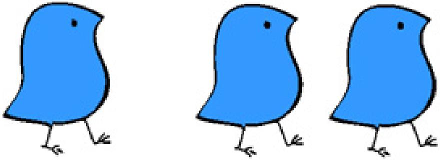
English Morphology

Linguistics 409 · Computational Linguistics

Rice University

January 28, 2013



This week

- Today: English Morphology
- Friday: Finite-State Transducers

Words

- Finite-state methods are particularly useful in dealing with a lexicon
- Many devices, most with limited memory, need access to large lists of words
- And they need to perform fairly sophisticated tasks with those lists
- So first we'll talk about some facts about words and then come back to computational methods

English Morphology

- Morphology is the study of the ways that words are built up from smaller meaningful units called *morphemes*
- We can usefully divide morphemes into two classes

1. *Stems:* The core meaning-bearing units
2. *Affixes:* Bits and pieces that adhere to stems to change their meanings and grammatical functions

Classes of morpheme

- We can further divide morphology up into two broad classes
  1. Inflectional
  2. Derivational
- Parametric variation; Cliticization
- Non-Concatenative Morphology; Arabic and Hebrew

Morphological ‘types’ of Languages

Degree of Synthesis

Average number of morphemes per word.

- isolating
- synthetic
- polysynthetic

Degree of Fusion

How easy it is to find morpheme boundaries.

- agglutinative (discrete)
- fusional (inseparable)

Inflectional Morphology

Inflectional morphology concerns the combination of stems and affixes where the resulting word:

- Has the same word class as the original
- Serves a grammatical/semantic purpose that is
  - Different from the original
  - But is nevertheless transparently related to the original

Word Classes

- By word class, we have in mind familiar notions like noun and verb
- We'll go into more detail in SLP Chapter 5
- Right now we're concerned with word classes because the way that stems and affixes combine is often based to a large degree on the word class of the stem

English inflection is boring

English presents, in some ways, a very simple morphology problem

- English nouns are simple: markers for plural and possessive
- Verbs are only slightly more complex: Markers appropriate to the tense of the verb

The Eight English Inflectional Morphemes

1. *plural -s*: cows, dishes, wugz
2. *possessive -s*: ‘Mary’s book’
3. *3rd singular present -s*: ‘he bores us’
4. *progressive -ing*: ‘she is singing’
5. *past -ed*: walked, jumped
6. *past participle -en*: beaten, given, hidden
7. *comparative -er*: taller, cleaner, happier
8. *superlative -est*: tallest, cleanest, happiest

Regulars and Irregulars

Things are made slightly more interesting by the fact that some words pattern differently.

Regular words follow productive grammatical rules, *irregular* words do not.

- ring/rang/rung, sing/sang/sung (ablaut)
- Mouse/mice, goose/geese, fall/fell (umlaut)
- ox/oxen, deer/deer
- Go/went, fly/flew

Regular and Irregular Verbs

Regular

- Walk, walks, walking, walked, walked
- stem, -s form (habitual present), -ing participle (progressive), -ed (perfect, passive)

Irregulars

- Eat, eats, eating, ate, eaten; preterite past
- Catch, catches, catching, *caught*, *caught*
- Cut, cuts, cutting, *cut*, *cut*

Inflectional Morphology

- So inflectional morphology in English is fairly straightforward
- But is complicated by the fact that there are irregularities

Derivational Morphology

Derivational morphology is known for:

- Quasi-systematicity
- Irregular meaning change
- Changes of word class

Derivational Examples

Verbs and Adjectives to Nouns

-ation	computerize	computerization
-ee	appoint	appointee
-er	kill	killer
-ness	fuzzy	fuzziness

Derivational Examples

Nouns and Verbs to Adjectives

-al	computation	computational
-able	embrace	embraceable
-less	clue	clueless

Example: Compute

Many paths are possible...

- Start with *compute*
  - Compute → Computer → computerize → computerization
  - Compute → Computer → computerize → computerizable
- But not all paths/operations are equally good (grammatical)
  - Clue → *clueable

Example: Compute

Many paths are possible...

- Start with *compute*
  - Compute → Computer → computerize → computerization
  - Compute → Computer → computerize → computerizable
- But not all paths/operations are equally good (grammatical)
  - Clue → *clueable

Example: Compute

Many paths are possible...

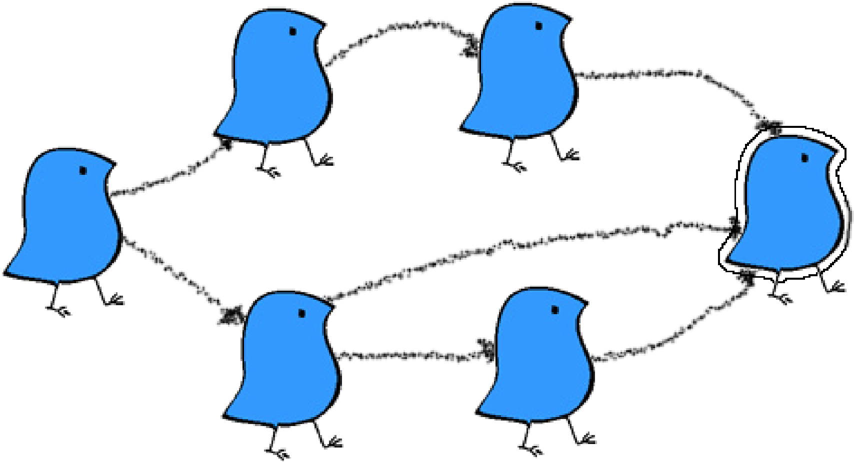
- Start with *compute*
  - Compute → Computer → computerize → computerization
  - Compute → Computer → computerize → computerizable
- But not all paths/operations are equally good (grammatical)
  - Clue → *clueable

Example: Compute

Many paths are possible...

- Start with *compute*
  - Compute → Computer → computerize → computerization
  - Compute → Computer → computerize → computerizable
- But not all paths/operations are equally good (grammatical)
  - Clue → *clueable

Morpholgy + FSAs



Morpholgy and FSAs

Morphology and FSAs

We'd like to use the machinery provided by FSAs to capture these facts about morphology

- Accept strings that are in the language
- Reject strings that are not
- And do so in a way that doesn't require us to carry around a giant dictionary

Finite State Morphological Parsing

English		Spanish		
Input	**Morphological Parse**	**Input**	**Morphological Parse**	**Gloss**
cats	cat +N +PL	pavos	pavo +N +Masc +Pl	'ducks'
cat	cat +N +SG	pavo	pavo +N +Masc +Sg	'duck'
cities	city +N +Pl	bebo	beber +V +PInd +1P +Sg	'I drink'
geese	goose +N +Pl	canto	cantar +V +PInd +1P +Sg	'I sing'
goose	goose +N +Sg	canto	canto +N +Masc +Sg	'song'
goose	goose +V	puse	poner +V +Perf +1P +Sg	'I was able'
gooses	goose +V +3P +Sg	vino	venir +V +Perf +3P +Sg	'he/she came'
merging	merge +V +PresPart	vino	vino +N +Masc +Sg	'wine'
caught	catch +V +PastPart	lugar	lugar +N +Masc +Sg	'place'
caught	catch +V +Past			

J&M Figure 3.2

Applications

Applications

- The kind of parsing we're talking about is normally called **morphological analysis**
- It can either be an important stand-alone component of many applications (spelling correction, information retrieval, machine translation, etc.)
- Simply a link in a chain of further analysis (e.g. part of speech tagging for syntactic parsing)
- Or as a tool to make linguists' jobs easier (e.g. providing automated interlinear glosses, part of speech tags for corpus analysis, etc.)

Morphological Parsing

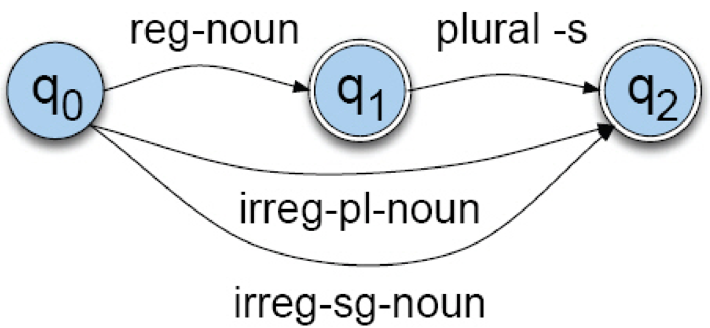
Ultimately we will need:

1. A **lexicon** of stems and affixes in our language.
2. A model of **morphotactics** to tell us how morphemes are ordered and connected in the language, and
3. A model of the **orthographic** conventions followed in the text we're analyzing.

Let's Start Simple: English Plurals

- Regular singular nouns are ok
- Regular plural nouns have an -s on the end
- Irregulars are ok as is

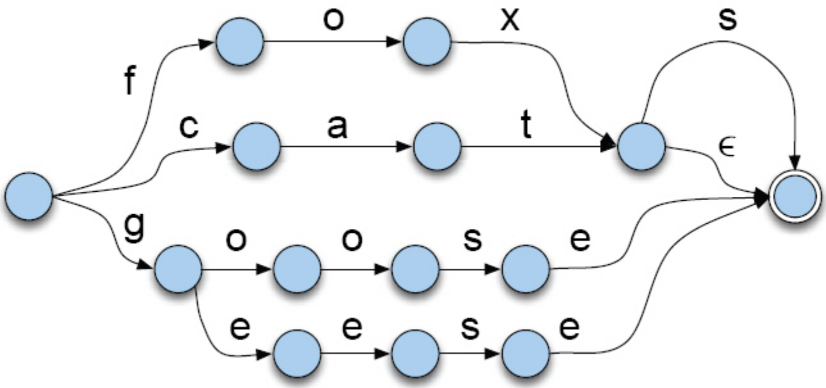
Simple Rules



J&M Figure 3.3

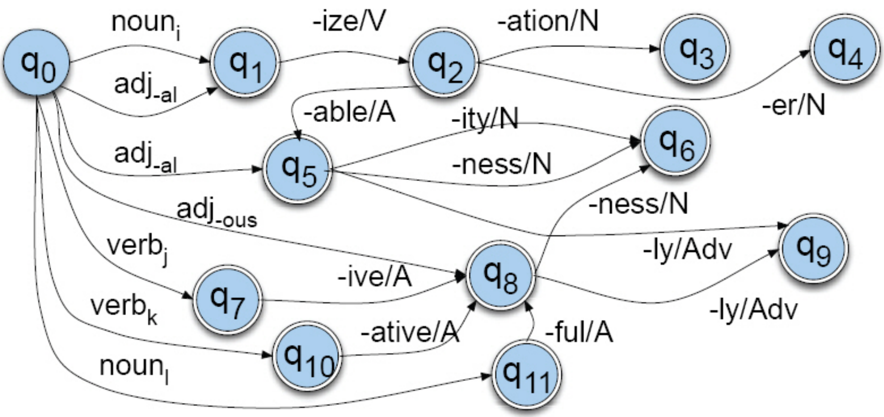
reg-noun	irreg-pl-noun	irreg-sg-noun	plural
fox	geese	goose	-s
cat	sheep	sheep	
aardvark	mice	mouse	

Morphological Recognition



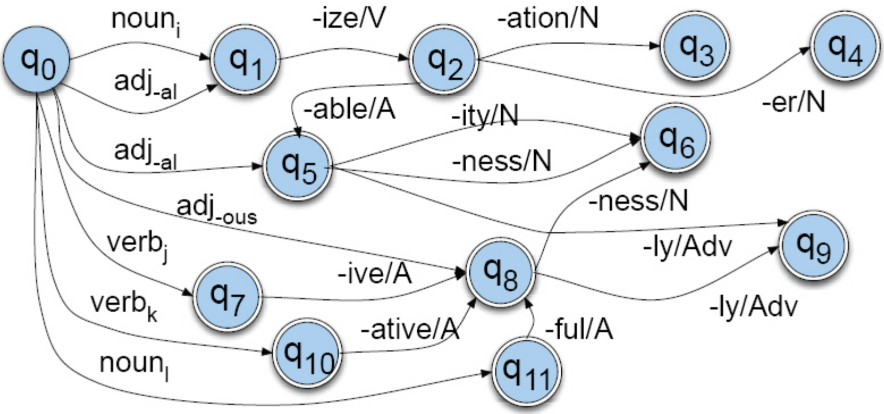
J&M Figure 3.7

Derivational Rules



J&M Figure 3.7

Derivational Rules



If everything is an accept state how do things ever get rejected?

Parsing/Generation vs. Recognition

- We can now run strings through these machines to recognize strings in the language. What are some applications for this technology?
- Often if we find some string in the language we might also like to assign a structure to it (parsing)
- Or we might have some structure and we want to produce a surface form for it (production/generation)

Example:

From “cats” to “cat +N +PL”
From “pickled to “pickle +V +PST

Finite State Transducers

Finite State Transducers define a relation between two sets of strings. These can be used for recognition, generation, relation, or (as we'll use them) translation. Schematically:

- Add a second tape to represent the second set of strings.
- Add extra symbols to the transitions (i.e. a:x instead of just a))
- On one tape we *read* "cats", on the other we *write* "cat +N +PL"
- Through a property called **inversion**, an FST morphological parser T can become an FST morphological generator $T^{-}1$ (or vice versa)
- Through a property called **composition**, FSTs can be chained together (see 'multiple tape machines' later)

Finite State Transducers

Finite State Transducers define a relation between two sets of strings. These can be used for recognition, generation, relation, or (as we'll use them) translation. Schematically:

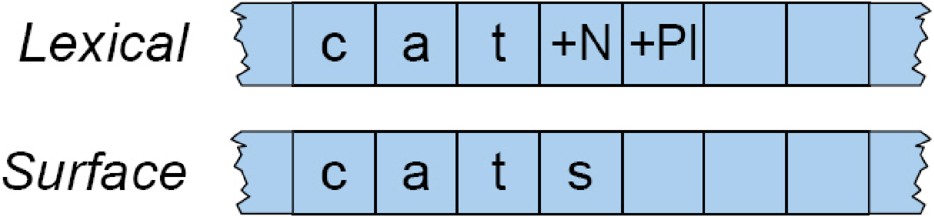
- Add a second tape to represent the second set of strings.
- Add extra symbols to the transitions (i.e. a:x instead of just a))
- On one tape we *read* "cats", on the other we *write* "cat +N +PL"
- Through a property called **inversion**, an FST morphological parser T can become an FST morphological generator T^{-1} (or vice versa)
- Through a property called **composition**, FSTs can be chained together (see 'multiple tape machines' later)

Finite State Transducers

Finite State Transducers define a relation between two sets of strings. These can be used for recognition, generation, relation, or (as we'll use them) translation. Schematically:

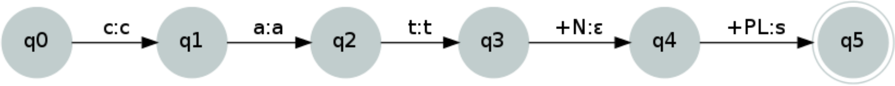
- Add a second tape to represent the second set of strings.
- Add extra symbols to the transitions (i.e. a:x instead of just a))
- On one tape we *read* "cats", on the other we *write* "cat +N +PL"
- Through a property called **inversion**, an FST morphological parser T can become an FST morphological generator $T^{-}1$ (or vice versa)
- Through a property called **composition**, FSTs can be chained together (see 'multiple tape machines' later)

Modelling Morphotactics: FSTs



J&M Figure 3.7

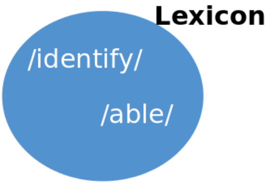
FST Transitions



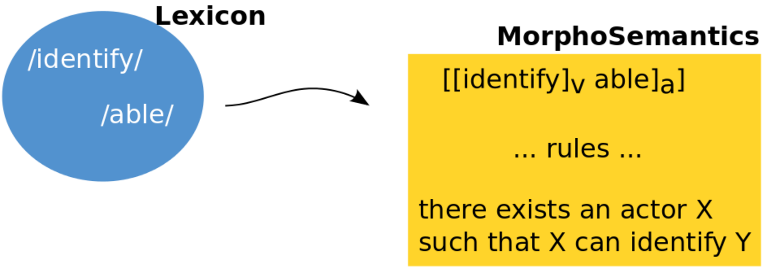
FSA with FST transitions

- c:c means read a c on one tape and write a c on the other
- +N:ϵ means read a +N symbol on one tape and write nothing on the other (notice that +N doesn't align with anything?)
- +PL:s means read +PL and write an s

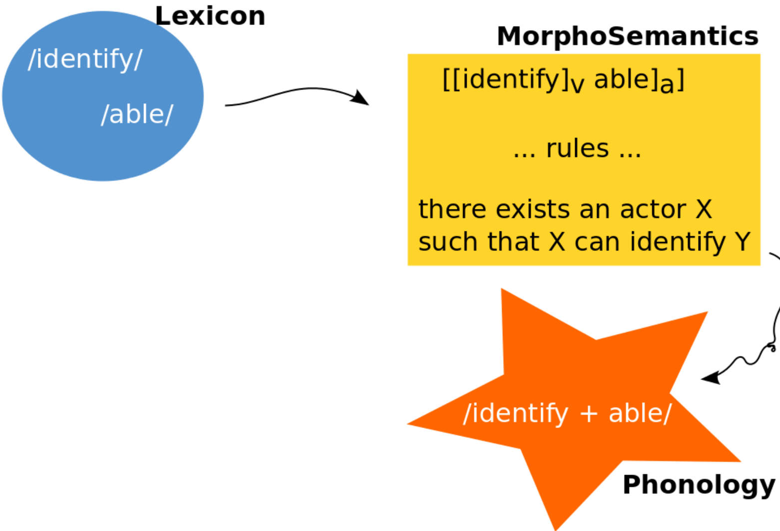
Morphology: a traditional view



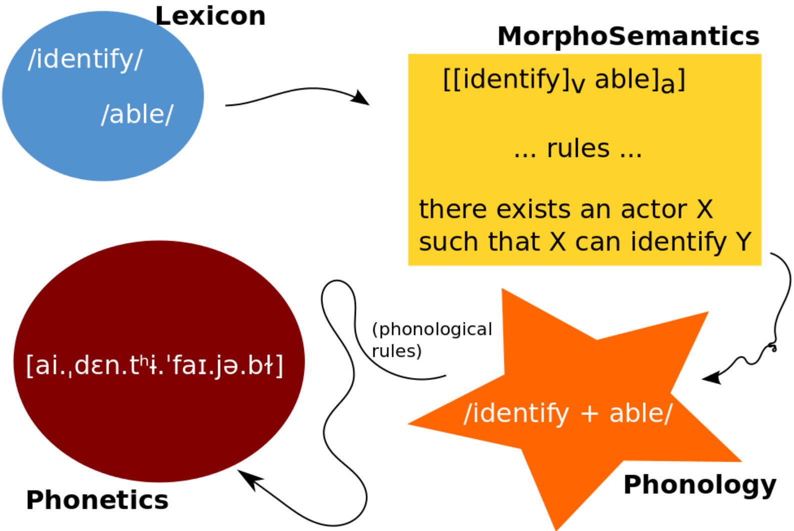
Morphology: a traditional view



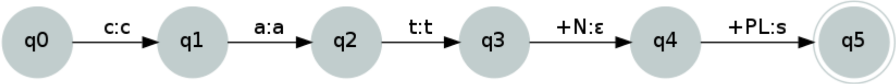
Morphology: a traditional view



Morphology: a traditional view



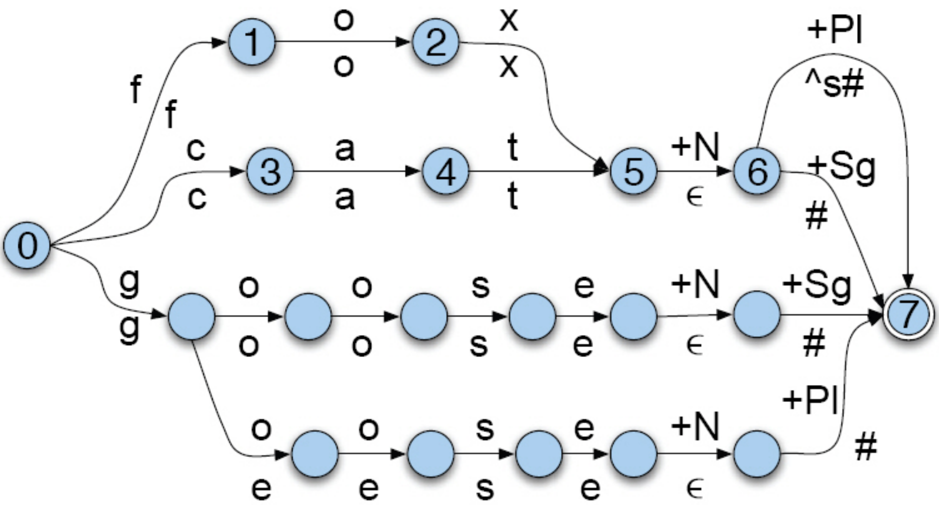
FST Transitions



FSA with feasible pairs

- We can also call these transitions like c:c or +N:ϵ**feasible pairs**
- A pair like c:c where c in one alphabet maps to c in the other can also be written as simply c.

Modelling Morphotactics: English Nominal Inflection



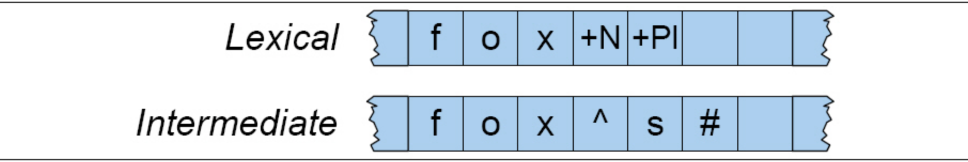
J&M Figure 3.14

For next time:

For next time:

1. Monday: **Stemming, Spelling, Edit Distance**
2. **Read** the rest of Chapter 3 in SLP

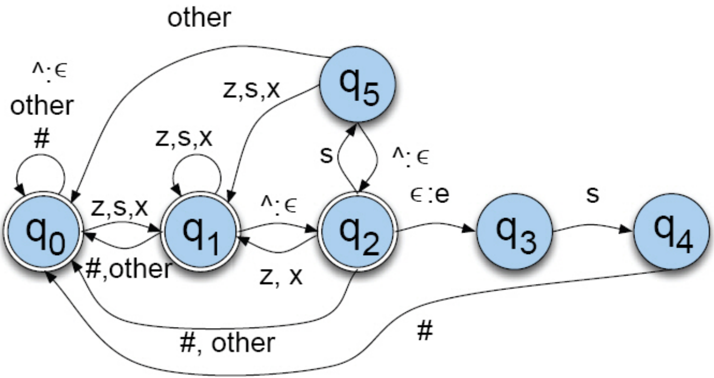
FST: Orthography Rules



J&M Figure 3.15

- So *now* all we have to do is get the orthography right. In the traditional view, this step is analogous to phonology.
- And, like traditional phonology, we're going to write a giant pile of rules (and then implement them as finite state transducers).
- e.g. $\epsilon \rightarrow$ e / ^ __ #

FST: Orthography Rules



J&M Figure 3.17

FST: Orthography Rules

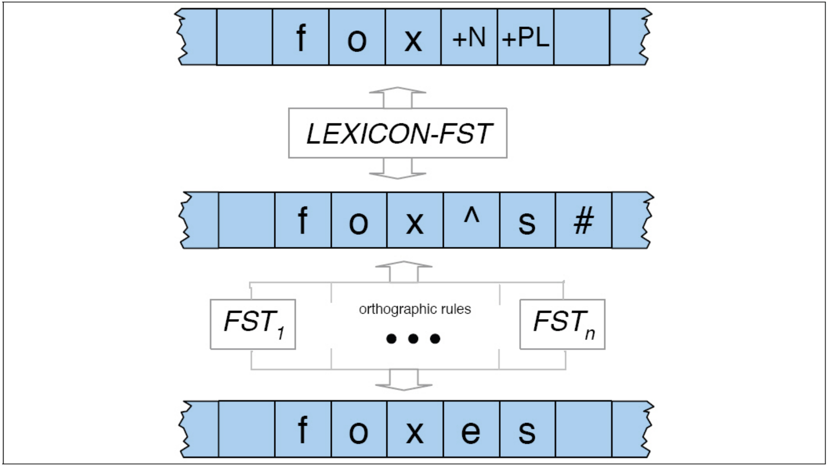
Name	Description of Rule	Example
Consonant doubling	1-letter consonant doubled before *-ing/-ed*	beg/begging
E deletion	silent e dropped before *-ing* and *-ed*	make/making
E insertion	e added after *-s*,*-z*,*-x*,*-ch*, *-sh* before *-s*	watch/watches
Y replacement	*-y* changes to *-ie* before *-s*, *-i* before *-ed*	try/tries
K insertion	verbs ending with *vowel* + *-c* add *-k*	panic/panicked

J&M Spelling Rule Examples

Putting it all together

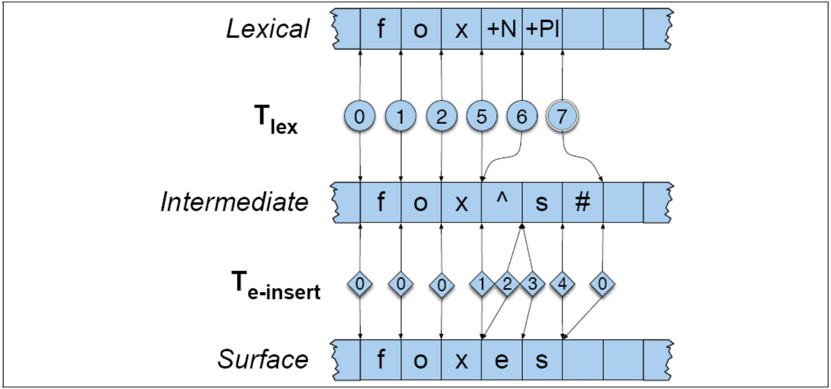
- Finally, we combine our lexical, morphotactic, and orthographic models
- This *cascade* is conceptually identical to the pipelines we built in UNIX last week.
- And will be a common design pattern throughout computational linguistics.

Putting it all together



J&M Figure 3.19

Putting it all together: accepting foxes



J&M Figure 3.20

Porter Stemmer

- The FSTs we have been looking at are an elegant way to handle morphological analysis somewhat intelligently
- But sometimes you don't *need* intelligence. Sometimes a not very good solution is good enough.
- Times like these call for the **Porter stemmer**.
- This is another example of the precision vs recall tradeoff. What are some applications where (inexpensive) recall is more important than (expensive) precision?

Porter Stemmer

- The FSTs we have been looking at are an elegant way to handle morphological analysis somewhat intelligently
- But sometimes you don't *need* intelligence. Sometimes a not very good solution is good enough.
- Times like these call for the **Porter stemmer**.
- This is another example of the precision vs recall tradeoff. What are some applications where (inexpensive) recall is more important than (expensive) precision?

Porter Stemmer

- The Porter stemmer (Porter 1980) is a cascading set of regular expressions that attempts to extract stems from a text.
- A stemmer stems stems (deaffixes affixes) by attempting to match (and undo) patterns of affixation.
- **Q:** If we do this using regular expressions, how does it differ from the Finite State Transducers we have been discussing?

Porter Stemmer

- The Porter stemmer (Porter 1980) is a cascading set of regular expressions that attempts to extract stems from a text.
- A stemmer stems stems (deaffixes affixes) by attempting to match (and undo) patterns of affixation.
- **Q:** If we do this using regular expressions, how does it differ from the Finite State Transducers we have been discussing?

Porter Stemmer: Example errors

Errors of Commission		Errors of Omission	
organization	organ	European	Europe
doing	doe	analysis	analyzes
numerical	numerous	noise	noisy
policy	police	sparse	sparsity

J&M p. 68

Tokenization

- Tokenization is very often the first step in any CL task.
- (so errors introduced here cast a shadow over everything else you do with the data).
- We have already tried tokenizing a text (poorly):

```
% convert spaces to new lines
sed -e 's/\s\+/\n/g' file.txt > wordlist.txt
```

```perl
#!/usr/bin/perl

$letternumber = "[A-Za-z0-9]";
$notletter = "[^A-Za-z0-9]";
$alwayssep = "[\\?!()\";/\\|‘]";
$clitic = "('|:|-|'S|'D|'M|'LL|'RE|'VE|N'T|'s|'d|'m|'ll|'re|'ve|n't)";

$abbr{"Co."} = 1; $abbr{"Dr."} = 1; $abbr{"Jan."} = 1; $abbr{"Feb."} = 1;

while ($line = <>){ # read the next line from standard input

    # put whitespace around unambiguous separators
    $line =~ s/$alwayssep/ $& /g;

    # put whitespace around commas that aren't inside numbers
    $line =~ s/([^0-9]),/$1 , /g;
    $line =~ s/,([^0-9])/ , $1/g;

    # distinguish singlequotes from apostrophes by
    # segmenting off single quotes not preceded by letter
    $line =~ s/^'/$& /g;
    $line =~ s/($notletter)'/$1 '/g;

    # segment off unambiguous word-final clitics and punctuation
    $line =~ s/$clitic$/ $&/g;
    $line =~ s/$clitic($notletter)/ $1 $2/g;

   # now deal with periods.  For each possible word
   @possiblewords=split(/\s+/,$line);
   foreach $word (@possiblewords) {
      # if it ends in a period,
      if (($word =~ /$letternumber\./)
             && !($abbr{$word})  # and isn't on the abbreviation list
                # and isn't a sequence of letters and periods (U.S.)
                # and doesn't resemble an abbreviation (no vowels: Inc.)
             && !($word =~
                 /^([A-Za-z]\.([A-Za-z]\.)+|[A-Z][bcdfghj-nptvxz]+\.)$/)) {
          # then segment off the period
          $word =~ s/\.$/ \./;
      }
      # expand clitics
      $word =~s/'ve/have/;
      $word =~s/'m/am/;
      print $word," ";
   }
 print "\n";
}
```

String Distance: Levenshtein Distance

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

J&M Figure 3.23

- Begin with two **aligned** strings
- cost 1 for insertions
- cost 1 for deletions
- (therefore substitutions cost 2)
- it often makes sense to have the cost depend on the characters involved (e.g. qwerty for typos, confusability matrices for speech, etc.).

String Distance: Levenshtein Distance

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

J&M Figure 3.23

- Begin with two **aligned** strings
- cost 1 for insertions
- cost 1 for deletions
- (therefore substitutions cost 2)
- it often makes sense to have the cost depend on the characters involved (e.g. qwerty for typos, confusability matrices for speech, etc.).

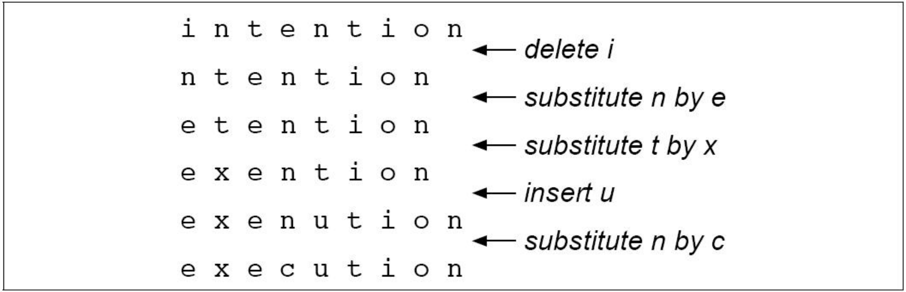
Edit Distance: intention to execution

```
i n t e n t i o n
                    ← delete i
n t e n t i o n
                    ← substitute n by e
e t e n t i o n
                    ← substitute t by x
e x e n t i o n
                    ← insert u
e x e n u t i o n
                    ← substitute n by c
e x e c u t i o n
```

J&M Figure 3.24

- We are going to use the hugely important **dynamic programming** approach.
- Dynamic programming is badly named!
- Basically it means you solve all of the sub-parts of a problem only once, rather than recalculate them many times.

Edit Distance: Dynamic Programming



J&M Figure 3.24

- Think back to our discussion of search (depth first vs breadth first) for FSAs
- We *could* calculate the Levenshtein distance between our two strings one character at a time
- But then we'd end up doing the same calculations many times.

Edit Distance: Pseudocode

function MIN-EDIT-DISTANCE(*target*, *source*) **returns** *min-distance*

$n \leftarrow$ LENGTH(*target*)
$m \leftarrow$ LENGTH(*source*)
Create a distance matrix *distance[n+1,m+1]*
Initialize the zeroth row and column to be the distance from the empty string
  distance[0,0] = 0
  for each column i **from** 1 **to** n **do**
    distance[i,0] $\leftarrow$ *distance*[i-1,0] + *ins-cost*(*target*[i])
  for each row j **from** 1 **to** m **do**
    distance[0,j] $\leftarrow$ *distance*[0,j-1] + *del-cost*(*source*[j])
for each column i **from** 1 **to** n **do**
  for each row j **from** 1 **to** m **do**
    $distance[i,j] \leftarrow \text{MIN}(\ distance[i-1,j] + ins\text{-}cost(target_{i-1}),$
    $distance[i-1,j-1] + sub\text{-}cost(source_{j-1},]target_{i-1}),$
    $distance[i,j-1] + del\text{-}cost(source_{j-1}))$
return *distance*[n,m]

J&M Figure 3.25

Edit Distance: Trace

n	*9*	8	9	10	11	12	11	10	9	8
o	*8*	7	8	9	10	11	10	9	8	9
i	*7*	6	7	8	9	10	9	8	9	10
t	*6*	5	6	7	8	9	8	9	10	11
n	*5*	4	5	6	7	8	9	10	11	10
e	*4*	3	4	5	6	7	8	9	10	9
t	*3*	4	5	6	7	8	7	8	9	8
n	*2*	3	4	5	6	7	8	7	8	7
i	*1*	2	3	4	5	6	7	6	7	8
#	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
	#	**e**	**x**	**e**	**c**	**u**	**t**	**i**	**o**	**n**

J&M Figure 3.26

Edit Distance: Path

n	9	↓ 8	↙←↓ 9	↙←↓ 10	↙←↓ 11	↙←↓ 12	↓ 11	↓ 10	↓ 9	**↙ 8**
o	8	↓ 7	↙←↓ 8	↙←↓ 9	↙←↓ 10	↙←↓ 11	↓ 10	↓ 9	**↙ 8**	← 9
i	7	↓ 6	↙←↓ 7	↙←↓ 8	↙←↓ 9	↙←↓ 10	↓ 9	**↙ 8**	← 9	← 10
t	6	↓ 5	↙←↓ 6	↙←↓ 7	↙←↓ 8	↙←↓ 9	**↙ 8**	← 9	← 10	←↓ 11
n	5	↓ 4	↙←↓ 5	↙←↓ 6	↙←↓ 7	**↙←↓ 8**	↙←↓ 9	↙←↓ 10	↙←↓ 11	↙↓ 10
e	4	↙ 3	← 4	**↙← 5**	**← 6**	← 7	←↓ 8	↙←↓ 9	↙←↓ 10	↓ 9
t	3	↙←↓ 4	**↙←↓ 5**	↙←↓ 6	↙←↓ 7	↙←↓ 8	↙ 7	←↓ 8	↙←↓ 9	↓ 8
n	2	**↙←↓ 3**	↙←↓ 4	↙←↓ 5	↙←↓ 6	↙←↓ 7	↙←↓ 8	↓ 7	↙←↓ 8	↙ 7
i	**1**	↙←↓ 2	↙←↓ 3	↙←↓ 4	↙←↓ 5	↙←↓ 6	↙←↓ 7	↙ 6	← 7	← 8
#	**0**	1	2	3	4	5	6	7	8	9
	#	**e**	**x**	**e**	**c**	**u**	**t**	**i**	**o**	**n**

J&M Figure 3.27

For next time:

For next time:

1. Wednesday: **Probability for Linguists**
2. **Read:** Abney & (optionally) Goldsmith (both on OwlSpace)