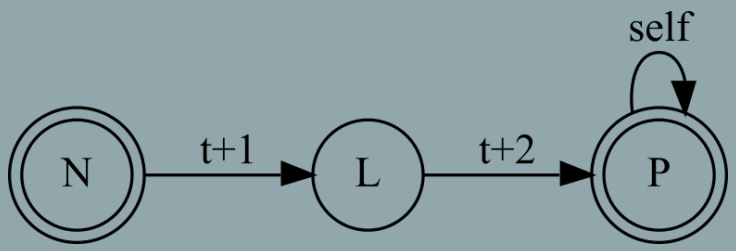
LING-362

# Introduction to **N**atural **L**anguage **P**rocessing

Eliza II / Finite State Methods I



# Recap: Dictionaries

- New empty dictionary:

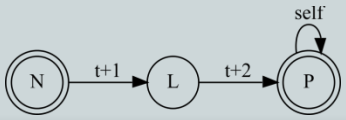
```
prefixes = {}
```

- Or with some initial values already:

```
prefixes = {"+1":"US", "+972":"IL", "+63":"PH"}
```

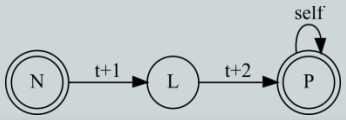
- Adding and accessing values:

```
prefixes["+52"] = "MX"
print(prefixes["+52"])
'MX'
```



# Better phone scraper: intl. prefix dictionary

```
# Make a prefix dictionary
prefixes = {}
prefixes["+1"] = "US"
prefixes["+52"] = "MX"
prefixes["+63"] = "PH"
prefixes["+972"] = "IL"
```

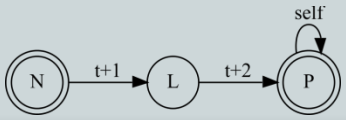


# Better phone scraper: intl. prefix dictionary

```
# Regex with two groups this time:
lines = text_about_phones.split("\n")
phone_pattern = r'\(?(\+[0-9]+)?\)? ?((\(?[0-9]+\)? ?)+)'
```

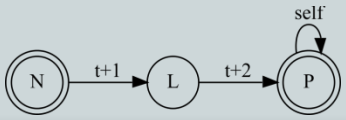
(+972)

(123) 456 (789)



# Better phone scraper: intl. prefix dictionary

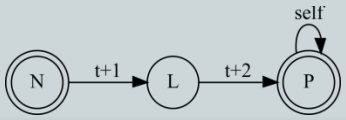
```
for line in lines:
    match = re.search(phone_pattern,line)
    if match is not None:
        prefix = match.group(1) # The prefix
        if prefix is not None: # Could be None, since it's (…)?
            if prefix in prefixes: # Could be an unknown prefix
                country = prefixes[prefix]
            else:
                country = "UNKNOWN"
        else:
            country = "UNKNOWN"
        phone_number = match.group(2)
        phone_number = clean_number(phone_number)
        print(country + " " + phone_number)
```



# Output

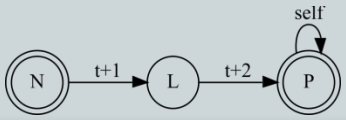
***> python phones_prefixes.py***

```
MX 015512345678
MX 015512345678
MX 5512345678
PH 45551234
IL 26333333
US 2021234567
```



# Pig Latin

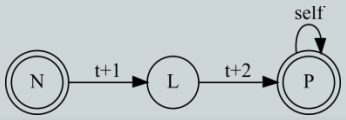
```
def pig_latin(human_word):

    edited_word = human_word
    if re.search(r'^[aeiou]',edited_word) is not None:
        edited_word += "hay"  # a += b  is same as: a = a + b
    else:
        edited_word = re.sub(r'^([^aeiou])(.*)',r'\2\1ay',edited_word)
    return edited_word
```



# Recap: while loop

- Keeps running, checks condition every new run
- Indentation indicates scope, like **for** loop
- Note nested indentation for **if** inside loop

```
count = 0
words = ["I","like","cake",",","too","."]
output = []
while count < 3:   # Get first 3 words
    output.append(words[count])
    count += 1
```

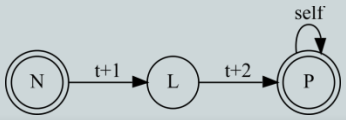


# Did you find any problems with Eliza?

**Problems:**
Morphology
Syntax
Semantics
Pragmatics
Memory?

**Solutions:**
More patterns
More reflections
Completely new mechanisms

```
>I want to challenge myself
If you got to challenge myself, then what would you
do?
```



# Making Eliza better

- Some ideas:
  - Talking about brothers and sisters (be conservative!)
  - Support for reflexive pronouns to fix this:

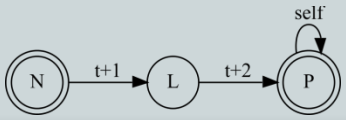
```
>I want to challenge myself
If you got to challenge myself, then what would you do?
```

  - A pattern to make Eliza repeat what you said:

```
> Just tell me "you're beautiful".
you're beautiful
```

  - … more clever stuff??

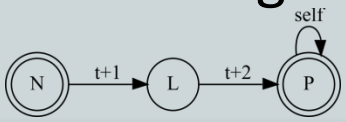
- Brainstorm at:
  https://corpling.uis.georgetown.edu/etherpad/p/eliza



# Homework – nltk+regex practice

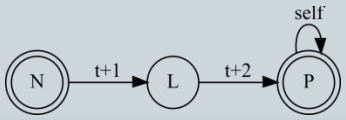
**Finding negations**
(submit **before class Wednesday**)

- You are given a poem in a text file
  - Read the file into Python using **argparse** [2 points]
  - Tokenize it with **NLTK** [2 points]
  - Loop through tokens, using **re** to find and print all **negations**: [2 points]
    - *No, not, n't, never, none, nobody, …*
    - Adjectives in: *un-, non-, im-, in-…*
    - You should catch these in upper and lowercase!!
  - Extra credit: use a **dictionary** to track frequencies of each negative word and **print the frequencies** [2 point]



# From regex to natural language

- Regular expressions describe a simple type of grammar
  - For example, you could think of a regex:
    /D?A*N/
  - As describing a Noun Phrase:
    Determiner? (Adj)* Noun
  - Just replace each noun with N, each Adj with A…
  - We can now recognize noun phrases!
    *(why should we want to?)*



# From regex to natural language

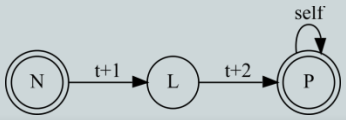
- In fact, syntax is more complex than what we can express with regex:

  *pick the kids up*: /VDNP/

  - But only certain verbs take certain particles, objects…
  - Can't prevent matching:

  *sleep the kids up*

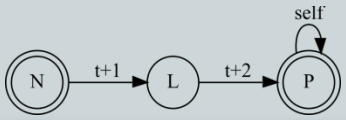
  *pick the kids over*



# From regex to natural language

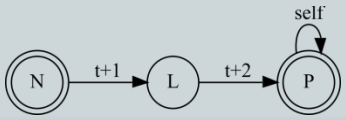
- But for **morphology**, word formation is often describable using something like regex:
  - Super anti adverbs: /(super)?(anti)?ADJ(ly)?/
  - Noun compounds: /N+N/
- But what is ADJ? or N?
- Can we do regex with a different 'alphabet'?

- A grammar of **re**xpressions using any 'alphabet' is called a **regular language**



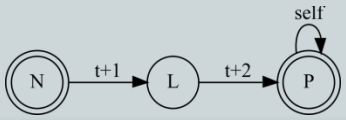
# Regular languages

- In fact we can create a regular language grammar using: (see reading from **J&M 2008**)
  - Some alphabet Σ with symbols a, b, c…
  - Any single symbol is a possible regular grammar (just a)
  - Any union (a OR b), concatenation (a THEN b) or Kleene star (a*) of a symbol or language
- Using these constraints, we can build any regular grammar using any set of symbols



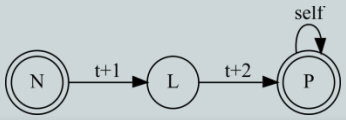
# Finite State Automaton - FSA

- Each **FSA** can **recognize** a certain language, using:
  - A finite number of states, including start and end
  - Transitions depending on input
- More formally:
  - FSA ≡ {Q, $q_0$, F, Σ, δ(q,i)}
- Where:
  - Q is a set of possible states $q_i$… $q_n$
  - $q_0$ is the starting state within Q
  - F is a subset of end states within Q
  - Σ is the alphabet
  - δ(q,i) is a set of allowable transitions from state q given input i



# Finite State Morphology

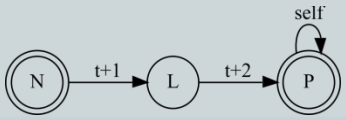
- Among most successful applications of FSA
- Popular in agglutinative languages (Turkish, Japanese), and highly inflected more or less concatenative ones (e.g. Slavic, Finnish)
- Basic tasks:
  - Morphological parsing
  - Generation



# From NL input to states

- Famous Turkish example (Jurafsky & Martin 2008, after Kemal Oflazer):
  - Uygarlaştıramadıklarımızdanmışsınızcasına
    civil-bec-caus-npot-part-pl-p1pl-abl-past-2pl-adv
    *"such that you can't be made civilized by us"*
    (civil-ize-ate-unable-ing-s-our-from-did-you-ly)

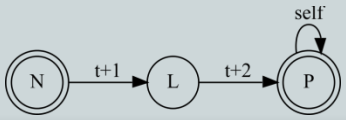
- Morphemes follow a **particular** order
- Many are **optional**
- Possible word formations can be described via states…



# Morphological parsing

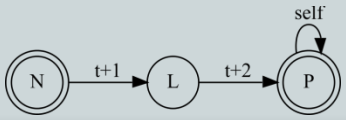
- The task:
  - Given some word in language X as input:
  - Output lexicon forms of constituents
  - Give morphological analysis to the units

- Ambiguity is possible:
  - friendly (ADJ) = friend:N + ly:ADJ
  - friendly (ADV) = friend:N + ly:ADJ + 0:ADV
    (for ?friendlyly, Bauer 1992)
- In ambiguous cases: give all possible analyses
- One way of dealing with **O**ut **O**f **V**ocabulary items



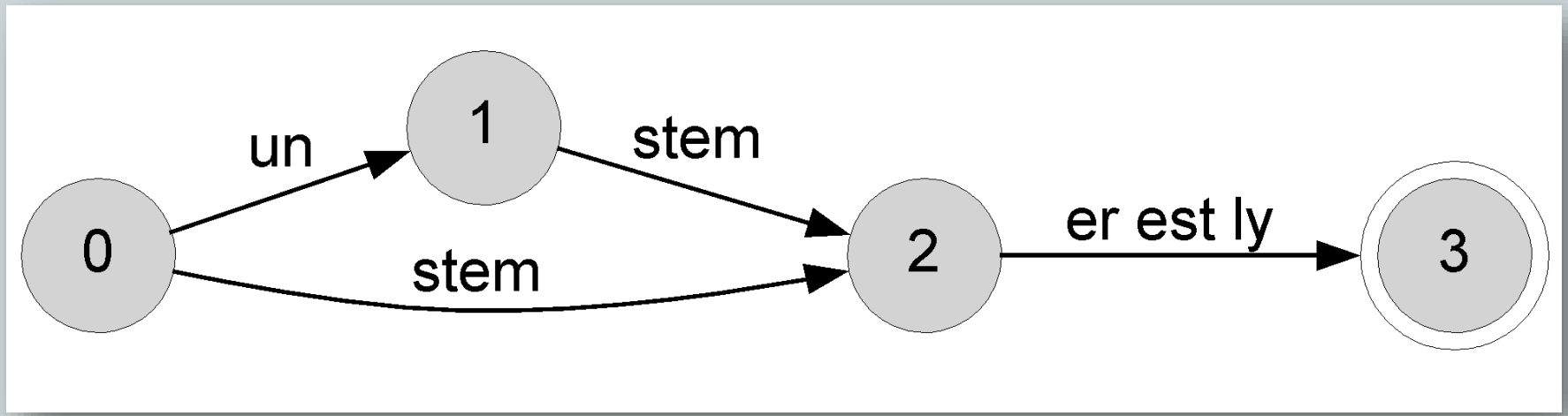
# English adjectives

- What would we need to model forms like these?
  - *happy, happier, unhappy, happily, unhappily*
  - *lucky, luckiest, unlucky, luckily, unluckily*
  - *big, bigger, biggest*
  - *…*
- What is the alphabet like?
- What transitions are possible?

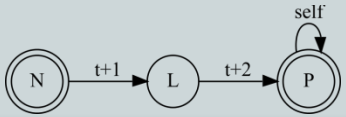


# First approximation

- A first approach would be to model states for each morpheme
- Allow transitions based on order (Antworth 1990)

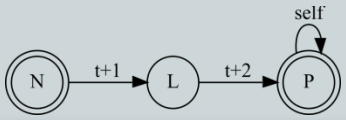



- Problems?



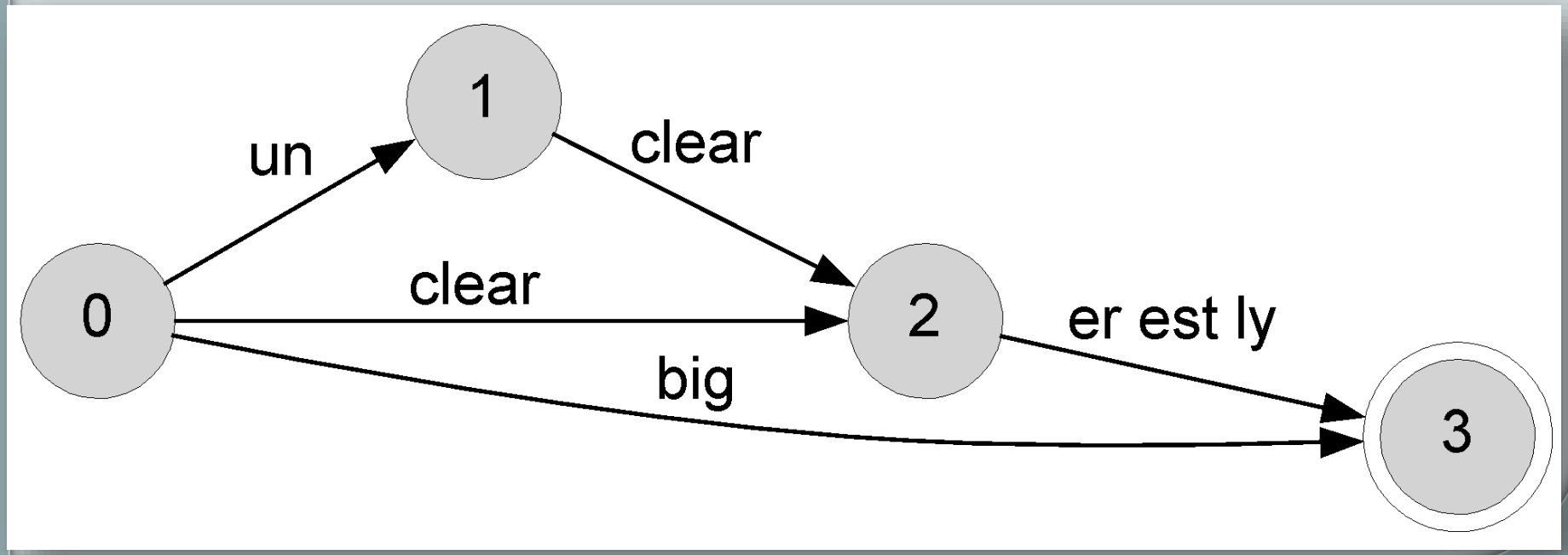
# Problems

- Some strange forms will be possible:
  - *unbiggest*
  - *bigly*
  - *…*
- Orthography would need to be handled:
  - *happyer*
  - *happyly*



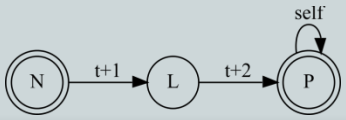
# Solutions

- Automata must become more complex to model the phenomenon
- Just the beginning:



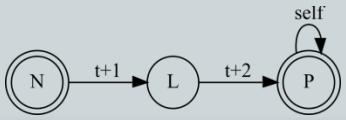
# Writing automata

- Many frameworks exist for FSM
- Influential early framework: Xerox FSM (XFSM)
  - Beesley & Karttunen (2003)
- Many free (re)implementations:
  - HFSM, Foma, OpenFST/PyFST
  - Compiled in C++ for performance
  - Bindings for Python available (though may be tricky to compile, OS dependent)
- We will use a simple version in Python



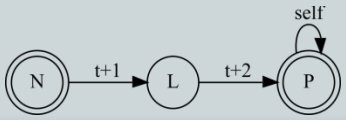
# Concatenating and regex

- The easiest way to represent morphology with automata: **composition**
- Simply concatenate automata to recognize complex expression:
  - Input1 -> Automaton1
  - Input2 -> Automaton2
  - Input1Input2 -> Automaton1+Automaton2
- Can be implemented using **regex concatenation** (a way of describing *some* FSAs)



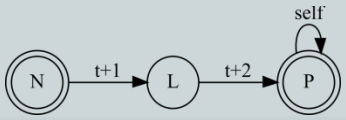
# Regex as symbol names

```
import re

first = r"(Bobby|Amir)"
last = r"(Zeldes)"

composed = "^" + first + last + "$"

print(re.search(composed, "BobbyZeldes"))
print(re.search(composed, "BobbySchmidt"))
```



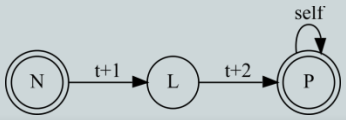
# Let's try the English adjectives

- Suppose adjectives look like this:
  - Can start with ***un-***
  - Have a stem like ***big*** or ***clear***
  - Can end in ***-er***, ***-est***
- Can we compose a three part regular expression to identify these?



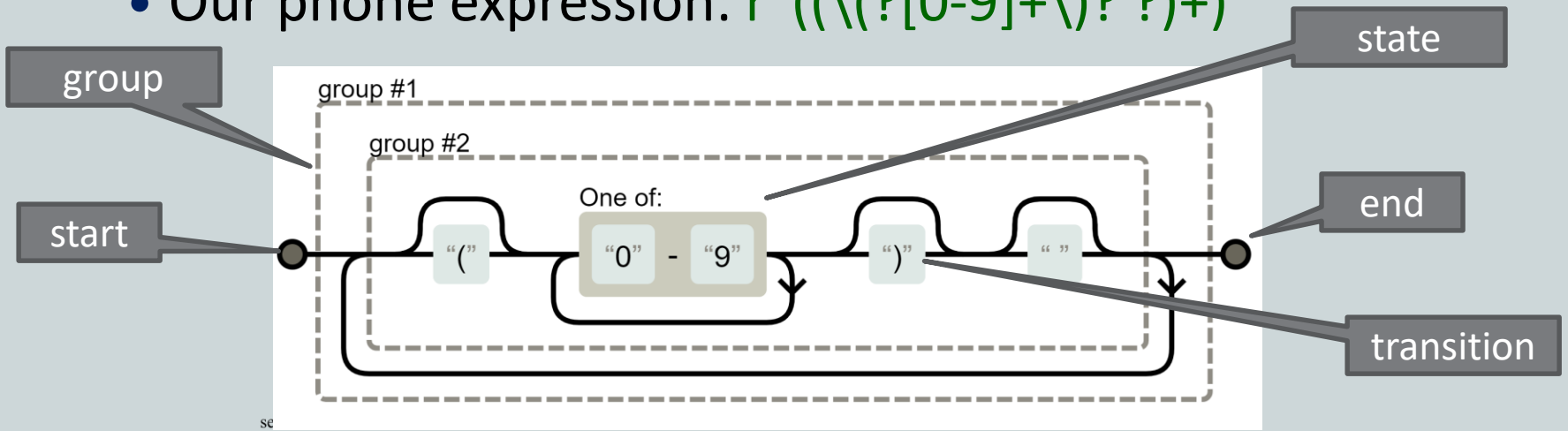
# Solution

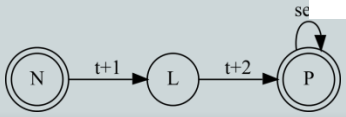
```
import re

un = r"(un)?"
stem = r"(big|clear)"
suffix = r"(er|est)?"

composed = "^" + un + stem + suffix + "$“

# Now we can recognize if a word is such an adjective!
```



# Visualization

- It can be useful to plot out automata
- We will see multiple plot types later
- For simple regex strings you can use:
  - https://regexper.com/
  - Our phone expression: r"((\(?[0-9]+\)? ?)+)"



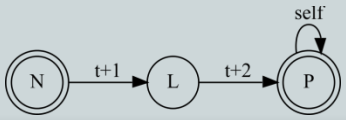



# What about generation?

- We can use library **exrex** to randomly walk through regex states

  *> python -m pip install exrex*

```
import exrex
# Generate some forms
print("Generating all forms:\n")
outputs = exrex.generate(composed)

for output in outputs:
    print(output)
```

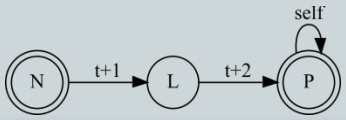


# A word about generators

- exrex.generate() returns something that looks like a list
- But:

```
adjectives = exrex.generate(r"(un)?(big)(er)?")
print(adjectives[0])
```

{TypeError}'generator' object is not subscriptable

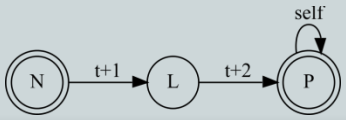


# A word about generators

- Generators act like lists in ***for*** loops
- But they only fetch one item at a time
- Save memory by not representing whole list
- Can be **converted** to lists:

```
print(list(adjectives)[0])
```

***big***



# Looping through our generator

Generating all forms:

- big
- biger
- bigest
- clear
- clearer
- clearest
- unbig
- unbiger
- unbigest
- unclear
- unclearer
- unclearest