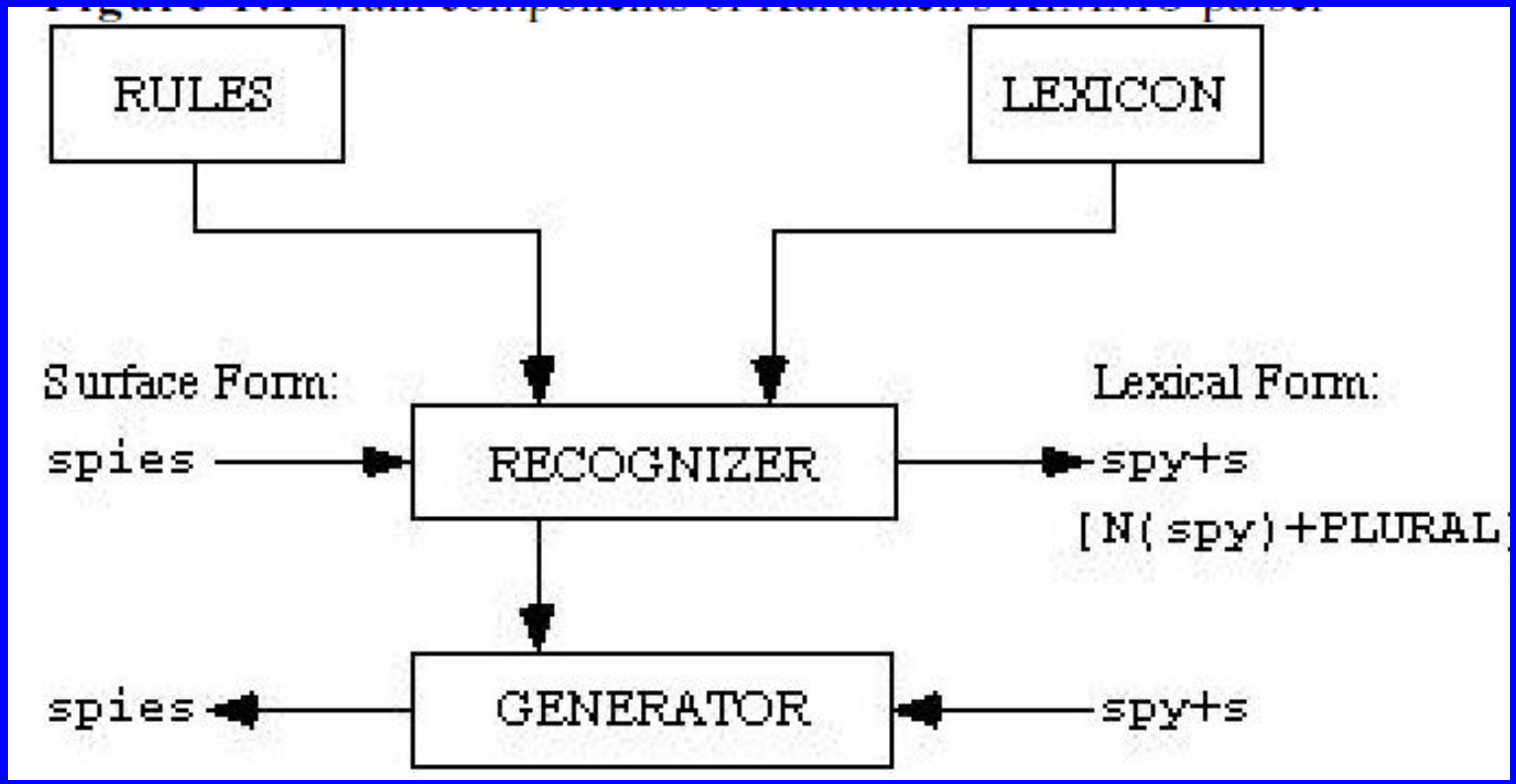
6.863J Natural Language Processing
Lecture 3: From morphology to Spanish

Instructor: Robert C. Berwick

The Menu Bar

- Administrivia
  - Lab1b, Spanish morphology: now posted; (we'll go over this at the beginning of class)
  - *Note* due date change: February 24
  - Sample lexicon, Kimmo docs posted
- *Agenda:*
- Summary of two-level machinery
- Some of the computational details of finite-state transducers, and the problems with 'conventional' linguistic rules
- Review lab1b goals and how-to

Big picture of what Kimmo does



Summary of two-level morphology

- Two-level Morphology: *two* FSA devices, one for words – a "word tree" fsa; one for spelling change rules
  - phonology + "morphotactics" (= morphology)
- Both components use finite-state devices:
  - phonology: "two-level rules", converted to FST's
    - e:0 ⇔ _ +:0 e:e r:r
  - morphology: linked lexicons, or FSA's
    - root-dic: book/"book" ⇒ end-noun-reg-dic
    - end-noun-reg-dic: +s/"NNS"
- Integration of the two possible (and simple)

Recognizing (or generating) a word means following the joint *path through both finite-state devices*

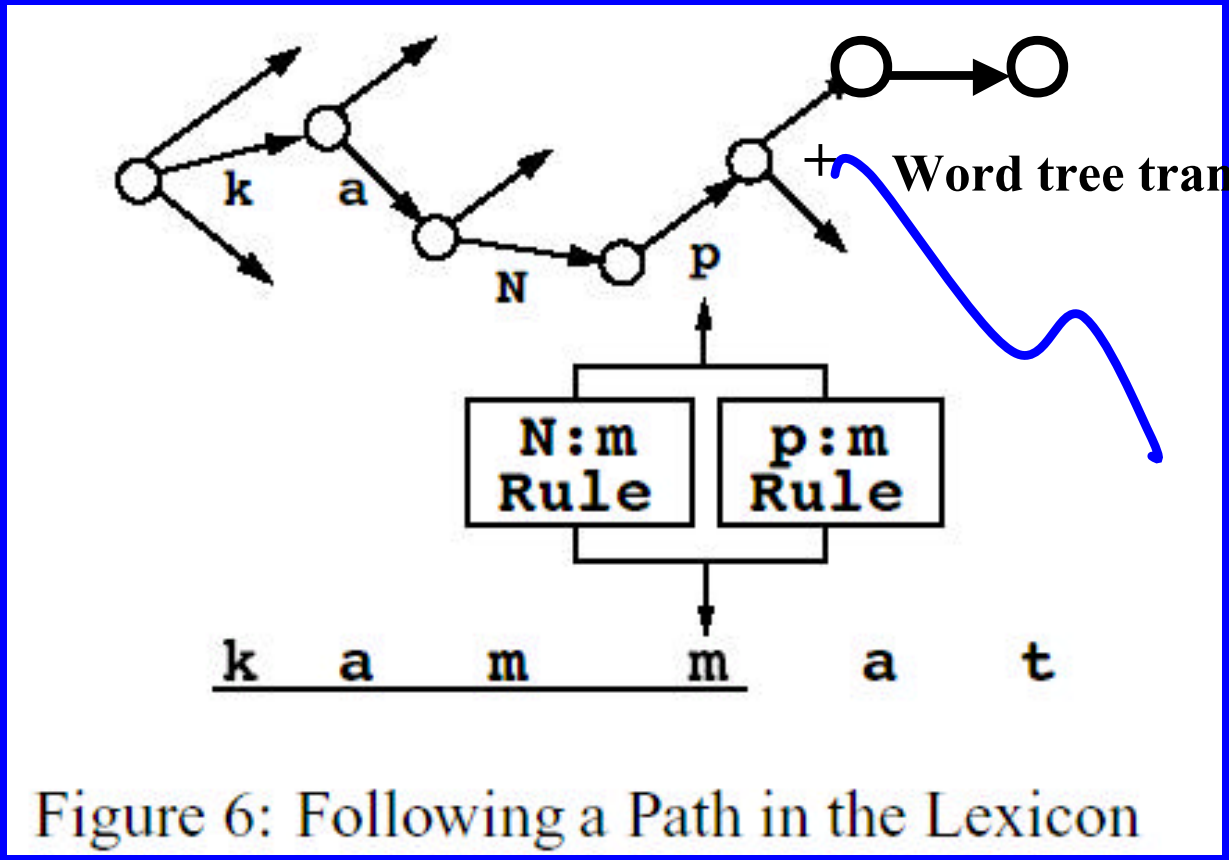
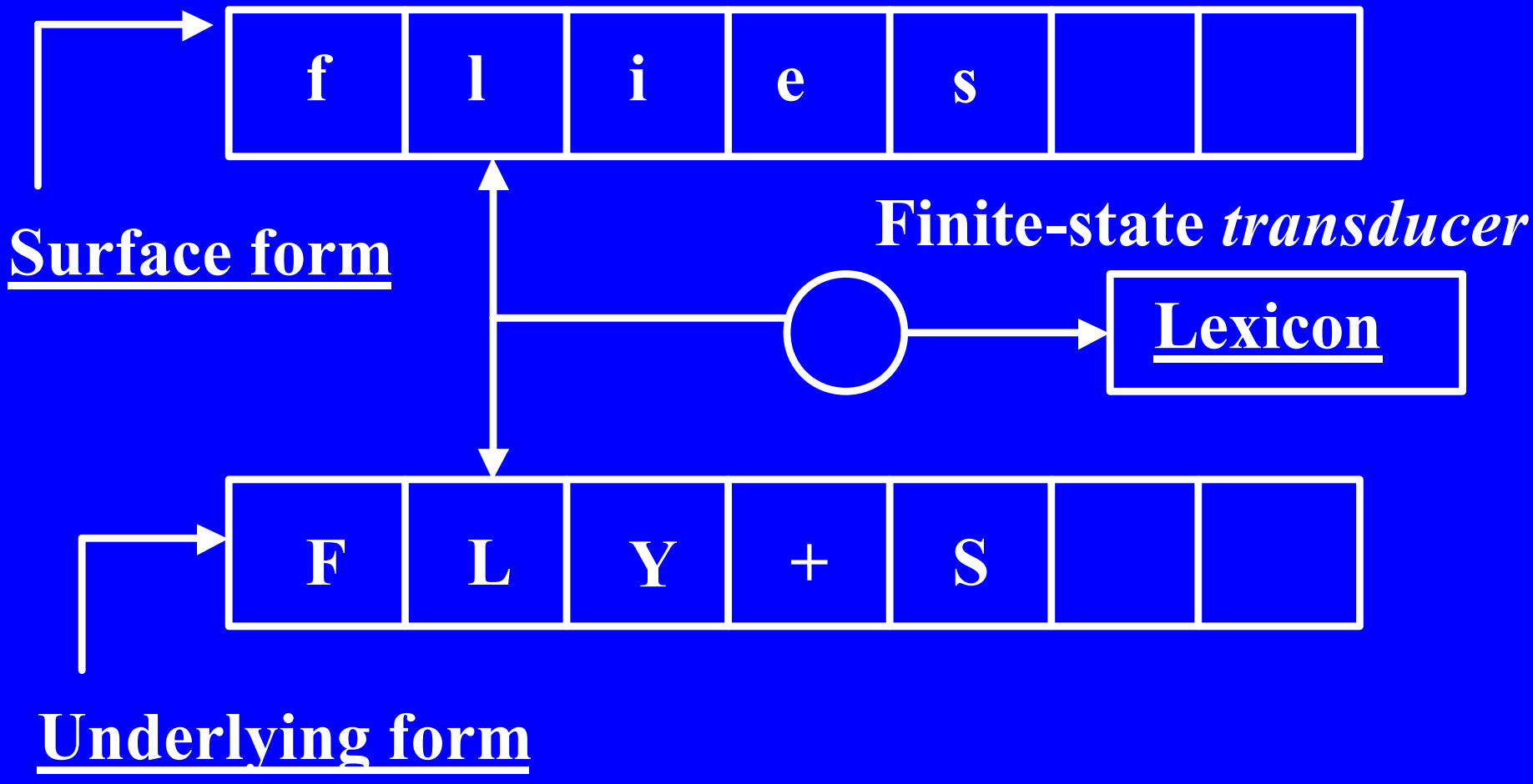
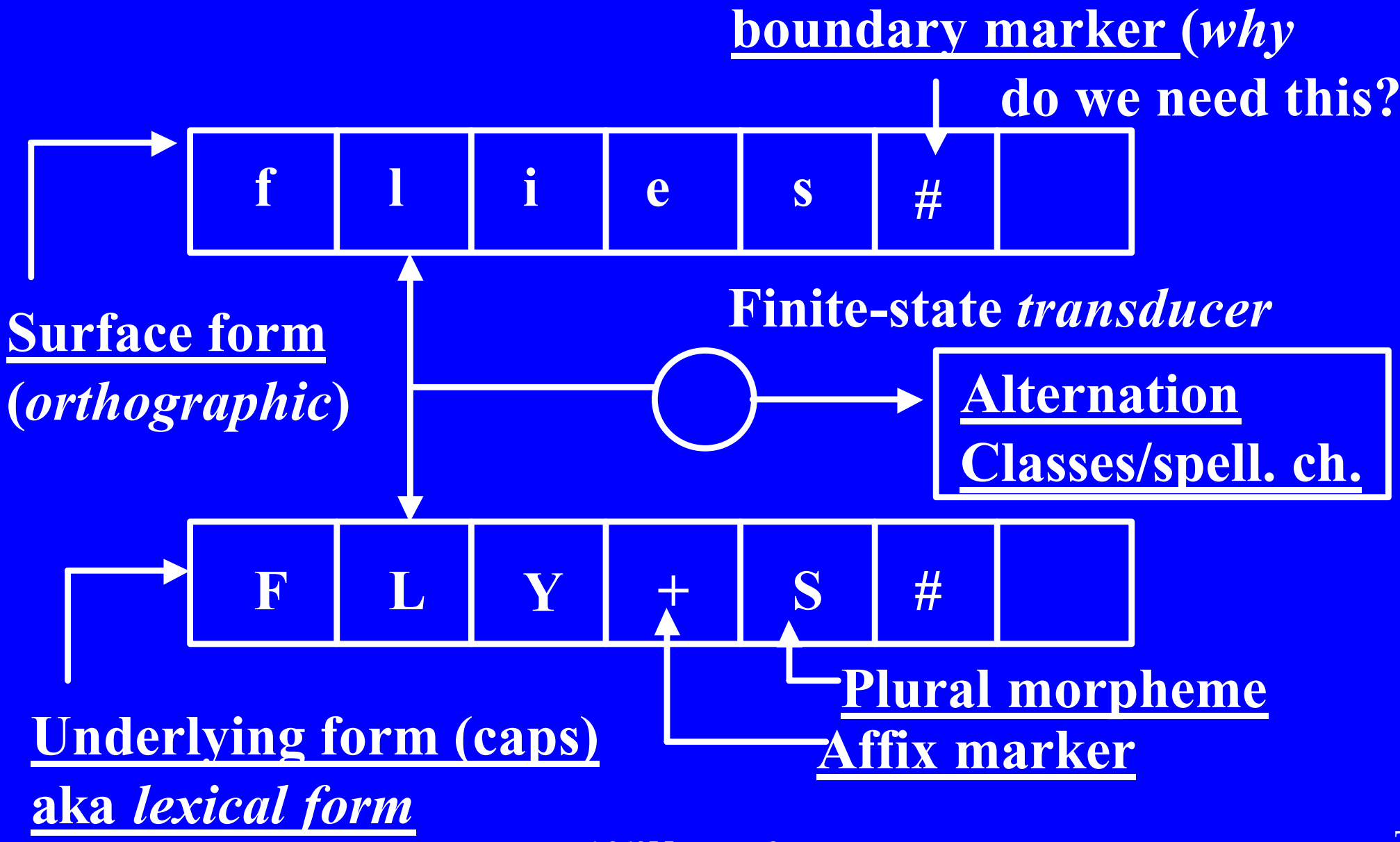


Figure 6: Following a Path in the Lexicon

2 tapes


f l i e s
Surface form
Finite-state *transducer*
Lexicon
F L Y + S
Underlying form

Kimmo terminology



The two components

1. One to check that the *right* affixes follow the *right* stems, e.g., *big-er* but not *big-ly*

   This includes 'spelling out' the right stems, e.g., '*p n e u m ...*' but not '*p t k …*'

   These are called *alternation classes* in the Kimmo system (They are in the file **`english.lex`**)

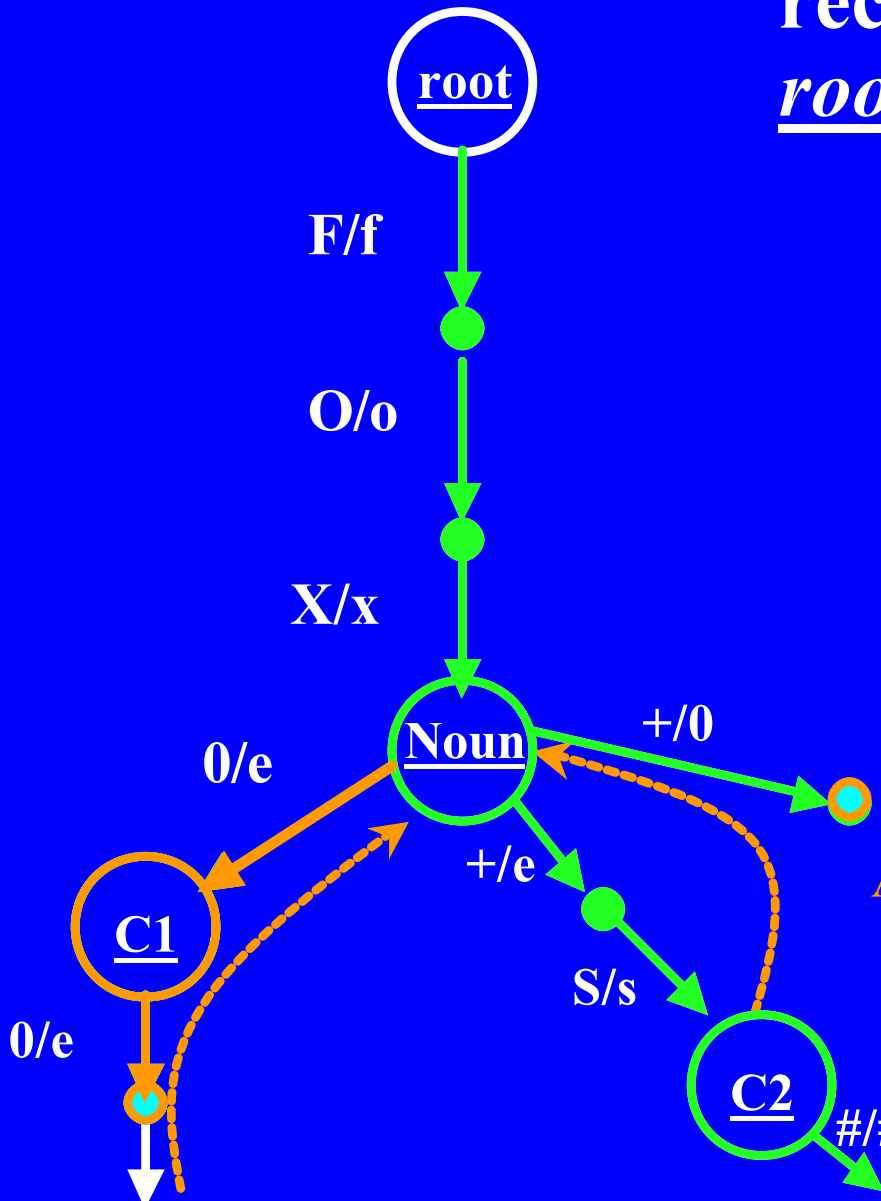
2. One to check that the *right* surface 'letters' correspond to the *right* underlying form letters (or vice versa), taking into account the phonological/morphological changes (the spelling change rules). These are called *rules*. (They are in the file **`english.rul`**)

The Word tree is pretty simple as an fsa

- Just a set of 'states' representing root classes and beginning (or prefix) and ending (suffix) classes
- Transitions between the classes based on character sequences
- Example: *er* glued onto *big* transition to: Comparative

We trace through the finite-state devices in tandem

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



= FST1 (word classes)

= FST2 (spell changes)

Automaton blocks

END!

Let's see an example - English

races'

```
Recognizing surface form "races'".
0 (r.r) --> (1 1  1 2 1  1)
              EP G Y EL I

1   (a.a) --> (1 1  4 1 2  1)
                EP G Y EL I

2     (c.c) --> (1 2 16 2 11 1)

3       (e.0) --> (1 1 16 1 12 1)
                    EP G Y EL I

4         Entry |race| ends --> new lexicon N, config (1 1 16 1 12 1)
                                                      EP G Y EL I
```

- *note* that each of the 5 automata are running "in parallel" (what is the first one?)

Backup search

Problem: ***e*** is paired with 0 (null)…!
(which is wrong - it's guessing that the form is "racing" - has stuck in an *empty* (zero) *character* after *c* but before *e*) - *elision* automaton has 2 choices
This is *nondeterminism* in action (or inaction)!

```
5          Entry /0 ends --> new lexicon C1, config (1 1 16 1 12 1)
                                                      EP G Y EL I
6            Entry /0 is word-final --> path rejected (leftover input).
5          (+.0) --> (1 1 16 1 13 1)
                       EP G Y EL I
6            Nothing to do.
5          (+.e) --> automaton Epenthesis blocks from state 1.
4        Entry |race| ends --> new lexicon P3, config (1 1 16 1 12 1)
                                                        EP G Y EL I
```

22 steps later

```
3        (e.e) --> (1 1 16 1 14 1)
                        EP G Y EL I
4         Entry |race| ends --> new lexicon N, (1 1 16 1 14 1)
                                                  E  G Y EL I
5          Entry /0 ends --> new lexicon C1, config (1 1 16 1 14 1)
6           Entry /0 is word-final -->rejected (leftover input)
5            (+.0) --> (1 1 16 1 15 1)
6             (s.s) --> (1 4 16 2 1 1)
7              Entry +/s ends--> new lexicon C2,  (1 4 16 2 1 1)
8               Entry /0 is word-final -->rejected(leftover input)
8                ('.') --> (1 1 16 1 1 1)
9                 End --> lexical form ("race+s'" (N PL GEN))
```

What is the source of this nondeterminism?

1. Empty elements inserted/deleted (optionally)- we don't know until we see more
   - *rac…* could be *racing:* so it pairs surface 0 with underlying *e.* (oops, will be a poor guess…)
2. Alternative sublexicons searched (depends on order they are placed in the .lex file)

So there can be a lot of searching

R/r A/a C/c 0/e…

```
5       Entry /0 ends --> new lexicon C1, config (1 1 16 1 12 1)
                                                   EP G Y EL I
6         Entry /0 is word-final --> path rejected (leftover input).
5       (+.0) --> (1 1 16 1 13 1)
                     EP G Y EL I
6         Nothing to do.
5       (+.e) --> automaton Epenthesis blocks from state 1.
4     Entry |race| ends --> new lexicon P3, config (1 1 16 1 12 1)
                                                     EP G Y EL I
```

C1= class that represents end of word
C2= class that allows plural endings
these fail! Why? (consider form)
Why does epenthesis automaton block? (consider 'foxes' vs. 'rac')

Also nondeterminism in generation of word forms

- Because automaton can posit null characters (zeroes) on the *surface,* as in 'foxes'
- Generate from 'FOX+S':
  1. f
  2. fo
  3. fox
  4. foxt (gem. fails)
  5. foxs (gem. Fails)
  6. foxr "
  7. foxp
  8. foxn
  9. …
  15. fox0
  16. fox0s epenthesis
  17. foxe
  18. foxes

The problem with classical sequential rewrite rules

- Get different results when you go in different directions – let's see an example
- Upshot: deterministic in one direction (unambiguous), but not in another

Finite-state transducers + replacing 'classical' rules – trickier – an Intractable problem?

- Example showing the important of order
- *iNpractical vs. iNtractable*
- How do these surface?

- The underlying N surfaces differently!

- To ensure that these and only these results are obtained, the following rules must be treated as obligatory and taken in the order given

Rules needed

Rule 1

N→m / ___ [+labial]

Rule 2

N → n

First rule must feed the second – otherwise, we would be *inpracticable* as well (must kill off the N)

This gives a unique result in this 'forward' order, but *not* in the reverse order…

Reverse gear often ambiguous

- *intractable* → apply Rule 2 inverted → results?
- *iNtractable* and *intractable* produced
- Also, sometimes *no* results when inverting
  - *iNput* cannot be generated by Rule 1, because the *N* always gets converted to *m*
  - So, no output at all when Rule 1 is inverted on *iNput*

BUT if you invert Rule 2 on *iNput* you get 2 results, as we have seen (input and iNput)

Inverting Rule 1 *removes* the ambiguity created by inverting Rule 2

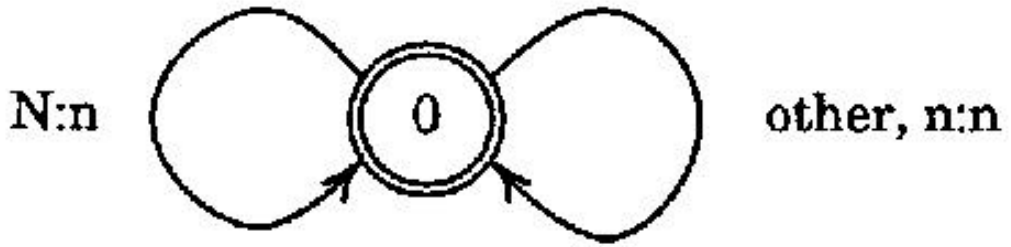
And in general…

- If recognition is carried out by inverting grammar rules in turn, then later rules in the sequence act as filters on ambiguities produced by later ones
- This effect is multiplicative in a cascade, since the info to cut off paths does not become available until quite far down the road, in some cases

Finite state transducer

- Imagine *two* tapes (lexical, surface)
- Transition arcs between states in form x:y
- A transition can be taken if the two symbols separated by the colon in its label are found at the current position on the corresponding tapes, and the current position advances across those tape symbols.
- A pair of tapes is accepted if a sequence of transitions can be taken starting at the start-state (conventionally labeled 0) and at the beginning of the tapes and leading to a final state (indicated by double circles) at the end of both tapes.

Let's consider the obligatory transducer versions of these rules – Rule 2 first (n-machine)



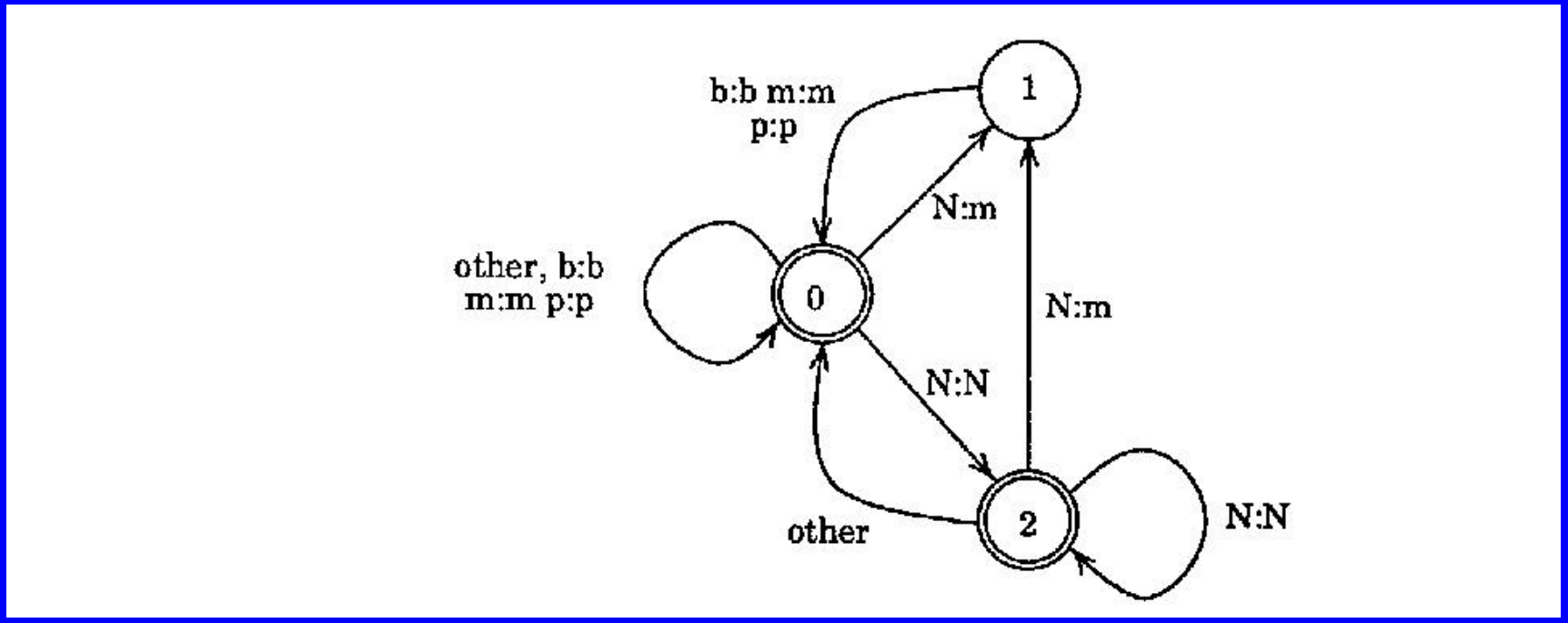
Assume 2 tapes: top, bottom symbol

This machine *accepts* just in case a pair of strings (tapes) stands in a certain relation w/ e.o – viz., *N* on first tape is replaced with *n* on the second, and *no N:N* possible

I.e., this is Rule 2, obligatory

Rule 1 – more complicated

N→m / ___ [+labial]



Rule 1 (*m*-machine) description

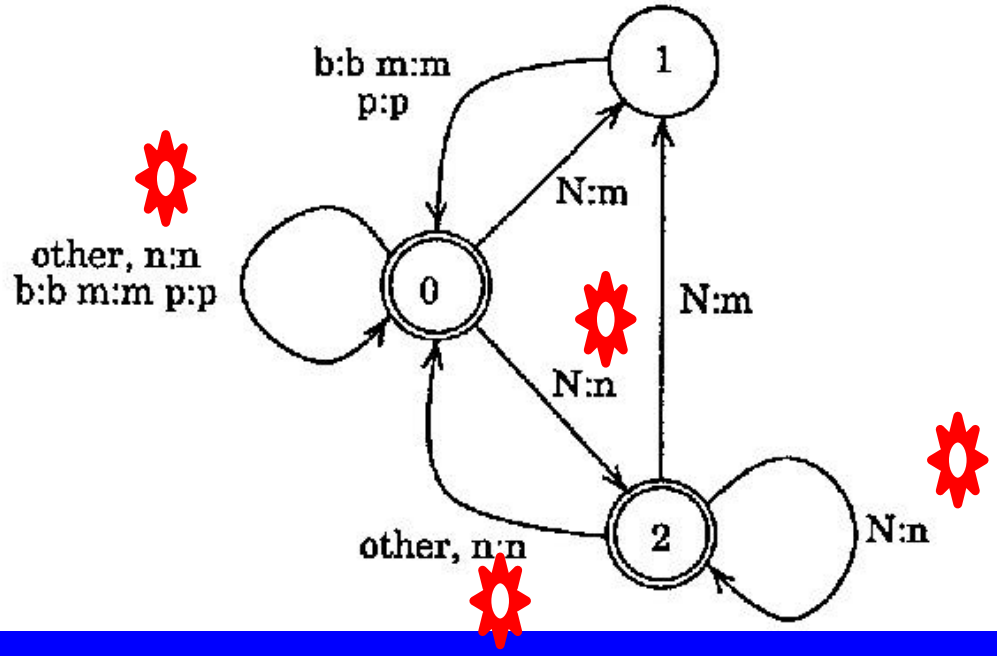
- This machine blocks in state 1 if it sees the pair *N : m* not followed by one of the labials p, b, m

- It blocks in state 2 if it encounters the pair *N : N* followed by a labial on both tapes, thus providing for the situation in which the rule is not applied even though its conditions are satisfied
- If it does not block and both tapes are eventually exhausted, it accepts them just in case it is then in one of the *final* states, 0 or 2, shown as double circles
- It rejects the tapes if it ends up in the nonfinal state 1, indicating that the second tape is not a valid translation of the first one

Advantages of transducer model

- Goes both ways – generate or recognize (depending on which tape contains the input)
- A pair of transducers connected through a common tape models the composition of the relations that those transducers represent
- That is, the relations accepted by finite-state transducers are closed under serial composition

Picture please… model the cascade = composition *of the 2 transducers*



This machine is constructed so that it encodes all the possible ways in which the m-machine and n-machine could interact through a common tape. The only interesting interactions involve N, and these are summarized in the following table:

Composition of the 2 machines

Rule 1 Rule 2

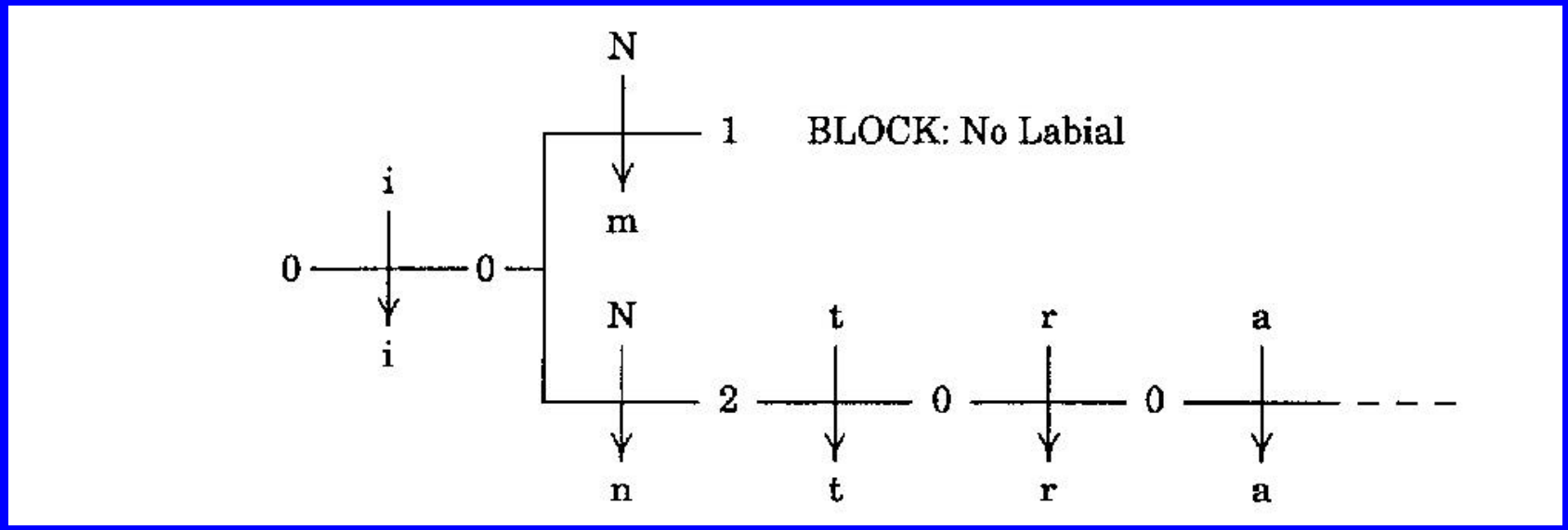
input	*m*-machine	output		
		input	*n*-machine	output
N	*labial follows*	m		m
N	*nonlabial follows*	N		n

An N in the input to the m-machine is converted to m before a labial and this m remains unchanged by the n-machine.

The only instances of N that reach the n-machine must
therefore be followed by nonlabials and these must be converted to n.

Accordingly, after converting N to m, the composed machine is in state 1, which it can leave only by a transition over labials. After converting N to n, it enters state 2, from which there is no labial transition. Otherwise, state 2 is equivalent to the initial state.

Generation from iNtractable



Starting in state 0, the first transition over the "other" arc produces i on the output tape and returns to state 0.

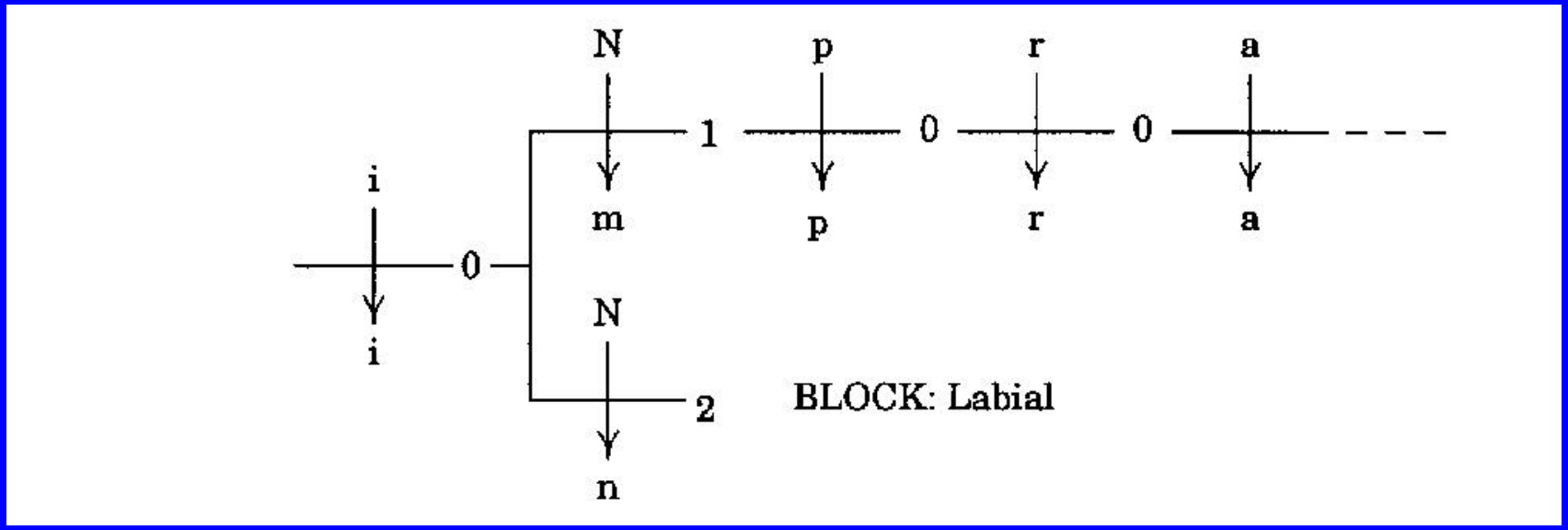
Two different transitions are then possible for the N on the input tape.

These carry the machine into states 1 and 2 and output the symbols m and n respectively.

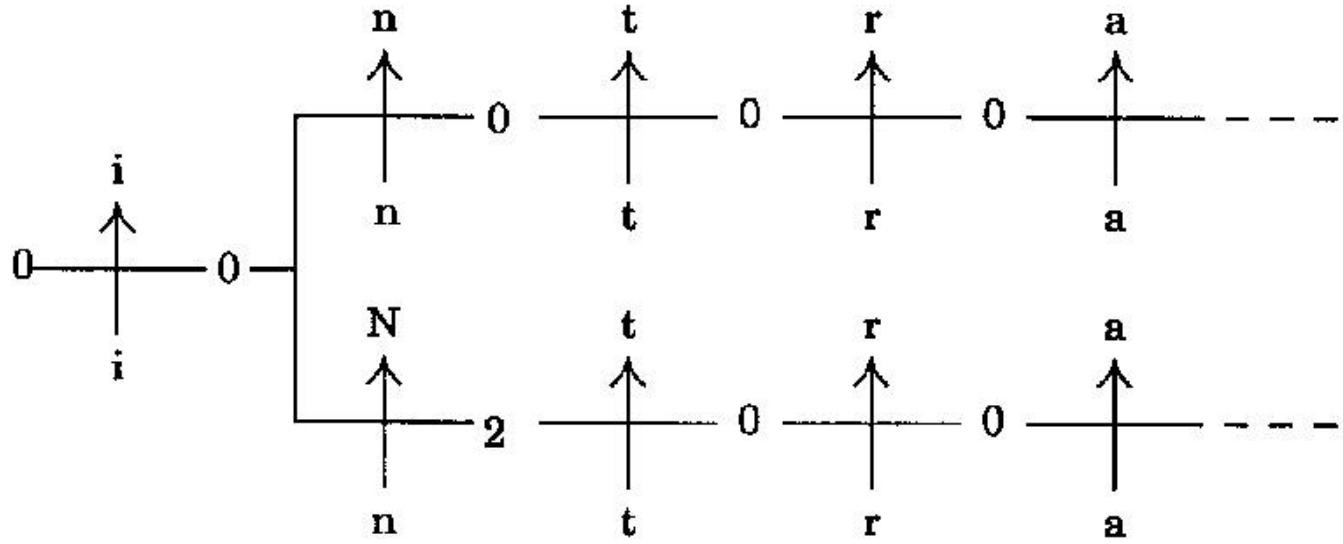
The next symbol on the input tape is t. Since this is not a labial, no transition is possible from state 1, and that branch of the process therefore blocks.

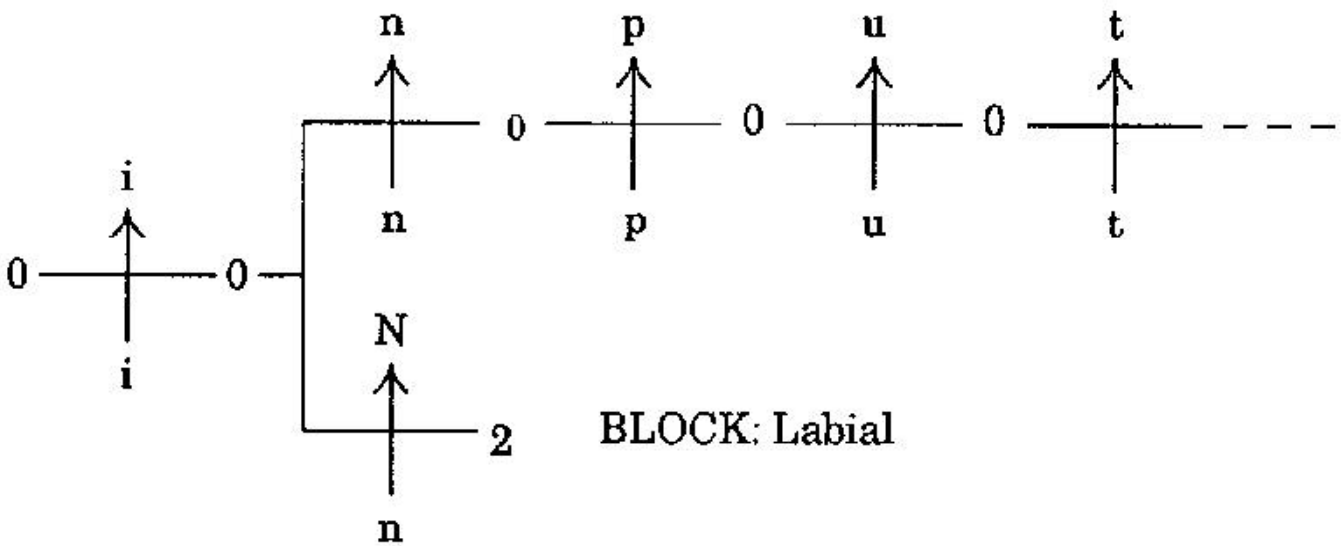
On the other branch, the t matches the "other" transition back to state 0 and the machine stays in state 0 for the remainder of the string. Since state 0 is final, this is a valid derivation

Generation of impracticable



Same machine, as recognizer





Conditions on re-write rules for these to be re-expressed as an fst

- $\varphi \rightarrow \chi / \lambda \_\_ \rho$
- The part of the string that is actually rewritten by a rule is excluded from further rewriting by that same rule
- The following optional rule shows that this restriction is necessary to guarantee regularity:

  $\varepsilon \rightarrow ab / a \_\_\_\_ b$

If this rule applies repeatedly, what language does it produce? This is *not* a finite-state language!

Context conditions

- Rule cannot apply to *itself*
- *But* material produced in one application of a rule can serve as the *context* (triggering condition) for subsequent application of that rule
- Example: vowel harmony, as in Turkish:

YorgIn+sIniz → *Yorgunsunuz*

Context refers to that part of the string that the current application of the rule does *not* change, but which may have been changed in a previous application – so allows for interaction between successive rule applications

- Further constraint: no real ε transitions

Actually leaves a puzzle open

Consider: *unenforceable*

This is usually analyzed as 'bracketed' & analyzed from the 'inside out':

[*un* [*en* [*force*] *able*]]

Hmm… is this finite-state?

Suppose we need to add this

Laboratory 1b

- Goals:
  - How to use Kimmo to analyze another language (Spanish), as example "front end"
  - Build automata for some simple Spanish morphological/phonemic rules (that interact)
  - Build lexicon
  - Learn what is *hard* and what is *easy* about this
  - Recognize *all* and *only* the words in spanish.rec; Generate all the surface forms
- Resources:
  - Lab1b pdf file link from web page
  - File of all the surface words to parse/reject (covering the phenomena) spanish.rec, also linked from web page
  - PC-Kimmo & documentation
  - Program to `compile' rules into automata: fst

What you must turn in (via URL)

1. A description of *how your system operates*
2. URL ptrs to your **`.lex`** and .**`rul`** files

   span.lex

   span.rul
3. A log of a recognition run on the file **`spanish.rec`** which is linked on the web page & also at toplevel on course locker
4. Discussion of what you built/why
5. You must answer 3 questions:

The questions

- What is your name?
- What is your quest?
- What…

The phenomena under study

- 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 (rul) 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
- 3 kinds of phenomena

The phenomena

1. *g-j mutation*
2. *z-c mutation*
3. *pluralization*
4. *Noun endings*
5. *Verb conjugation - 1 form*

What output should look like - recognition

PC-KIMMO>recognize spanish.rec

coger

coger [V(catch, seize, grab)Infinitival]

cojo

coger+o [V(catch, seize, grab)1p, sg, indicative]

coges

coger+es [V(catch, seize, grab)2p, sg, indicative]

coge

coger+e [V(catch, seize, grab)3p, sg, indicative]

Phenomenon 1: g-j mutation

- *g-j* mutation

  g→*j* before a back vowel

  coger (catch, infinitive); *cojo, coges, coge, cogemos, cogen, coja*
- But some verbs not subject to this (exceptions!)

  llegar (arrive); *llego, llegan, pagar* (pay); *pago, pagan*
- *Don't acccept* **llejo, *lleja, *cogo, *coga* (the words don't come marked with * on their sleeves, of course!)
- Hint: can use the lexical (underlying) character J to solve (but there are other ways to do it)

How to build Kimmo systems

How to build lexicons using alternation classes and the actual lexical entries

How to build automata for spelling changes

Format for .lex file - 2 parts

1. Alternation classes – name all the states, some transitions

(1 or more blank lines)

2. Lexicon entries: transitions between the states

(Recall that we consider only the underyling form combinations here – stems + affixes, *not* spelling change rules on the surface)

Use alternation classes to choose between different roots + affixes (to 'remember' whether a N or V of a certain type)

Example: lexicon design

Phenomena: Nouns and Verbs take different endings

Answer:

Different *alternation* classes for Nouns and Verbs

Example surface (s) underlying (u) pairs tell us what to do

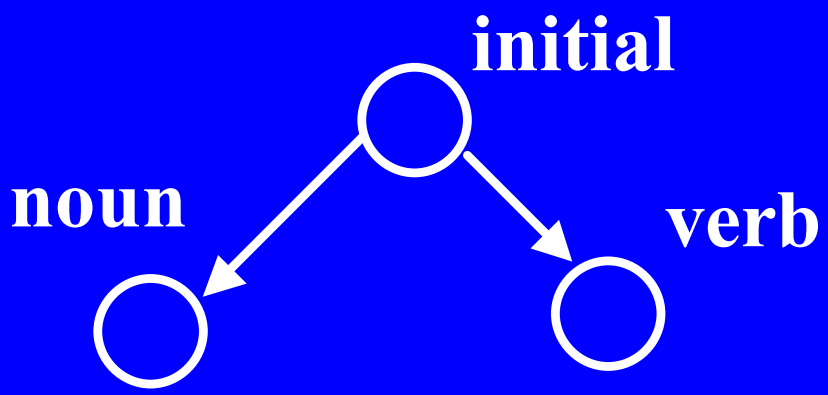
- *ciudad*

  ciudad [N(city)]

- *ciudades*

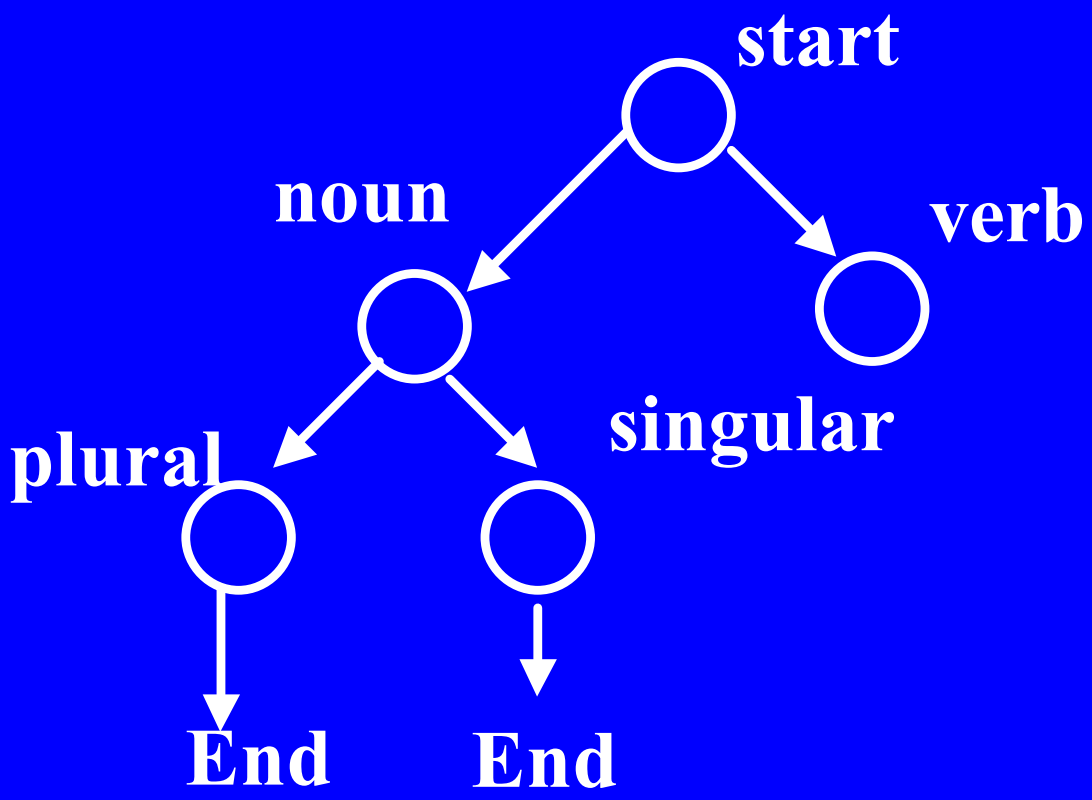
  ciudad+s [N(city)pl]

Automaton design for lexicon



Q: what do we need to add to noun alternation?

Adding plural



Lexicon Design

- 2 parts:

1. Alternation classes (specify state names & most next states)
2. Lexicons (specify transition arcs, a few next states)

How does this translate to lexicon design?

- *Alternations* specify the possible states in the word tree fsa, and most of the transitions (though not the transition symbols)

ALTERNATION Start N V

ALTERNATION N SUFFIX_PLURAL

- So this says: in the *Start* class can be *followed* by either N or V;
- We want the *N* class to be followed by Suffix_Plural (actually, of course, we could put this in V as well, but this would be wrong…)

How to build the lexicon

; To load this file, enter the command LOAD LEXICON ENGLISH.

```
ALTERNATION Begin          N_ROOT ADJ_PREFIX V_PREFIX
ALTERNATION N_Root1         N_SUFFIX NUMBER
ALTERNATION N_Root2         GENITIVE
ALTERNATION N_Suffix        ADJ_SUFFIX3
ALTERNATION Number          GENITIVE
ALTERNATION Genitive        End

ALTERNATION Adj_Prefix1      ADJ_ROOT1
ALTERNATION Adj_Prefix2      ADJ_ROOT1 ADJ_ROOT2
ALTERNATION Adj_Root1        ADJ_SUFFIX1 ADJ_SUFFIX2 ADJ_SUFFIX3
ALTERNATION Adj_Root2        ADJ_SUFFIX2 ADJ_SUFFIX3
ALTERNATION Adj_Suffix1      End
ALTERNATION Adj_Suffix2      ADJ_SUFFIX3
```

End of 'Alternation classes'

```
ALTERNATION V_Pref_Non        V_ROOT_NO_PREF V_ROOT_REVERSE
                                V_ROOT_REPEAT V_ROOT_NEG
ALTERNATION V_Pref_Reverse    V_ROOT_REVERSE
ALTERNATION V_Pref_Repeat     V_ROOT_REPEAT
ALTERNATION V_Pref_Neg        V_ROOT_NEG

ALTERNATION V_Root1           End
ALTERNATION V_Root2           V_SUFFIX1
ALTERNATION V_Root3           V_SUFFIX1 V_SUFFIX3
ALTERNATION V_Root4           V_SUFFIX1 V_SUFFIX2 V_SUFFIX3

ALTERNATION V_Suffix1         End
ALTERNATION V_Suffix2         NUMBER
```

Lexical entries

```
LEXICON      INITIAL
 0                Begin                "[ "
INCLUDE   n_root.lex      ; file containing noun roots
LEXICON      NUMBER
 +s               Number               "+PL"
 0                Number               ".SG"
LEXICON GENITIVE

 +'s              Genitive              "+GEN"
  0               Genitive                   "“
LEXICON          End
 0                #                   " ]"
END
```

Need lexicons for beginning & affix as well

How do we build spelling change automata?

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

Phenomenon 2: z-c mutation

- *z-c mutation*

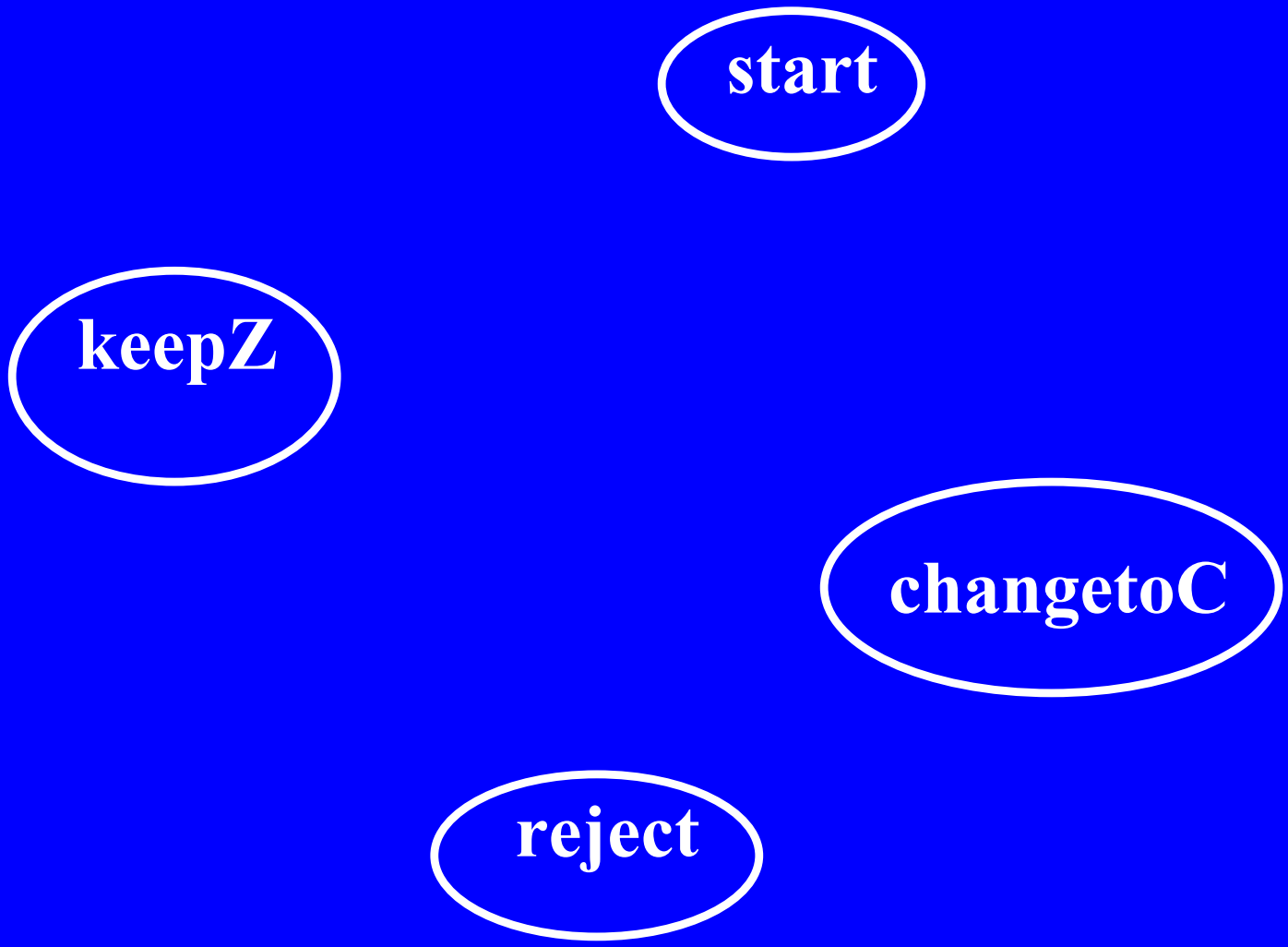
  $z \rightarrow c$ before front vowels, z otherwise

  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:

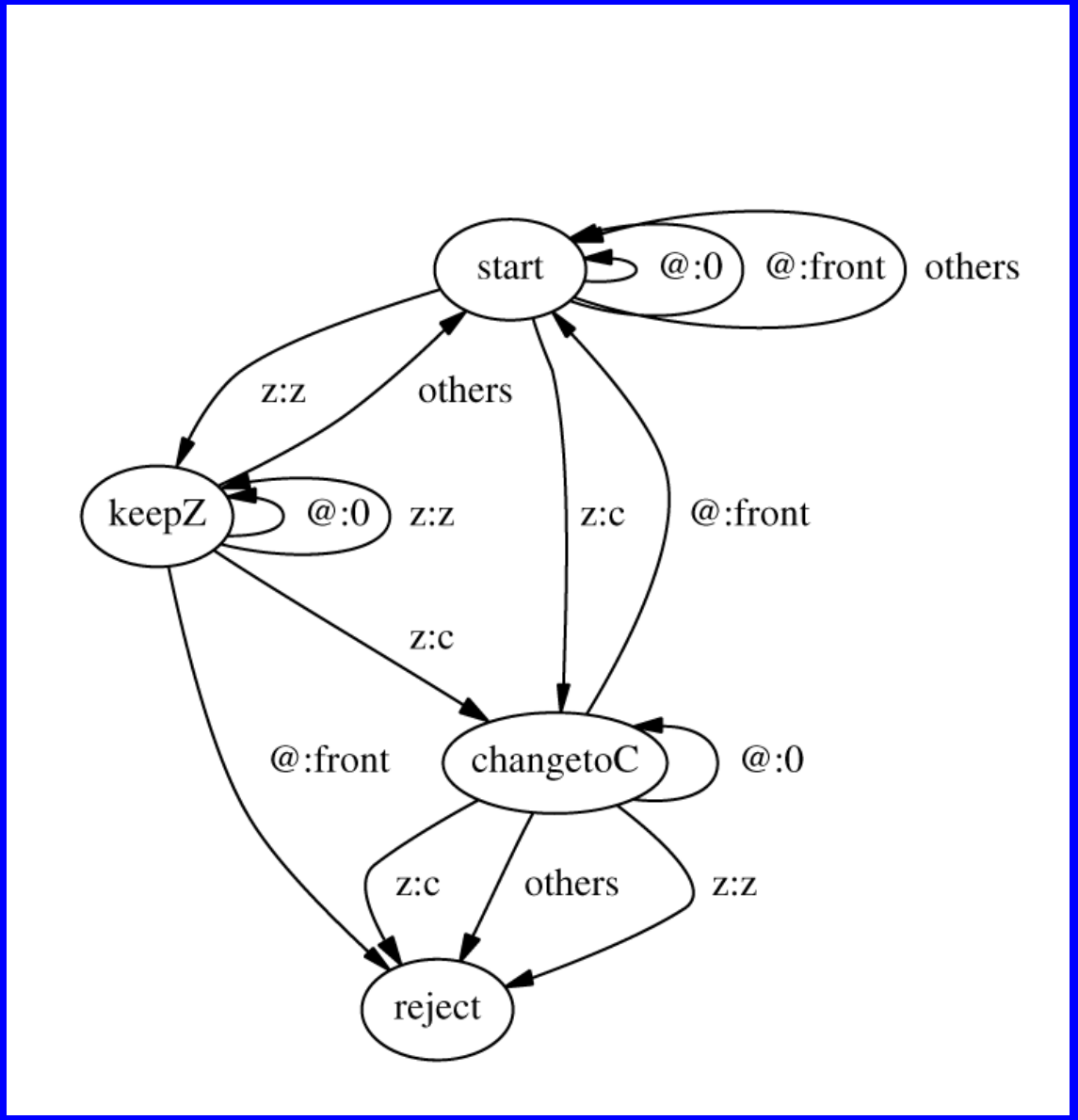
  lápiz, lápices (pencil, pencils) [*l^piz, l^pices*]

What's the automaton got to do?



Now add the arcs…

Automaton


start
@:0
@:front
others
z:z
others
keepZ
@:0
z:z
z:c
@:front
z:c
@:front
changetoC
@:0
z:c
others
z:z
reject

Z -c mutation

Fst description

```
machine "z -> c mutation“
state start
z:z keepZ
z:c changetoC
@:0 start
@:front start
others start

state keepZ
@:0 keepZ
z:z keepZ
z:c changetoC
@:front reject
others start

rejecting state changetoC
@:0 changetoC
z:z reject
z:c reject
@:front start
others reject
```

Corresponding state table

RULE "z -> c mutation" 3 5

z	**z**	**@**	**@**	**@**
z	**c**	**0**	**front**	**@**
1: 2	**3**	**1**	**1**	**1**
2: 2	**3**	**2**	**0**	**1**
3. 0	**0**	**3**	**1**	**0**

Phenomenon 3: pluralization

- Adding *s* to a noun that ends in a consonant forces a surface *e* to appear:

  ciudad (city); *ciudades*
- This can interact with other rules, e.g., *z-c-mutation:*

  lápiz, lápices
- Nouns ending in a vowel are not subject to this rule: *bota, botas*

The lexicon – take 2

- Add a gloss at the very end of the process, so as to return the feature list and 'translation', e.g., *venzo* [1p sg pres indic conquer, defeat] (first person, singular, present tense, indicative)
  - We'll show *how* to add this in a moment
- You will deal with two types of 'endings'

1. Noun endings: plural suffix +*s*
2. Verb endings: verb stem + tense markers

Simplest: infinitive marker `+ar, +er, +ir`

See table in pdf file for details: 5 x 3 table for Present tense; ditto for Subjunctive tense ("*I might…*")

Some implementation details

- For the automata, we will describe the character set, format of the .rul file, and the use of fst
- First 9 lines of .rul file: (*Note: fst* will insert these for you):

```
ALPHABET
a ^ b c C d e < f g . . .
NULL 0
ANY @
BOUNDARY #
RULE "BOGUS RULE FOR KIMMO BRAIN LOSSAGE" 1 33

 <automaton table>
RULE PLURALIZE
 <automaton table>
...
END
```

Instead of writing fst tables…

- You can use the program fst
- This lives in dir /fst/ in course locker
- To run:

  build fst type rules in file spanish.fst, then

```
./fst -o ~yourpath/spanish.rul ~yourpath/spanish.fst
```

- Format for fst rules:

FST rules

- "*b* after a vowel turns to *a*"

```
subset vowel a e
machine "bintoa"
state foo
vowel:vowel   bar
b:b  foo
c:c  foo
d:d  foo
others reject

rejecting state bar
b:e foo
b:b reject
others foo
```

The End