Walk-ing the walk, talk-ing the talk – My Fair Lady Lecture

Professor Robert C. Berwick
berwick@csail.mit.edu
The Menu Bar

• Administrivia: read the instructions before opening; Lab 2 due next Monday at 6 pm; Lab 3 released Monday evening.
• My Fair Lady: “words, words, words, I’m so sick of words…”
• A detailed walk-through of a 2-level example
• What’s in Laboratory 3?
• What can the two-level (finite-state) transducers do?
• What can’t fst’s do? Complexity issues & representational issue: decentralize the method?!
• The coup de…Bambarra
The story so far...

- We divide morpho-syntax, word parsing knowledge, into 2 parts: (1) Lexicon (roots+endings); and (2) spelling changes;
- We further divided spelling changes into a (small) set
- We implemented both as finite state transducers (FSTs), to capture (1) output ‘glosses’ in the case of the lexicon; and (2) lexical/surface pairings in the case of spelling changes
- We showed that FSTs don’t behave quite like FSAs
- We implement their action together as a intersected FST, all in the system demo’d last time (‘kimmo’)

Today…

- We’ll trace through an example in detail, to show how both components work
- We will see that both lexicon and spelling change rules invoke nondeterminism in essential (though slightly different) ways
- We will see how spelling change rules must interact; showing how multiple spelling changes go together
- We will add 1 more bit about how FSTs aren’t like finite-state automata
- We will show how to write a new spelling change rule
- We will see how all this fits together in the next Lab
- We’ll probe the computational complexity of this system and the strengths & weaknesses of this way of parsing words
AUTUMN: A TIME FOR REFLECTION

AUTUMN... I WONDER WHY THE "N" IS SILENT.
Morphology is finite-state: regular relations

A finite-state transducer $T$ (FST) is a sextuple, $(Q, \Sigma_1, \Sigma_2, \delta, I, F)$ where:

1. $Q$ is a finite set of states;
2. $\Sigma_1$ is a finite set of input symbols;
3. $\Sigma_2$ is a finite set of output symbols;
4. $\delta \subseteq Q \times \Sigma_1^* \times \Sigma_2^* \times Q$ is the transition mapping;
5. $I \subseteq Q$ is a finite set of initial states;
6. $F \subseteq Q$ is a finite set of final states

$T$ defines the regular relation $R(T)$, the set of pairs $(x, y)$ s.t. $\delta^*(x, y) \subseteq F$
Our implementation is a so-called **two-level** system

Each spelling change FST must pass all pairs of lexical, surface character pairs: thus, the FSTs are really acting as constraint filters (they are failure driven)

This is the intersection of all the FSTs

Set of parallel two-level rules (constraints) compiled into finite-state machines interpreted as transducers; one fst is a ‘letter tree’ for the roots+affixes
The “same length” constraint

- So that FSTs are closed under intersection

The same-length constraint
How are underlying/surface pairings done in linguistic theory?

- Insert e after ‘sh’, ‘x’, etc: “epenthesis”
- Statement of rule is actually quite complex:
  - Rewrite rule: $x \rightarrow y | \alpha \ _\beta$ (Chomsky & Halle, 1968)
  - $0:e \rightarrow [Csib (c h) (s h) y:i] +:0 _ s$
    (transducer notation)

Note that the re-write notation is very powerful (in general, an arbitrary Turing machine)

So the problem always was, how to implement this efficiently

This was not solved until the 2-level method was invented
How hairy can these rules be, after all?

(1) $\text{Bn}+\text{a}--\text{a} \rightarrow \text{Sn}+\text{a}--\text{a}$; ultimately, "bana'ī"; 
but
$\text{zN}+\text{a}--\text{a} \rightarrow \text{zN}+\text{a}--\text{a}$; but is ultimately
"nâkây" (see MR10).

(2) $\text{b}--\text{Pek}+\text{a}--\text{a} \rightarrow \text{b}--\text{Pek}+\text{a}--\text{a}$; ultimately, "lifēk".
From underlying forms bubbling to the surface (from Halle, 1960)...

\[ \text{accede} \quad \text{recede} \quad \text{assign} \quad \text{resign} \]

<table>
<thead>
<tr>
<th></th>
<th>R1 dupe</th>
<th>R2 s-to-z</th>
<th>R3 k-to-s</th>
<th>R4 Vowel</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a+k ked</td>
<td>a+k ked</td>
<td>a+k ked</td>
<td>akseyd</td>
</tr>
<tr>
<td></td>
<td>re+k ed</td>
<td>a+ssin re+zin</td>
<td>a+s sin re+z in</td>
<td>reseyd</td>
</tr>
<tr>
<td></td>
<td>a+sin</td>
<td>re+sin</td>
<td>re+sin</td>
<td>assayn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>reziyn</td>
</tr>
<tr>
<td>Shift</td>
<td>accede</td>
<td>recede</td>
<td>assign</td>
<td>resign</td>
</tr>
</tbody>
</table>
4 ordered rules - write out lexical:surface pairs (L:S pairs)

- Lexical:  a+ked re+ked a+sin re+sin
  Surface:  akseyd reseyd assayn rezayn

Pad out so both of equal length, also noting +:0 correspondence

  a+ked   re+ked   a+sin   re+sin
  aksed   re0sed   assin   re0zin
Extract contexts to find declarative constraints

\[
\begin{align*}
\text{a+k ed} & \quad \text{re+k ed} & \quad \text{a+sin} & \quad \text{re+sin} \\
\text{a+sed} & \quad \text{re0sed} & \quad \text{assin} & \quad \text{re0zin}
\end{align*}
\]

Rule: \[+:k \iff \text{a:a} \_\_ \text{k:s}\]
\[+:s \iff \text{a:a} \_\_ \text{s:s}\]

Rule: \[\text{k:s} \iff e\ +:0 \_\_ \text{V} \mid +:k \_\_ \text{V}\]
\[+:s \iff \text{a:a} \_\_ \text{s:s}\]

6.863J/9.611J SP11
Why don’t we need to look at the derivational steps now??

- We can look at the lexical:surface substrings simultaneously
- So, if a rule has applied (conversely, not applied), its effects should be visible via the joint lexical:surface pairs surrounding it in some lexical:surface pairing example (or else not visible if the rule did not apply)
- Otherwise, the rule must have been superfluous (it has no visible effects on the relation between any lexical:surface pairs)
Examples for ‘e’ insertion...

Fox - foxes; church - churches; bus-buses

What elses?

Must look at non-examples as well... as - ases?

What is the rule?

In traditional ‘rewrite form’:

$$\varepsilon \rightarrow e / \{x,s,z\} + ___ s #$$

Now redo this in terms of (lexical, surface) pairs, which tells us how to build the transducer:

Lexical nothing (0) is paired with e, or 0:e in context of:

x:x, +:0 _____ s:s, #:#
Turning the data into a finite-state transducer with pairings

• Write down the left, center, and right contexts as declarative constraints
• In this case:
  - x:x +:0 0:e s:s #:#
  - Csib:Csib
• Pad out with nulls (0’s) [to obey the same length constraint]
• Write an FST that accepts exactly this string, and rejects everything else (we want the FSTs to work basically as filters)
And acceptance (cook until done)
Tabular format for FST - the state table

Rules:
Epenthesis

<table>
<thead>
<tr>
<th>lexical</th>
<th>surface</th>
<th>Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>c h s</td>
<td>c h s</td>
<td>+ # 0 @</td>
</tr>
<tr>
<td>Csib</td>
<td>Csib</td>
<td>0 # e @</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
</tr>
<tr>
<td>2:</td>
</tr>
<tr>
<td>3:</td>
</tr>
<tr>
<td>4:</td>
</tr>
<tr>
<td>5:</td>
</tr>
<tr>
<td>6.</td>
</tr>
<tr>
<td>7.</td>
</tr>
</tbody>
</table>

For states:
0 = failure state; colon after state number: an accepting state

Note! We will have to change this table a bit…let’s see why..
What about… Spy vs. Spies?
Here there are several spelling change rules…
This example requires two

Epentheses rule

\[
\begin{array}{c}
\text{spy} + 0 \text{s} \\
\text{spi} 0 \text{e} \text{s}
\end{array}
\]

\[
0:e \leftrightarrow \text{Csib:|y:i_ s:s}
\]

y-i rule

\[
\begin{array}{c}
\text{spy} 0 + \text{s} \\
\text{spie} 0 \text{s}
\end{array}
\]

\[
y:i \leftrightarrow _0:e :+:0
\]

The epenthesis automaton must be ‘aware’ of what the y:i automaton does…and vice-versa
## More than 1 Spelling change rule

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consonant Doubling (gemination, G)</td>
<td>1-letter consonant doubled before -ing/ed</td>
<td>beg/begging</td>
</tr>
<tr>
<td>E deletion (elision, EL),</td>
<td>Silent e dropped before -ing, -ed</td>
<td>make/making</td>
</tr>
<tr>
<td>E insertion (epenthesis, EP)</td>
<td>e added after -s, -z, -ch, -sh before -s</td>
<td>fox/foxes</td>
</tr>
<tr>
<td>Y replacement (Y)</td>
<td>-y changes to -ie before -ed</td>
<td>try/tried</td>
</tr>
<tr>
<td>I spelling (I)</td>
<td>I goes to y before vowel</td>
<td>lie/lying</td>
</tr>
</tbody>
</table>
The Lexicon FST

- What’s the format?
- Make transitions for prefix-root-suffixes
- The transducer ‘output’ (the ‘right-hand side’) is **not** used to determine transitions, but it **is** used to produce ‘gloseses’, e.g., spies → (Noun(spy)+PL)
  (please remember this for Lab 3!)
- The Lexicon FST much in general have loops and ‘epsilon’ transitions (that is, transitions without consuming any input), because it must **guess** whether what it sees is, e.g., a ‘noun’ or a ‘verb’ (or possibly some other state), as well as whether we’ve reached the end of a word
- This could be improved by lookahead, and by using a part-of-speech tagger, but it’s **not** implemented in this system
Example: *spies*

Goal: to understand *where* and *why* it succeeds, and *where* and *why* it gets stuck & must backup.
Lexicon code: how the FST is specified

- Tabular format: <initial state> <next states> or
  <initial state>
  <transition symbols> <next state> <output symbols>

Begin: Prefix Root
Prefix: ADJ_PREFIX V_PREFIX
Root: N_ROOT ADJ_ROOT V_ROOT_PREF V_ROOT_NO_PREF

- Note 1: no transition symbol required - so an ‘epsilon’ (empty transition)
- Note 2: the first line describes epsilon transitions from the Begin state to 2 ‘next states,’ Prefix and Root; then these 2 states themselves branch
- Note 3: the trace won’t display all successive transitions in such cases; in this instance, we jump all the way to ADJ_PREFIX, to begin. But what happens next?
Lexicon FST code after ADJ_PREF

ADJ_PREF:
un+  ADJ_ROOT  Not+

- So here, the Lexicon FST says that, starting from the state ADJ_PREF, it must find the symbols un+ in the underlying form (the lexicon) – it does not care about the ‘surface form’ (the spelling change automata will take care of that).
- So what is happening here is that the Lexicon FST is ‘waiting’ for the spelling change automata to check whether un+ can be matched up with what appears on the surface (which is spies).
- If it can be matched up, then the Lexicon FST moves to state ADJ_ROOT and outputs the symbols Not+ (a ‘gloss’); if it can’t, then the automata fails and backs up, to try the next state, V_PREF.
But **V_PREFIX** will fail:

**V_PREFIX:**
- re+  **V_ROOT_PREF REP+**
- un+  **V_ROOT_PREF REV+**

So we will wind up at state:  

**N_ROOT**
Starting from **N_ROOT**, spelling changes take over for a while…

**N_ROOT:**

`'cat` **AfterNoun** **Noun**(cat)

…

`'spy` **AfterNoun** **Noun**(spy)

- Note: the first symbol is ` `, which is matched with a **surface 0** character (What’s this for??? We’ll ask you…)
- Note: it is **important** to see that an underlying character can be different from a surface character
- Beyond that, we start matching up underlying/surface characters, e.g., **s:s**
- To keep straight what character pairs are currently being processed in the trace, they are displayed in angle brackets
- Now all the spelling change automata are being run ‘in parallel’; if **any** of them rejects, the current attempt fails & we backup in the **spelling change** rules & make a new try
Epenthesis machine is in state 1, and we’ve matched up `sp:0sp, what happens next??
Input characters underlying: surface are: \[\text{y}:i \ (\`\text{spy}:0\text{spi})\]

now what does epenthesis machine do?

- This FST was in ‘state 1’, and now it changes to state 3. Why?
- Remember what epenthesis is about?
- Fox-foxes, buzz-buzzes, …
- Epenthesis **inserts** an e on the surface; this is paired with what in the underlying spelling?
- So why has the epenthesis machine changed state?
- Answer: there’s the y-i spelling rule operating as well… which has actual job of checking the y-i business
- So this change to state 3 is to accommodate the possibility that the y-i spelling change has taken place
y-i spelling change checking FST: not OK if before +:0 (an affix boundary)

So, ephensis machine must know this too… and we haven’t seen +:0 yet
Interaction of epenthesis & y-i rule: must modify epenthesis to include possibility of y:, and y-i rule must include possibility of epenthesis

Epenthesis rule

\[
\begin{align*}
\text{s} \text{p} \text{y} + 0 & \xrightarrow{} \text{s} \\
\text{s} \text{p} \text{i} 0 & \xrightarrow{} \text{e} \text{s}
\end{align*}
\]

y-i rule

\[
\begin{align*}
\text{s} \text{p} \text{y} 0 + & \xrightarrow{} \text{s} \\
\text{s} \text{p} \text{i} 0 & \xrightarrow{} \text{e} \text{s}
\end{align*}
\]

\[
0:e \leftrightarrow \text{Csib:| y:i +:0 } \_ \text{s:s} \quad \text{y:i } \leftrightarrow \_ 0:e +:0
\]

The Epenthesis machine must be ‘aware’ of y:i possibility…and vice-versa
So there’s some interaction; fortunately, it is local
So we must **modify** the original epenthesis rule...and it goes to state 3 if y-i is found. What happens if this happens **before** affix +:0?

**epenthesis3:** |

<table>
<thead>
<tr>
<th>FSA</th>
<th>c h s Csib y + # 0 @</th>
</tr>
</thead>
<tbody>
<tr>
<td>c h s Csib</td>
<td>i 0 # e @</td>
</tr>
<tr>
<td>1: 2 1 4 3</td>
<td>3 1 1 0 1</td>
</tr>
<tr>
<td>2: 2 3 3 3</td>
<td>3 1 1 0 1</td>
</tr>
<tr>
<td>3: 2 1 3 3</td>
<td>3 5 1 0 1</td>
</tr>
<tr>
<td>4: 2 3 3 3</td>
<td>3 5 1 0 1</td>
</tr>
<tr>
<td>5: 2 1 2 2</td>
<td>2 1 1 6 1</td>
</tr>
<tr>
<td>6: 0 0 7 0</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>7: 0 0 0 0</td>
<td>0 1 1 0 0</td>
</tr>
</tbody>
</table>
We **reject** this possibility - state 0

Why? (Hint: look at what **lexicon** state we are in)
So in effect, the machine has produced the pair:
`spy0# : 0spie#

N_ROOT:
`cat  AfterNoun Noun(cat)
...
`spy  AfterNoun Noun(spy)
AfterNoun: NounSuffix
NounSuffix: End GENITIVE PLURAL N_TO_ADJ
...
GENITIVE:
+s  AfterGenitive +GEN

PLURAL:
+s  AfterPlural  +PL
'#  End None

So we back up to last failure point...in the Lexicon & all the other FSTs...which is...
So machine backs up, and retries; first the Genitive state in the Lexicon, but this fails (why?); and then, finally, the Plural state (which succeeds)

Are we done?
Why can’t we get rid of the ugly nondeterminism?

- Problem: Finite-state transducers cannot always be made deterministic
- Still, this method is better than even more powerful devices, like context-free grammars and beyond... as we’ll see
- Here’s an example of the inextricable nondeterminism...
Finite-state automata can be **nondeterministic**

This machine accepts the language $x^+\{a|b\}$

Can you think of a natural language example? Suggestion: suppose $x$ is some verb…

Fact: we can always convert such a machine to a **deterministic** one. How? (Remember…?)
The conversion **can** blow up...

\[ \{a, b\}^* a \{a, b\}^n. \]
Converting nondeterministic fsa to deterministic one via subset construction

What does being in state \{1,2\} mean???

Is it some sort of existential dilemma?
Blow-up

8 states…
Now add 1 more state, resulting machine:

6.863J/9.611J SP11
Blow-up is exponential
But this conversion is **not** even always possible with a finite **transducer**

Why should we care?
OK, some final details, then onto Lab 3 & then an assessment of this approach…
Files you need

In file (eg, english.yaml):

1. Specify the boundary marker (usually #)
2. Specify where to find the lexicon automaton (e.g., english2.lex)
3. Specify the lexical, surface alphabet as a set of ‘defaults’
4. Specify any special subsets for abbreviatory purposes
5. Specify the spelling change automata
The english.yaml file

boundary: '#'
lexicon: english2.lex
defaults: "a:a b:b c:c d:d e:e f:f g:g h:h i:i j:j k:k l:l m:m n:n o:o p:p q:q r:r s:s t:t u:u v:v w:w x:x y:y z:z +:+0 `:`0 #:# ':': '-' --: --:0"
subsets:
  "@": "a b c d e f g h i j k l m n o p q r s t u v w x y z ' ` + # 0"
  "C": b c d f g h j k l m n p q r s t v w x y z
  "Csib": s x z
  "V": a e i o u
  "Vbk": a o u
(automata follow)
Laboratory 3: build a 2-level system for a (small!) subset of Spanish

- You will build the Lexicon and automata
- We provide you with an example automaton and the lexical, surface character alphabet, etc. to get started
Lab 3: Spanish – Your questions

1. What is your name?

2. What is your quest?

3. What is your favorite color?
Laboratory 3: Spanish

- What phenomena you’re covering
- How to build spelling-change FSTs - details
- How to build Lexicon FST - details
The phenomena

• You are given the orthography, including some special characters to stand for the accented ones á,é,ó,ü,ñ; and some underlying characters you may find essential, such as J, C, Z.

• Wise to proceed by *first* building the automata (yaml) file; *then* the lexicon(s) - because you can test the rules without any lexicon by *generation* of a surface form.

• The automata can be built (roughly) by considering each phenomenon separately.

• 4 kinds of phenomena & 2 morpheme patterns.
# this is a comment at the top of my spanish.yaml file
boundary: '#'
lexicon: spanish.lex
defaults: "a e i o u a' e' i' o' u' b c d f g h j k l m n~ p q r s t
      v w x y z +:0 #"
subsets:
  "Cons": "b c d f g h j k l m n~ p q r s t v w x y z"
  "V": " a a' e e' i i' o o' u u'"
  "FRONT": "e i e' i'"
  "BACK": "u o a u' o' a'"
  "LOW": "e o a a' e' o'"
  "HIGH": "i i' u u'"
  "@": "a e i o u a' e' i' o' u' b c d f g h j k l m n~ p q r s t v w
      x y z + ` # 0"
rules:
The phenomena

Spelling changes:
(freebie: u-insertion)
1. g-j mutation
2. z-c-z mutation
3. z-c mutation
4. Pluralization
(You can use Google translate to ‘hear’ some of the changes, but be careful…sometimes they will sound not so different)

Lexicon automaton:
Noun endings
Verb conjugation - 1 form
u-insertion

• Let’s see how to turn this into a spelling-change automaton
• The data: must insert u after g if followed by front vowel (e, i, é, í)

Accept:

  pague (1st person subjunctive. ‘pay’; ‘yo pague’
  pa0ue

More generally, Accept:

  XguF
  Xg0F

But Reject (no u!) if followed by anything else, ie, ‘yo pagar’
In words…

• Loop until we find a $g: g$

• Pair with $0: u$ and see if a Front Vowel (Front) vowel follows; if so, accept; otherwise, reject
In a picture:

- Start Loop
- until
- @: @
- Found
- @: @
- Inserted
- @: @
- Followed by
- @: @
- g: g
- @: 0
- g: g
- Else
- Reject
- @: @

Now insert ‘rejects’:
- If @: FRONT after g: g
- If 0: u at Start

Now insert ‘idling’:
- g: g Stay put
- @: 0 Stay put
pagar: yo pago (1st person present; yo pague (1st person subjunctive)

u-insertion:

start:
  'g': found_g
  '0:u': reject
  '@': start

found_g:
  '0:u': inserted
  '@:0': found_g
  'g': found_g
  '@:FRONT': reject
  '@': start

inserted:
  '@:FRONT': start
  '@': reject
Phenomenon 2: z-c mutation

• **z-c mutation**
  
  $z \rightarrow c$ before front vowels, $z$ otherwise
  
  *cruzar* (to cross); *cruzo, cruzas, cruza, cruzamos, cruzan, cruce*

• If $s$ causes a front vowel (e.g., $e$) to surface, then the rule still applies:
  
  *lápiz, lápices* (pencil, pencils) [ *la’piz, la’pices*]
Example: look at phenomenon, then see first how to describe

- What is the left and right context of the change?
- Write it as a declarative constraint
- Remember that you can use both the surface and the lexical characters to admit or to rule out a possibility
- Thinking in terms of constraints (what is ruled out by the rule) is the most difficult ‘mindset’ to attain...
Build automaton for lexical, surface pairs

• But what are the lexical pairs?
• Ah, your job!
• In general, the underlying form is not generally the infinitive, e.g. an ordinary dictionary will list ‘cruzar’ as the ‘infinitive form’ but this is not the same as the root!!!
  
  cruzar, cruzamos → legit pair?
  
  cruzar
cruzamos

Look at the other pairs – what do you think the root is?
Writing rules

• cruzar/cruzamos cruzar/cruce?
• We can try a (tentative) lexical/surface pair, and from that extract the right spelling change
• Do it step by step: use the alignment to write down the ‘straight-line’ acceptance path:
  cruz
  cruce
Pad out length by using 0’s (nulls) (why is this important)—Remember the equal length constraint?
  cruz0 cruz0
  cruce cruz0
Outline context – hmm, perhaps we do need the root?
Writing rules

From context to rule:

\[
cruz0, \text{cruce} \ c:c, \ r:r, \ u:u, \ z:c, \ 0:e - \text{accept}
\]

But… is this the correct root?
Design of morpheme automaton

• One big automaton, that handles two phenomena:
  • plurals and
  • verb endings
Automaton design for lexicon

Q: what do we need to add to noun sequence?
A: take a look at English as a guide
The morpheme tree: Adding plurals - **ciudades**

Output:

```
Noun(city) + Number: Plural
```

Final output: `[Noun(city)+Number: Plural]`
The lexicon – take 2

You will deal with two types of ‘endings’
Noun endings: plural suffix +s
Verb endings: verb stem + tense markers
  Simplest: infinitive marker +ar, +er, +ir
  See table in lab file for details: 5 x 3 table for Present tense; ditto for Subjunctive tense (“I might....”)
Note: this verb table is not meant to be complete or perfectly accurate; it has been simplified to make Spanish easier! (and so your job easier)
Lexicon specification details

- Lowercase: *alternation* states - epsilon transitions
- Uppercase: *lexical* states – actual spell-out of prefixes, roots, suffixes

Begin: Prefix Root
Prefix: ADJ_PREFIX V_PREFIX
Root: N_ROOT ADJ_ROOT V_ROOT_PREF V_ROOT_NO_PREF N_ROOT:

V_PREFIX:
re+ V_ROOT_PREF REP+
un+ V_ROOT_PREF REV+
...

N_ROOT:
`cat` AfterNoun Noun(cat)
`dog` AfterNoun Noun(dog)
...

End: #