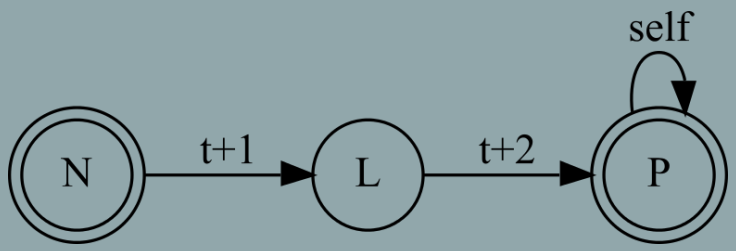
LING-362

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

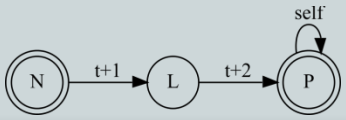
Finite State Methods (ctd.)



Information Session

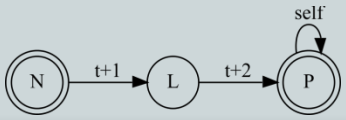
Data Science and Analytics MS and BS-MS Programs / Ami Gates

- Time: **Nov 15, 2021 12:30 PM**
- Zoom Meeting:
  - https://georgetown.zoom.us/j/91335226763



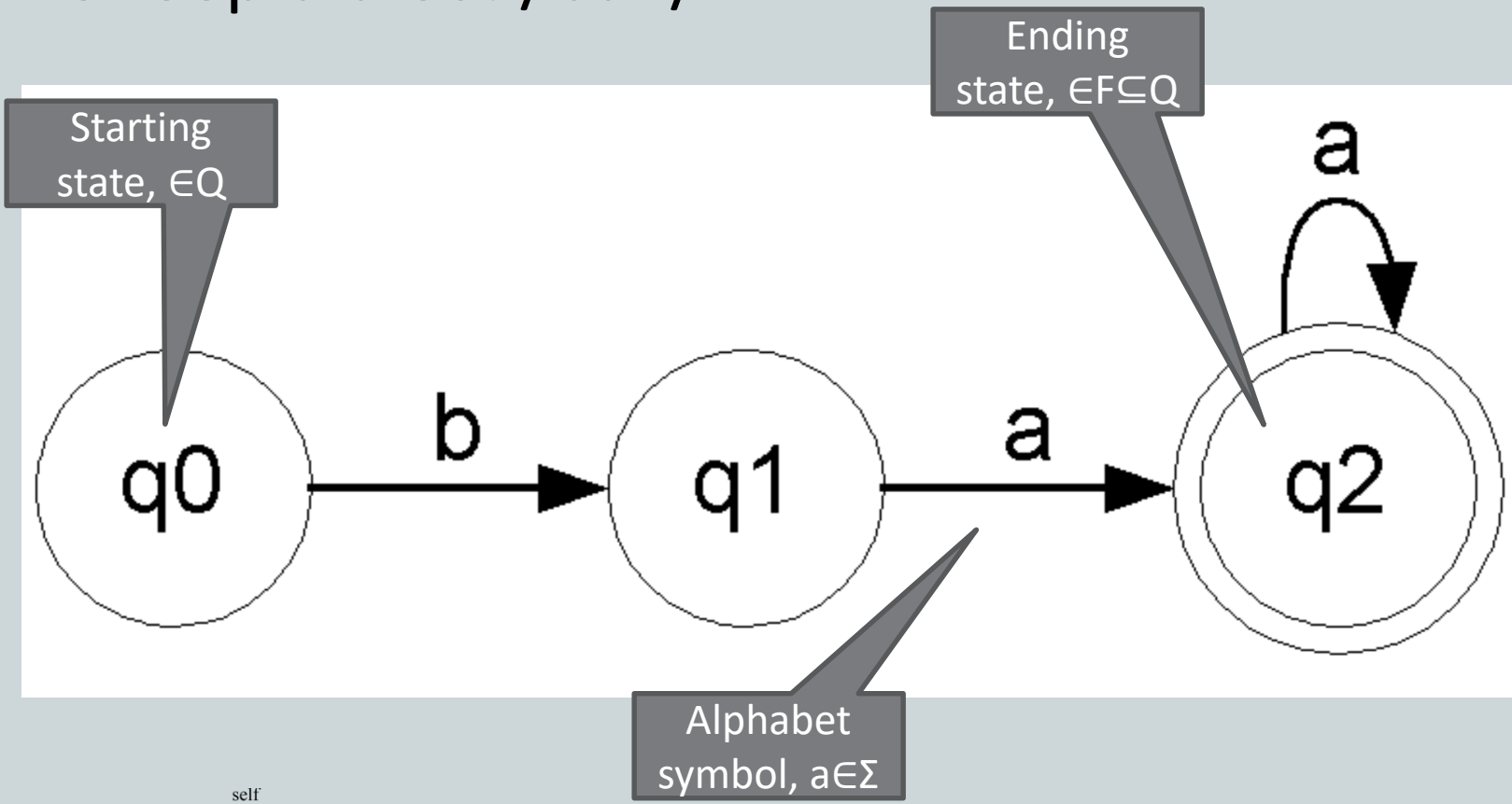
Finite State Automata

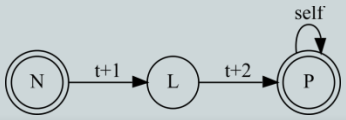
- Each **FSA** can **recognize** a certain language, using:
  - A finite number of states, including start and end
  - Transitions depending on input
- More formally:
  - FSA ≡ {Q, q_0, F, Σ, δ(q,i)}
- Where:
  - Q is a set of possible states q_i… q_n
  - q_0 is the starting state within Q
  - F is a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



FSA for the sheep language

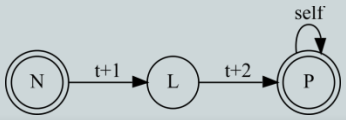
- Sheep dialect /ba+/:





Finite State Morphology

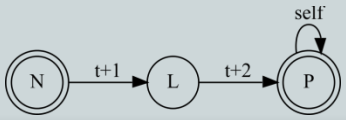
- Among most successful applications of FSA
- Popular in agglutinative languages (Turkish, Japanese), and highly inflected more or less concatenative ones (e.g. Slavic, Finnish)
- Basic tasks:
  - Morphological parsing
  - Generation



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)

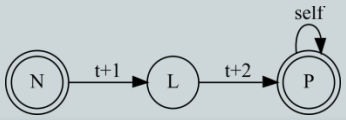
- Morphemes follow a **particular** order
- Many are **optional**
- Possible word formations can be described via states…



Morphological parsing

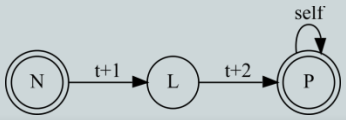
- The task:
  - Given some word in language X as input:
  - Output lexicon forms of constituents
  - Give morphological analysis to the units

- 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



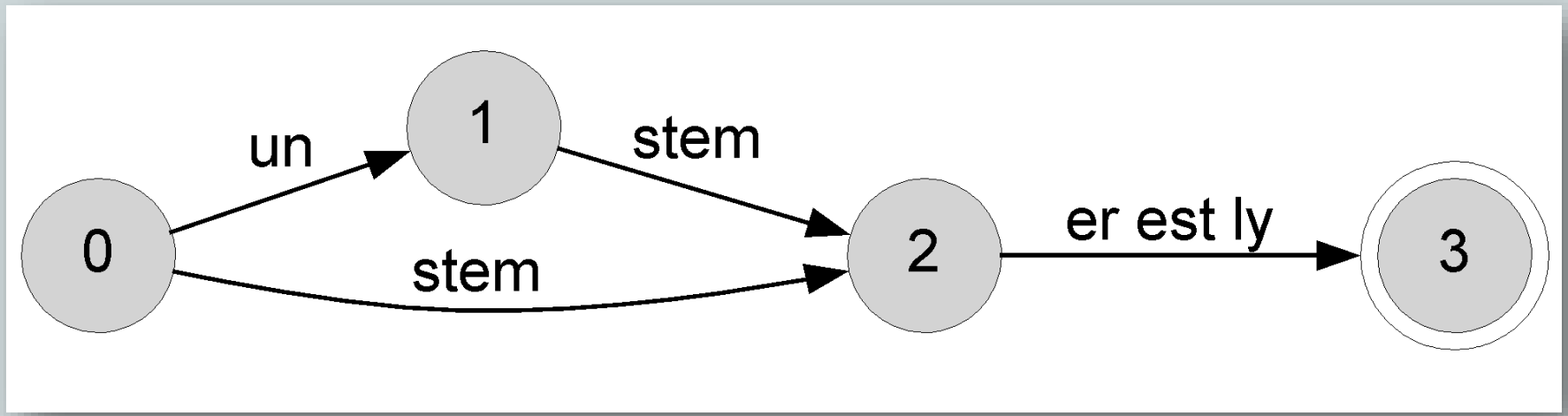
English adjectives

- 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?

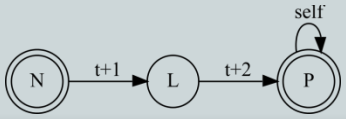


First approximation

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

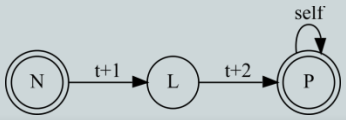


- Problems?



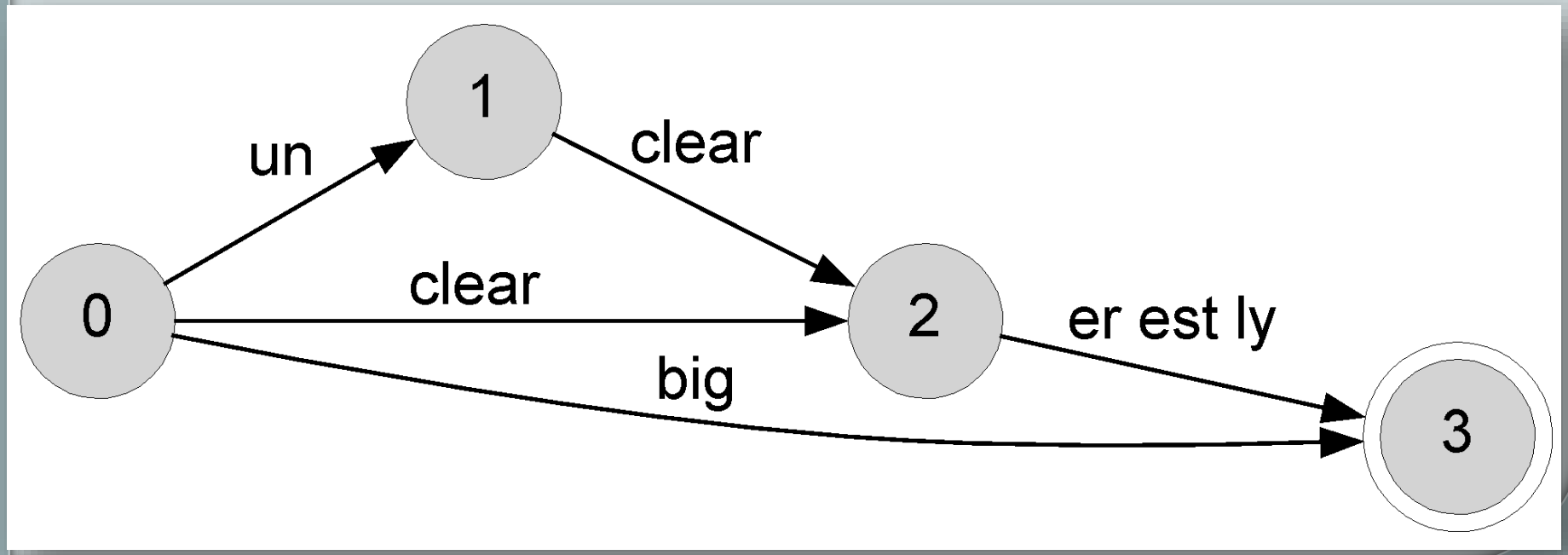
Problems

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



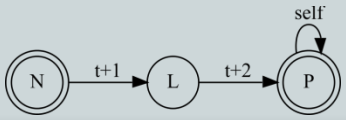
Solutions

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



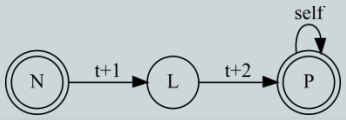
Writing automata

- 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



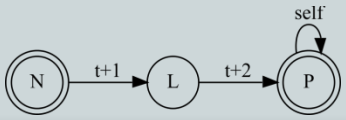
Concatenating and regex

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



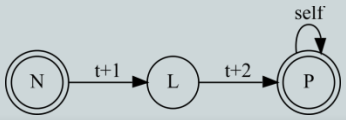
Regex as symbol names

```
import re

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

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

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



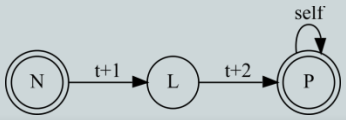
Let's try the English adjectives

- 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?



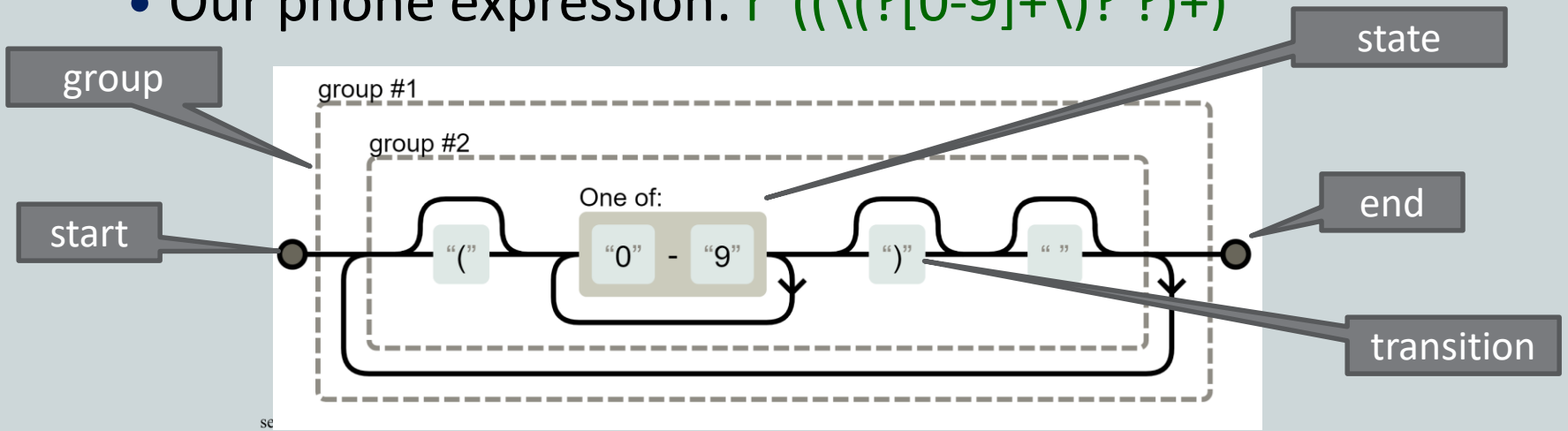
Solution

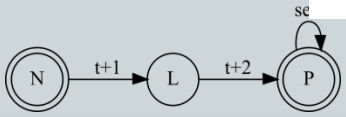
```
import re

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"((\(?[0-9]+\)? ?)+)"



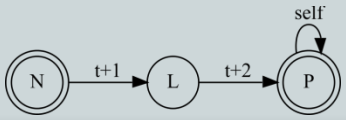


What about generation?

- 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)
```

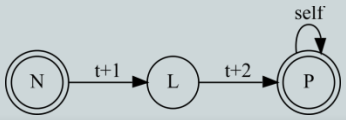


A word about generators

- 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

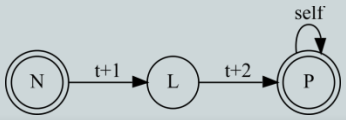


A word about generators

- 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])
```

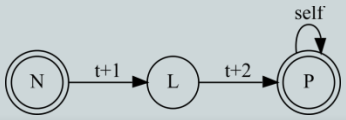
big



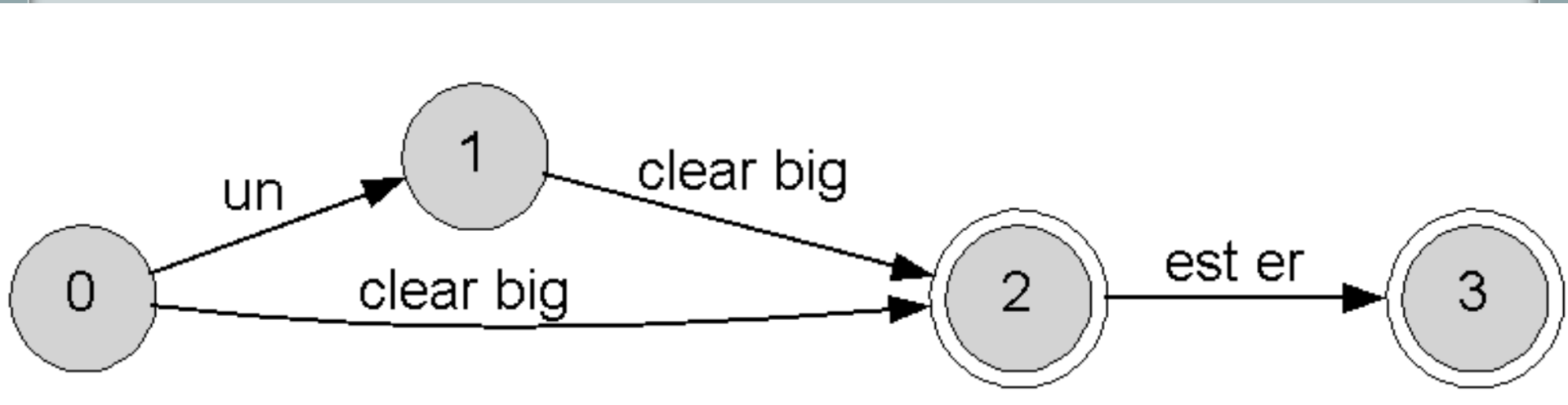
Looping through our generator

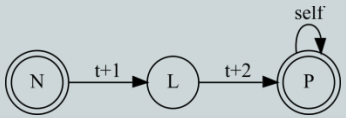
Generating all forms:

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



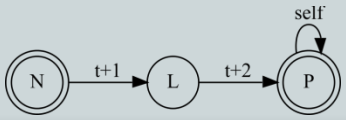
FSA – where are final states?





Is concatenation all we need?

- Some word formations are more complex:
  - Circumfixes: **en-/em-** **-en**
    - *enliven*
    - *embolden*
    - *embiggen*
  - Reduplication:
    - *no-no*
    - *night-night*

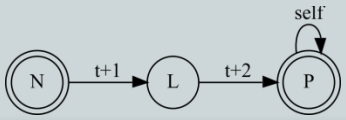


Is concatenation all we need?

- These processes are not very productive in English, but they are in other languages:
  - Japanese honorifics: *o*-VERB-*i-suru*
  - German past tense: *ge*-VERB-*t*
  - And recall the Greek perfect…
- Some languages have productive triplication!

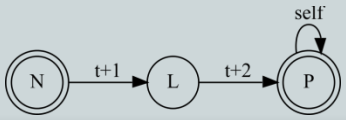
Pingelapese (Micronesia)
(Rehg 1981:11)

pei	*float*
peipei	*floating*
peipeipei	*still floating*



Nesting expressions is useful!

- Even for English, it can be necessary or just convenient to **nest** expressions
- Consider the 'wordy' part of URLs/e-mails
  - WORD = ([A-Za-z][A-Za-z0-9]*-?)+
- What's better?
  - Email = ([A-Za-z][A-Za-z0-9]*-?)+@([A-Za-z][A-Za-z0-9]*-?)\. ([A-Za-z][A-Za-z0-9]*-?)
  - Email = WORD+@WORD.WORD
- Can be reused for URLs, file nams…



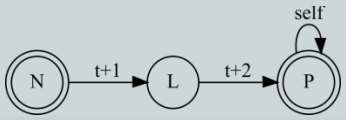
Nesting with f-strings

- Since Python 3.6, we can embed variables into strings using **f-strings:**

```
username = "Amir"
message = f"Hello {username}!"
print(message)
```

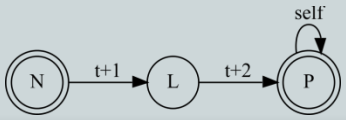
- f-strings also automatically convert to str():

```
iteration = 1
print(f"running trial {iteration}")
```



Application in morphology

- As an example application of nesting in computational morphology, we will look at **numbers**
  - Infinite set
  - Important for some NLP (NLU & NLG) applications
  - Fairly easy to define using finite-state methods
  - Essentially impossible to capture perfectly using statistical methods (neural nets)
- Download ***Files > Code > fsm > numerals.py***

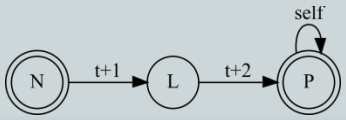


An example: English numerals

- Suppose we want to model the grammar of numbers from 1 – 99
- First we need one – nine:

```
# Adapted from Karttunen (2004)
import re, exrex

one_to_nine = "(one|two|three|four|five|six|seven|eight|nine)"
```

- What do the next numbers consist of?



An example: English numerals

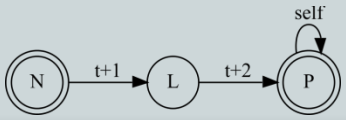
- Adding teen and ten prefixes:

```
teen_ten = "(thir|fif|six|seven|eigh|nine)"
```

- We have to add ten, eleven, twelve (unpredictable)
- But how do we combine "-teen" with each of the prefixes above? teens = (ten|eleven|twelve|???)

- Answer: use f-strings!

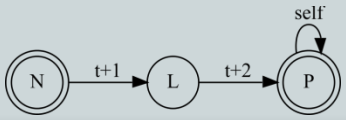
```
teens = f"(ten|eleven|twelve|(({teen_ten}|four)teen))"
```



An example: English numerals

- Now the ‘tens’ (numbers in X-ty)
- Remember they have the same prefixes as teens: thir(teen|ty)
- But we should also add “twen”(ty) and “for”(ty)

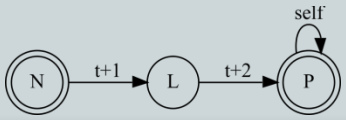
```
teen_ten = "(thir|fif|six|seven|eigh|nine)"
ten_stem = ?  # covers twenty, thirty, forty, fifty …
```



An example: English numerals

- Now the ‘tens’ (numbers in X-ty)
- Remember they have the same prefixes as teens: thir(teen|ty)
- But we should also add “twen”(ty) and “for”(ty)

```
teen_ten = "(thir|for|fif|six|seven|eigh|nine)"
ten_stem = f"({teen_ten}|twen|for)ty"
```



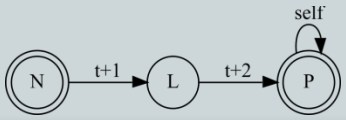
An example: English numerals

- Now we combine individual digits with tens to produce “twenty-three” etc.

```
tens = f"({ten_stem}(-{one_to_nine})?)“
one_to_ninety_nine = f"^({one_to_nine}|{teens}|{tens})$“
```

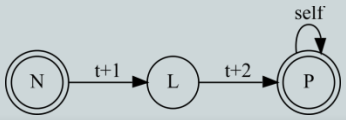
- Generate forms:

```
max_forms = 10
print(f"Generating {max_forms} random forms:\n")

for i in range(max_forms):  # range generates numbers up to its argument
    output = exrex.getone(one_to_ninety_nine)
    print(output)
```



Output

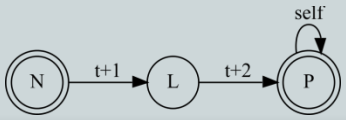
- Generating 10 random forms:
  - four
  - two
  - eight
  - twenty
  - forty
  - twenty-five
  - twelve
  - eleven
  - forty-one
  - twenty-two



NLU vs NLG

- We can also **recognize** or reject numbers:

```
# Test inputs
print("\nTesting inputs:\n")
inputs = ["ten","twenty-three","eleventy","fifty-ten"]

for word in inputs:
    if re.search(one_to_ninety_nine,word) is None:
        print("input " + word + " does not pass validation")
    else:
        print("input " + word + " is valid")
```



Output

- Testing inputs:
  - input ten is valid
  - input twenty-three is valid
  - input eleventy does not pass validation
  - input fifty-ten does not pass validation