Building Blocks: Finite State Machines and Regular Expressions

Catherine Havasi & Rob Speer
MAS.S60 Day 3
Computational Morphology

Analysis

leaf N Pl  
leave N Pl  
leave V Sg3

leaves

Generation

hang V Past

hanged  
hung
Breaking up Words

• Suffixes
  – Dogs = Dog +s
  – Doer = Do + er
  – Beer != Be + er
Spelling Changes

• Bigger -> Big + er
• Unbelievable -> Un + believe + able
How many “words” for snow?

Lexical: Paris+mut+nngau+juma+niraq+lauq+sima+nngit+junga
Surface: Pari mu nngau juma nira lauq sima nngit tunga

Paris = (root = Paris)
+mut = terminalis case ending
+nngau = go (verbalizer)
+juma = want
+niraq = declare (that)
+lauq = past
+sim = (added to -lauq- indicates "distant past")
+nngit = negative
+junga = 1st person sing. present indic (nonspecific)

Figure 2: Inuktitut: Parimunngaujumaniralauqsimanngittunga = “I never said I wanted to go to Paris”
*-fixes

• Prefix
  – un-, anti-, etc.

• Suffix
  – -ity, -ation, etc

• Infix
  – Tagalog: um+hinigi + humingi (borrow)
  – Infixes in English?
Infixes are the shiznit
What do we need?

• Understand which units can glue to one another (morphotactics)
  – roots, affixes, suffixes

• What spelling changes are needed
  – Dropping the e, changing y to i

• **Lexical form vs. Surface form**
I know! Let’s use a finite-state machine!
What’s a finite state machine?
What’s a FSM?

• A concept from theory of computation
• Models formal languages using states and state transitions
• A Markov Chain is an FSM (with weighted random choices as inputs)
Formal languages vs. real languages

• The purpose of a state machine is to “recognize” a language
  – Decide whether a string is in the language or not
  – A language is simply a set of strings (probably infinite)

• English is not a formal language
Definition of a FSM

A finite-state automaton (FSA) is a quintuple $(Q, \Sigma, \delta, I, F)$ where

- $Q$ is a finite set of states
- $\Sigma$ is a finite set of terminal symbols, the alphabet
- $\delta \subseteq Q \times \Sigma \times Q$ the transition mapping
- $I \subseteq Q$ the initial states
- $F \subseteq Q$, the final states
Nondeterminism (NFA)

• There can be none, one, or more than one transition from a given state for a given possible input
• You can be in several states at one time
Roots and affixes

- Big, bigger, biggest
- cool, cooler, coolest, coolly
- red, redder, reddest
- clear, clearer, clearest, clearly
- happy, unhappy, unhappiest, happier
A FSM for for stems + ends

- Not that simple
  - complicated !-> complicateder
- Also: what does it do? Shouldn’t it output something?
Definition of finite-state transducers

\[(Q, \Sigma_1, \Sigma_2, \delta, I, F)\] where

- \(Q\) is a finite set of states (non-null);
- \(\Sigma_1\) a finite set of input symbols;
- \(\Sigma_2\) a finite set of output symbols;
- \(\delta \subseteq Q \times \Sigma_1^\epsilon \times \Sigma_2^\epsilon \times Q\), the transition mapping
- \(I \subseteq Q\), the initial states
- \(F \subseteq Q\), the final states
An Example
Epenthesi
Outfoxed!
Let’s build one for i-y
Let’s build one for i-y
It gets complicated
Beyond s and er

• Hypothesize a lexical form of a word that's different from how it appears on the surface
  – Conjunction in Spanish
  – Vowel harmony in Turkish
  – Pronunciation of Japanese
  – Conjugation in Hebrew
How do you combine these rules?

• Create a hierarchy of transducers
  – Xerox has patented the hell out of this, but the key patent expires in 2014

• Precombine all the transducers into one big one
  – Invented by Kimmo Koskenniemi
  – Only surface and lexical forms remain
  – Can be used in very low-memory environments
Regular Expressions

• Another use for formal languages
• Often relevant in sub-tasks of NLP
  – find boundaries between sentences
  – treat e-mail addresses as proper nouns
  – treat dollar amounts as nouns
  – recognize #hashtags, @-mentions, and retweets on Twitter
This guy has great slides.
Regular Expression Quick Guide

^ Matches the **beginning** of a line

$ Matches the **end** of the line

. Matches **any** character

\s Matches **whitespace**

\S Matches any **non-whitespace** character

* Repeats a character zero or more times

*? Repeats a character zero or more times (non-greedy)

+ Repeats a character one or more times

+? Repeats a character one or more times (non-greedy)

[aeiou] Matches a single character in the listed **set**

[^XYZ] Matches a single character **not in the listed set**

[a-z0-9] The set of characters can include a **range**

( Indicates where string **extraction** is to start

) Indicates where string **extraction** is to end
The Regular Expression Module

- Before you can use regular expressions in your program, you must import the library using "import re"

- You can use `re.search()` to see if a string matches a regular expression similar to using the `find()` method for strings

- You can use `re.match()` extract portions of a string that match your regular expression similar to a combination of `find()` and slicing: `var[5:10]`
The Find Command, Restyled

```python
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('From:', line):
        print line
```
Matching and Extracting Data

- The `re.search()` returns a True/False depending on whether the string matches the regular expression.

- If we actually want the matching strings to be extracted, we use `re.findall()`.

```
>>> import re
>>> x = 'My 2 favorite numbers are 19 and 42'
>>> y = re.findall('[0-9]+', x)
>>> print(y)
['2', '19', '42']
```

[0-9]+: One or more digits
Matching and Extracting Data

- When we use \texttt{re.findall()} it returns a list of zero or more sub-strings that match the regular expression

```python
>>> import re
>>> x = 'My 2 favorite numbers are 19 and 42'
>>> y = re.findall('[0-9]+', x)
>>> print(y)
['2', '19', '42']
>>> y = re.findall('[AEIOU]+', x)
>>> print(y)
[]
```
Warning: Greedy Matching

- The repeat characters (* and +) push outward (greedy) to match the largest possible string

```python
>>> import re
>>> x = 'From: Using the : character'
>>> y = re.findall('^[^F:+]', x)
>>> print y
['From: Using the :']
```

First character in the match is an F

Last character in the match is a :

Why not 'From:'?
Escape Character

- If you want a special regular expression character to just behave **normally** (most of the time) you prefix it with `\`

```python
>>> import re
>>> x = 'We just received $10.00 for cookies.'
>>> y = re.findall('\$[0-9.]+', x)
>>> print(y)
['$10.00']
```

- `\$[0-9.]+`: A real dollar sign \$ followed by one or more digits or periods
- At least one or more

---

- A digit or period
Regular expressions for the win!

- Suppose social media users are commenting on events
- If it’s a negative event, they say #TopicFail
- If it’s a positive event, they say #TopicFTW
- The topic might have CamelCaseWords
- Your task: recognize these strings and separate the words
- If time: grep, puzzles