete $<_2$

k i k i k i p a p u

Rewrite Rules to FST

$a \rightarrow b \,/\, b \; \_\_ \; b$
$b \rightarrow a \,/\, b \; \_\_ \; b$

Input: ababb

- Mark right contexts: a > b a > b > b
- Mark a and b before > with $<_1$ and $<_2$

$<_1 a > b <_1 a > <_1 b > b$

$<_2 a \quad <_2 a \quad <_2 b$

- Match $<_1$ LHS > and convert to $<_1$ RHS >; delete >

$<_1 b \; b <_1 b <_1 a \; b$

$<_2 a \quad <_2 a <_2 b$

- Allow $<_1$ RHS when left context exists; delete $<_1$

$<_1 b \; b <_1 b <_1 a \; b \quad = <_2 a \; b \; (b \mid <_2 a) \; (a \mid <_2 b) \; b$

$<_2 a \quad <_2 a <_2 b$

- Allow $<_2$ LHS when left context does not exist; delete $<_2$

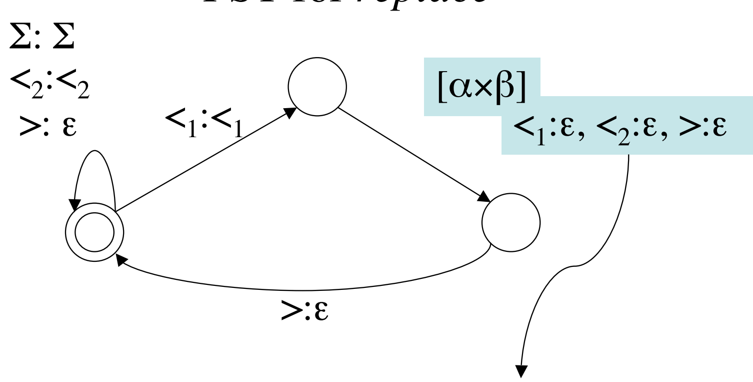
a b b a b

Rewrite rules to FST

- For every rewrite rule: $\alpha \rightarrow \beta$ / λ __ ρ:
  - FST *r* that inserts > before every ρ
  - FST *f* that inserts $<_1$ & $<_2$ before every α followed by >
  - FST *replace* that replaces α with β between $<_1$ and > and deletes >
  - FST λ_1 that only allows all $<_1 \beta$ preceded by λ and deletes $<_1$
  - FST λ_2 that only allows all $<_2 \beta$ **not** preceded by λ and deletes $<_2$
- Final FST = $r \circ f \circ replace \circ \lambda_1 \circ \lambda_2$
- This is only for left-right iterative obligatory rewrite rules: similar construction for other types

Rewrite Rules to FST

FST for *replace*



Create a new FST by taking the cross product of the languages α and β and each state of this new FST: $[\alpha \times \beta]$ has loops for the transitions $<_1:\varepsilon$, $<_2:\varepsilon$, $>:\varepsilon$

Ambiguity (in parsing)

- Global ambiguity: (de+light+ed *vs*. delight+ed)

  foxes → fox+N+PL (*I saw two foxes*)

  foxes → foxes+V+3SG (*Clouseau foxes them again*)

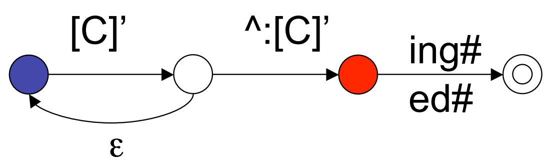
- Local ambiguity:

  assess has a prefix string *asses* that has a valid analysis:
  asses → ass+N+PL

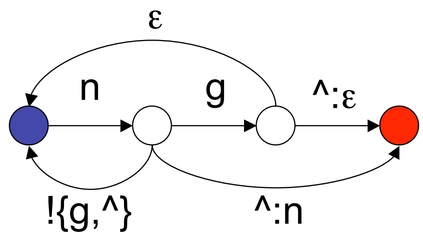
- Global ambiguity results in two valid answers, but local ambiguity returns only one.
- However, local ambiguity can also slow things down since two analyses are considered partway through the string.

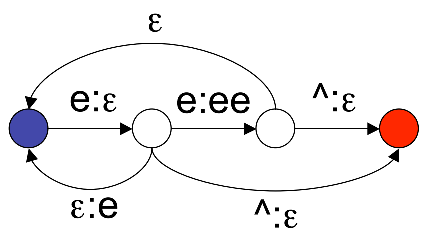
Summary

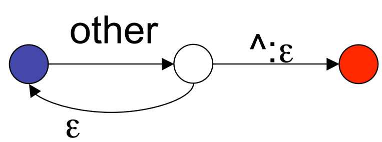
- FSTs can be applied to creating lexicons that are aware of morphology
- FSTs can be used for simple stemming
- FSTs can also be used for morphographemic changes in words (spelling rules), e.g. fox+N+PL becomes foxes
- Multiple FSTs can be composed to give a single FST (that can cover all spelling rules)
- Multiple FSTs that are length preserving can also be run in parallel with the intersection of the FSTs
- Rewrite rules are a convenient notation that can be converted into FSTs automatically
- Ambiguity can exists in the lexicon: both global & local



[C]' = [C]-{n}







other = Σ-[C]'-{n,e}