Morphology and Finite State Transducers

L545
Spring 2017
Morphology

- **Morphology** is the study of the internal structure of words
  - morphemes: (roughly) minimal meaning-bearing unit in a language, smallest “building block” of words

- Morphological parsing is the task of breaking a word down into its component morphemes, i.e., assigning structure
  - *going* \( \rightarrow \) *go* + *ing*
  - *running* \( \rightarrow \) *run* + *ing*

  • Note: spelling rules are different from morphological rules

- Parsing can also provide us with an analysis
  - *going* \( \rightarrow \) *go*:VERB + *ing*:GERUND
Kinds of morphology

• **Inflectional** morphology = grammatical morphemes that are required for words in certain syntactic situations
  
  - *I run*
  
  - *John runs*
    
    • **-s** is an inflectional morpheme marking 3rd person singular verb

• **Derivational** morphology = morphemes that are used to produce new words, providing new meanings and/or new parts of speech
  
  - *establish*
  
  - *establishment*
    
    • **-ment** is a derivational morpheme that turns verbs into nouns
Kinds of morphology (cont.)

• **Cliticization**: word stem + clitic
  - Clitic acts like a word syntactically, but is reduced in form
  - e.g., ‘ve or ‘d

• **Non-Concatenative** morphology
  - Unlike the other morphological patterns above, non-concatenative morphology doesn’t build words up by concatenating them together
  - Root-and-pattern morphology:
    • Root of, e.g., 3 consonants – *lmd* (Hebrew) = ‘to learn’
    • Template of CaCaC for active voice
      - Results in *lamad* for ‘he studied’
More on morphology

• We will refer to the stem of a word (main part) and its affixes (additions), which include prefixes, suffixes, infixes, and circumfixes.

• Most inflectional morphological endings (and some derivational) are productive – they apply to every word in a given class:
  - -ing can attach to any verb (running, hurting)
  - re- can attach to virtually any verb (rerun, rehurt)

• Morphology is more complex in agglutinative languages like Turkish:
  - Some of the work of syntax in English is in the morphology
  - Shows that we can’t simply list all possible words
Overview

A. Morphological recognition with finite-state automata (FSAs)

B. Morphological parsing with finite-state transducers (FSTs)

C. Combining FSTs

D. More applications of FSTs
A. Morphological recognition with FSA

• Before we talk about assigning a full structure to a word, we can talk about recognizing legitimate words

• We have the technology to do this: finite-state automata (FSAs)
Overview of English verbal morphology

• 4 English regular verb forms: base, -s, -ing, -ed
  - walk/walks/walking/walked
  - merge/merges/merging/merged
  - try/tries/trying/tried
  - map/maps/mapping/mapped

• Generally productive forms

• English irregular verbs (~250):
  - eat/eats/eating/ate/eaten
  - catch/catches/catching/caught/caught
  - cut/cuts/cutting/cut/cut
  - etc.
Analyzing English verbs

• For the –s & –ing forms, both regular & irregular verbs use base forms
• Irregulars differ in how they treat the past and the past participle forms

• So, we categorize words by their regularity and then build an FSA
  - e.g., walk = vstem-reg
  - ate = verb-past-irreg
FSA for English verbal morphological analysis

- $Q = \{0, 1, 2, 3\}; S = \{0\}; F = \{1, 2, 3\}$
- $\sum = \{\text{verb-past-irreg}, \ldots\}$
- $E = \{ (0, \text{verb-past-irreg}, 3), (0, \text{vstem-reg}, 1),$
  $(1, +\text{past}, 3), (1, +\text{pastpart}, 3),$
  $(0, \text{vstem-reg}, 2), (0, \text{vstem-irreg}, 2),$
  $(2, +\text{prog}, 3), (2, +\text{sing}, 3) \}$

NB: FSA for morphotactics, not spelling rules (requires a separate FSA):
  rules governing classes of morphemes
FSA Exercise: Isleta Morphology

- Consider the following data from Isleta, a dialect of Southern Tiwa, a Native American language spoken in New Mexico:

  - [temiban] ‘I went’
  - [amiban] ‘you went’
  - [temiwe] ‘I am going’
  - [mimiay] ‘he was going’
  - [tewanban] ‘I came’
  - [tewanhi] ‘I will come’
Practising Isleta

• List the morphemes corresponding to the following English translations:
  - ‘I’
  - ‘you’
  - ‘he’
  - ‘go’
  - ‘come’
  - +past
  - +present_progressive
  - +past_progressive
  - +future

• What is the order of morphemes in Isleta?

• How would you say each of the following in Isleta?
  - ‘He went’
  - ‘I will go’
  - ‘You were coming’
An FSA for Isleta Verbal Inflection

• $Q = \{0, 1, 2, 3\}; S = \{0\}; F = \{3\}$
• $\Sigma = \{\text{mi, te, a, wan, ban, we, ay, hi}\}$

• $E = \{(0, \text{mi}, 1), (0, \text{te}, 1), (0, \text{a}, 1),$
     $(1, \text{mi}, 2), (1, \text{wan}, 2),$
     $(2, \text{ban}, 3), (2, \text{we}, 3), (2, \text{ay}, 3), (2, \text{hi}, 3)\}$
B. Morphological Parsing with FSTs

• Using a finite-state automata (FSA) to recognize a morphological realization of a word is useful

• But we also want to return an analysis of that word:
  - e.g. given *cats*, tell us that it’s *cat* + N + PL

• A finite-state transducer (FST) do this:
  - Two-level morphology:
    • Lexical level: stem plus affixes
    • Surface level: actual spelling/realization of the word
  - So, for a word like *cats*, the analysis will (roughly) be:
    c:c  a:a  t:t  ε:+N  s:+PL
Finite-State Transducers

• While an FSA recognizes (accepts/rejects) an input expression, it doesn’t produce any other output
  - An FST, on the other hand, produces an output expression → we define this in terms of relations

• FSA is a recognizer; an FST translates from one expression to another
  - Reads from one tape, and writes to another tape
  - Can also read from the output tape and write to the input tape
    • FSTs can be used for both analysis and generation (bidirectional)
Transducers and Relations

• Goal: translate from the Cyrillic alphabet to the Roman alphabet

• We can use a mapping table, such as:
  - А : A
  - Б : B
  - Г : G
  - Д : D
  - etc.

• We define \( R = \{<A, A>, <B, B>, <Г, G>, <Д, D>, ..\} \)
  - We can think of this as a relation \( R \subseteq \text{Cyrillic X Roman} \)
Relations and Functions

• The **cartesian product** $A \times B$ is the set of all **ordered pairs** $(a, b)$, where $a$ is from $A$ and $b$ is from $B$

  $A = \{1, 3, 9\}$   $B = \{b, c, d\}$

  $A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$

  $= \{1, 3, 9\} \times \{b, c, d\}$

  $= \{(1, b), (1, c), (1, d), (3, b), (3, c), (3, d), ((9, b), (9, c), (9, d))\}$

• A **relation** $R(A, B)$ is a subset of $A \times B$

  $R1(A, B) = \{(1, b), (9, d)\}$

• A **function** from $A$ to $B$ is a binary relation where for each element $a$ in $A$, there is exactly one ordered pair with first component $a$.

• The **domain** of a function $f$ is the set of values that $f$ maps, and the **range** of $f$ is the set of values that $f$ maps to
The Cyrillic Transducer

\[ S = \{0\}; \quad F = \{0\} \]

\[(0, A:A, 0)\]
\[(0, Б:B, 0)\]
\[(0, Г:G, 0)\]
\[(0, Д:D, 0)\]

... 

- Transducers implement a mapping defined by a relation

- \( R = \{<A, A>, <Б, B>, <Г, G>, <Д, D>, ..\} \)

- These relations are called regular relations = sets of pairs of strings

- FSTs are equivalent to regular relations (akin to FSAs being equivalent to regular languages)
FSAs and FSTs

• FSTs, then, are almost identical to FSAs … Both have:
  - Q: finite set of states
  - S: set of start states
  - F: set of final states
  - E: set of edges (cf. transition function)

• Difference: alphabet for FST comprised of complex symbols (e.g., X:Y)
  - FSA: $\Sigma$ = a finite alphabet of symbols
  - FST: $\Sigma$ = a finite alphabet of complex symbols, or pairs
    • We can alternatively define an FST as using 4-tuples to define the set of edges E, instead of 3-tuples
    • Input & output each have their own alphabet

• NB: As a shorthand, if we have X:X, we often write this as X
FSTs for morphology

- For morphology, using FSTs allows us to:
  - set up pairs between the lexical level (stem+features) and the morphological level (stem+affixes)
    - \texttt{c:c a:a t:t +N:^ +PL:s}
  - set up pairs to go from the morphological level to the surface level (actual realization)
    - \texttt{c:c a:a: t:t ^:ε s:s}
    - \texttt{g:g o:o o:o s:s e:e ^:ε s:ε}
  - Can combine both kinds of information into the same FST:
    - \texttt{c:c a:a t:t +N:ε +PL:s}
    - \texttt{g:g o:o o:o s:s e:e +N:ε +SG:ε}
    - \texttt{g:g o:e o:e s:s e:e +N:ε +PL:ε}
**Isleta Verbal Inflection**

- I will go
- Surface: temihi
- Lexical: te+PRO+1P+mi+hi +FUTURE

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
<th>ε</th>
<th>mi</th>
<th>hi</th>
<th>ε</th>
</tr>
</thead>
<tbody>
<tr>
<td>te</td>
<td></td>
<td></td>
<td>+PRO</td>
<td>+1P</td>
<td></td>
</tr>
</tbody>
</table>

- Note: the cells line up across tapes:
- If an input symbol gives rise to more/less output symbols, epsilons are added to the input/output tape in the appropriate positions.
An FST for Isleta Verbal Inflection

• NB: teεε : te+PRO+1P is shorthand for 3 separate arcs …

• Q = {0, 1, 2, 3}; S ={0}; F ={3}

• E is characterized as:

0-> miεε : mi+PRO+3P  -> 1
  teεε : te+PRO+1P
  aεε : a+PRO+2P
1-> mi  -> 2
  wan
2-> banε : ban+PAST  -> 3
  weεε : we+PRES+PROG
  ayεε : ay+PAST+PROG
  hiε : hi+FUT
A Lexical Transducer

• FSTs can be used in either direction: property of **inversion**

• `leave+VBZ : leaves`
  `leave+VB : leave`
  `leave+VBG : leaving`
  `leave+VBD : left`
  `leave+NN : leave`
  `leave+NNS : leaves`
  `leaf+NNS : leaves`
  `left+JJ : left`

• Left-to-Right Input: `leave+VBD` ("upper language")
  Output: `left` ("lower language")

• Right-to-Left Input: `leaves` (lower language)
  Output: `leave+NNS` (upper language)
  `leave+VBZ`
  `leaf+NNS`
Transducer Example

• L1= [a-zA-Z]+  

• Consider language L2 that results from replacing any instances of "ab" in L1 by "x".  

• So, to define the mapping, we define a relation $R \subseteq L_1 \times L_2$  
  - e.g., <"abacab", "xacx">  

• Note: “xacx" in lower language is paired with 4 strings in upper language, "abacab", "abacx", "xacab", & "xacx"  

NB: ? = [a-zA-Z]\{a,b,x\}
C. Combining FSTs: Spelling Rules

• So far, we have gone from a lexical level (e.g., cat+N+PL) to a surface level (e.g., cats) in two steps
  - Or vice versa
• We’d like to combine those two steps
  - The lexical level of “fox+N+PL” corresponds to “fox’s”
  - And “fox’s” corresponds to “foxes”
• Start: make the two stages clearer
  - Note that, in the following, we’ll handle irregular plurals differently than before
  - We’ll basically follow Jurafsky & Martin, although there are other ways to do this.
Lexicon FST (1st level)

• The lexicon FST converts a lexical form to an intermediate form
  - dog+N+PL → dog^s
  - fox+N+PL → fox^s
  - dog+V+SG → dog^s
  - mouse+N+PL → mice … because no spelling rules apply
• This will be of the form:
  - 0-> f ->1 3-> +N:^ ->4
  - 1-> o ->2 4-> +PL:s ->5
  - 2-> x ->3 4-> +SG:ε ->6
  - and so on …
English noun lexicon as a FST (Lex-FST)

Expanding the aliases

J&M (1st ed.)
Fig 3.9

J&M (1st ed.)
Fig 3.11
Rule FST (2\textsuperscript{nd} level)

- The rule FST will convert the intermediate form into the surface form
  - dog^s \rightarrow dogs (covers both N and V forms)
  - fox^s \rightarrow foxes
  - mice \rightarrow mice

- Assuming we include other arcs for every other character, this will be of the form:
  - 0-> f ->0 1-> ^:\varepsilon ->2
  - 0 -> o ->0 2-> \varepsilon:e ->3
  - 0 -> x -> 1 3-> s ->4

- But this FST is too impoverished …
Spelling rule example

- Issues:
  - For *foxes*, we need to account for *x* being in the middle of other words (e.g., *lexicon*)
  - Or, what do we do if we hit an *s* and an *e* has not been inserted?

- The point is that we need to account for all possibilities
  - In the FST on the next slide, compare how word-medial and word-final *x*’s are treated, for example
E-insertion FST (J&M Fig 3.17, p. 64)

\[ \varepsilon \rightarrow e / \left( \begin{array}{c} x \\ s \\ z \end{array} \right)^{\_} s # \]
E-insertion FST

<table>
<thead>
<tr>
<th>f</th>
<th>o</th>
<th>x</th>
<th>^</th>
<th>s</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>o</td>
<td>x</td>
<td>e</td>
<td>s</td>
<td>#</td>
</tr>
</tbody>
</table>

- Trace:
  - generating foxes# from fox^s#:
    q0-f->q0-o->q0-x->q1-^:ε->q2-ε:e->q3-s->q4-#->q0
  - generating foxs# from fox^s#:
    q0-f->q0-o->q0-x->q1-^:ε->q2-s->q5-#->FAIL
  - generating salt# from salt#:
    q0-s->q1-a->q0-l->q0-t->q0-#->q0
  - parsing assess#:
    q0-a->q0-s->q1-s->q1-^:ε->q2-ε:e->q3-s->q4-s->FAIL
    q0-a->q0-s->q1-s->q1-e->q0-s->q1-s->q1-#->q0

Intermediate Tape
Surface Tape
Combining Lexicon and Rule FSTs

- We would like to combine these two FSTs, so that we can go from the lexical level to the surface level.

- How do we integrate the intermediate level?
  - Cascade the FSTs: one after the other
  - Compose the FSTs: combine the rules at each state
Cascading FSTs

• The idea of cascading FSTs is simple:
  - Input1 $\rightarrow$ FST1 $\rightarrow$ Output1
  - Output1 $\rightarrow$ FST2 $\rightarrow$ Output2

• The output of the first FST is run as the input of the second

• Since both FSTs are reversible, the cascaded FSTs are still reversible/bi-directional.
  - As with one FST, it may not be a function in both directions
Composing FSTs

• We can compose each transition in one FST with a transition in another
  - FST1: p0-> a:b -> p1       p0-> d:e ->p1
  - FST2: q0-> b:c -> q1       q0-> e:f -> q0

• Composed FST:
  - (p0,q0)-> a:c ->(p1,q1)
  - (p0,q0)-> d:f ->(p1,q0)

• The new state names (e.g., (p0,q0)) ensures that two FSTs with different structures can still be composed
  - e.g., a:b and d:e originally went to the same state, but now we have to distinguish those states
  - Why doesn’t e:f loop anymore?
Composing FSTs for morphology

• With our lexical, intermediate, and surface levels, this means that we’ll compose:

  - p2→ x → p3    p4→ +PL:s → p5
  - p3→ +N:^ → p4
  - p4→ ε:ε → p4 (implicit)
  - q0→ x → q1    q2→ ε:e → q3
  - q1→ ^:ε → q2    q3→ s → q4

• into:

  - (p2,q0)→ x →(p3,q1)
  - (p3,q1)→ +N:ε →(p4,q2)
  - (p4,q2)→ ε:e →(p4,q3)
  - (p4,q3)→ +PL:s →(p4,q4)
D. More applications of FSTs

• Syntactic (partial) parsing using FSTs
  - Parsing – more than recognition; returns a structure
  - For syntactic recognition, FSA could be used

• How does syntax work?
  - S $\rightarrow$ NP VP
  - D $\rightarrow$ the
  - NP $\rightarrow$ (D) N
  - N $\rightarrow$ girl
  - N $\rightarrow$ zebras
  - VP $\rightarrow$ V NP
  - V $\rightarrow$ saw

• How do we go about encoding this?
Syntactic Parsing using FSTs

FST1
S={0}; final ={2}
E = {(0, N:NP, 2),
(0, D:\epsilon, 1),
(1, N:NP, 2)}

Input

FST 1: NPs
FST 2: VPs
FST 3: Ss

<table>
<thead>
<tr>
<th>D</th>
<th>N</th>
<th>V</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>\epsilon</td>
<td>NP</td>
<td>V</td>
<td>NP</td>
</tr>
<tr>
<td>\epsilon</td>
<td>NP</td>
<td>\epsilon</td>
<td>VP</td>
</tr>
<tr>
<td>\epsilon</td>
<td>\epsilon</td>
<td>\epsilon</td>
<td>S</td>
</tr>
</tbody>
</table>
Noun Phrase (NP) parsing using FSTs

• If we make the task more narrow, we can have more success – e.g., only parse (base) NPs
  - The man on the floor likes the woman who is a trapeze artist
  - [The man]_{NP} on [the floor]_{NP} likes [the woman]_{NP} who is [a trapeze artist]_{NP}

• Taking the NP chunker output as input, a PP chunker then can figure out base PPs:
  - [The man]_{NP} [on [the floor]_{NP}]_{PP} likes [the woman]_{NP} who is [a trapeze artist]_{NP}