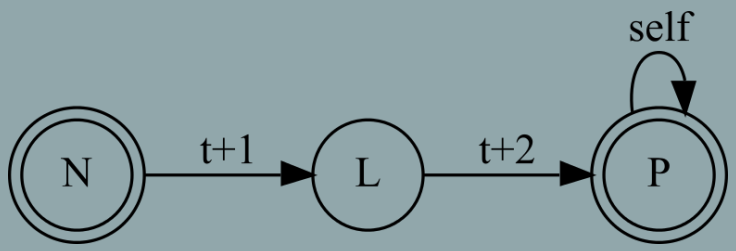
LING-362

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

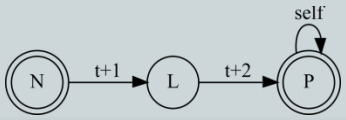
Finite State Methods (ctd.)



Python Coding Studio today!

- Join **guWeCode** on September 29th for an intro lesson on Python!
- If you are a beginner or just want a refresher, this is a great opportunity to sharpen your Python skills and meet others interested in coding. The session will be from 5-6:30pm in St. Mary's Room 120. We hope to see you there!

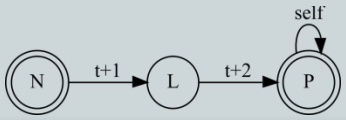
RSVP: https://forms.gle/cuhoPzt6TqMeUaSm8



GU CS grad research

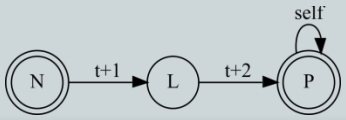
Friday, October 1 @1:30 PM ,
ZOOM: https://georgetown.zoom.us/j/94352180689

- Yang, Eugene and Lewis, David D. and Frieder, Ophir, **"Heuristic Stopping Rules For Technology-Assisted Review"**, Proceedings of the ACM Symposium on Document Engineering 2021 (DocEng '21) (2021)
- Yang, Eugene and Lewis, David D. and Frieder, Ophir, **"On Minimizing Cost in Legal Document Review Workflows"**, Proceedings of the ACM Symposium on Document Engineering 2021 (DocEng '21) (2021)
- Wang, Yanchen and Singh, Lisa, **"Analyzing the impact of missing values and selection bias on fairness"**, *Int J Data Sci Anal* 12**,** 101–119 (2021).
- Kornraphop Kawintiranon, Lisa Singh. **Knowledge Enhanced Masked Language Model for Stance Detection.** Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT 2021).



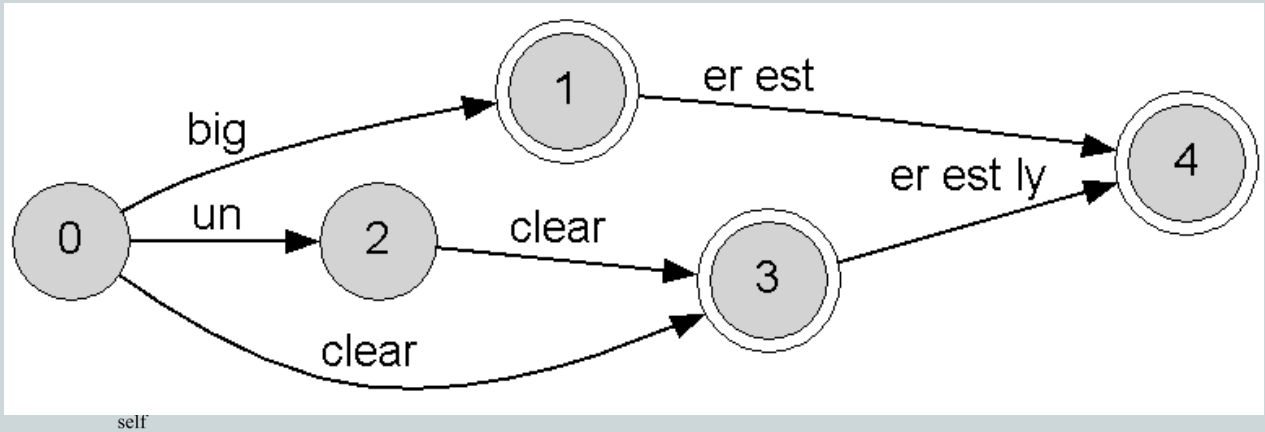
Finite state automata

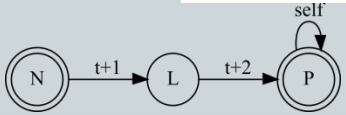
- Definition:
  - FSA ≡ {Q, q0, F, Σ, δ(q,i)}
- Where:
  - Q is a set of possible states qi… qn
  - q0 is the starting state within Q
  - F is a subset of end states within Q
  - Σ is the alphabet
  - δ(q,i) is a set of allowable transitions from state q given input i



Automata as graphs

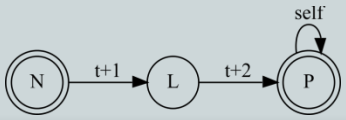
- States indicate steps in the derivation
- Morphology as transitions between lexicon items
  - What about linguistic order of processes?
  - Underlying and surface forms? (is -iest a suffix?)





Numerals

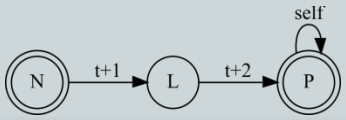
```
# Adapted from Karttunen (2004)
import re, exrex

one_to_nine = "(one|two|three|four|five|six|seven|eight|nine)"
teen_ten = "(thir|fif|six|seven|eigh|nine)"
teens = f"(ten|eleven|twelve|(({teen_ten}|four)teen))"
ten_stem = f"({teen_ten}|twen|for)ty"
tens = f"({ten_stem}(-{one_to_nine})?)"
one_to_ninety_nine = f"^({one_to_nine}|{teens}|{tens})$"
```



Numerals

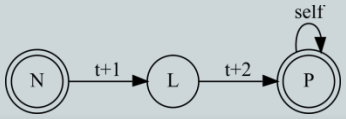
- Generate forms:

```
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

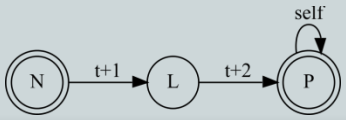
- four
- two
- eight
- twenty
- forty
- twenty-five
- twelve
- eleven
- forty-one
- twenty-two



NLU vs NLG

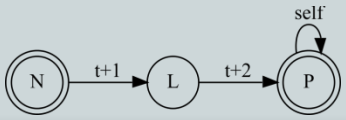
- We can also **recognize** or reject numbers:

```
# 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")
```



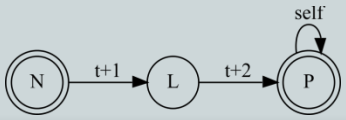
Output

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



From forms to analyses

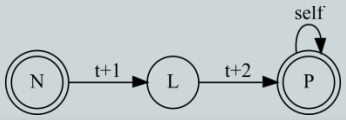
- So far we've been working to generate all possible forms in a grammar, but:
  - In NLP we usually want to analyze some natural language data: **text → analysis**
  - In NLG we want to create English realizations of underlying models: **analysis → text**



Better output

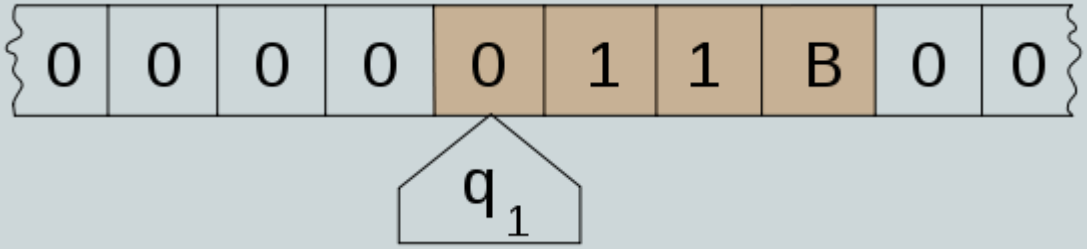
- FSAs give a simple kind of output: Booleans
- Either an input IS part of the language or it ISN'T:
  - grammatical == True / False
- Implementation in Python re:
  - **if** match **is not** None:

    …

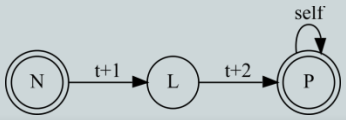


Getting output

- Our automaton goes over the input, symbol by symbol, and tries to find a valid set of states
- We can think of this as reading a tape with letters:

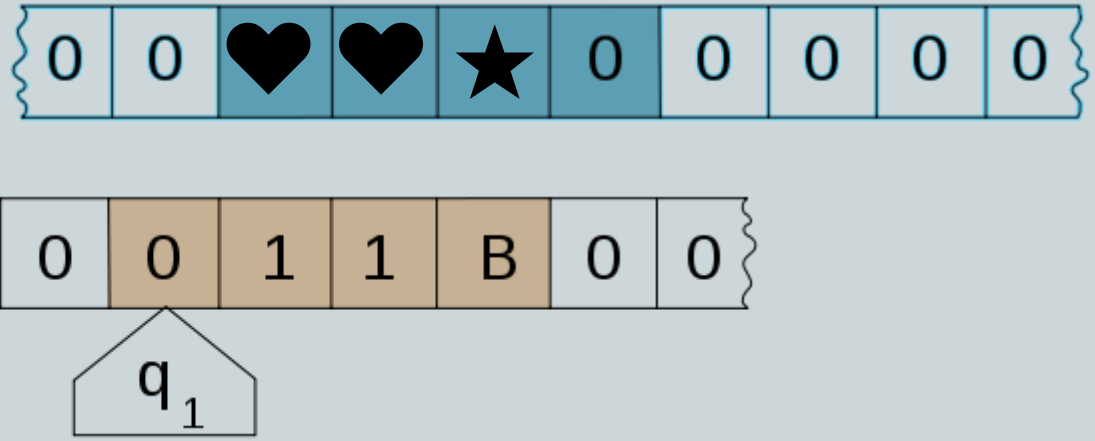


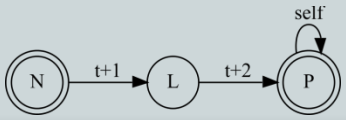
- But there's no place to give output…
- We need **another tape**!



Finite State Transducers (FSTs)

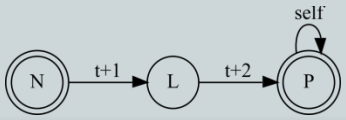
- A Finite State Transducer is a Finite Automaton with two tapes: **input** and **output** (also: **lower** and **upper**)





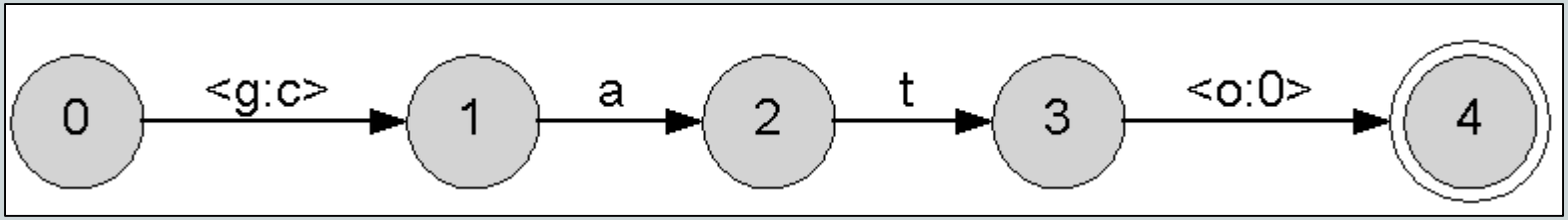
Formal definition

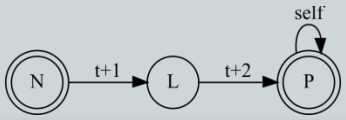
- FST ≡ {Q, q0, F, Σ, Δ, δ(q,i), σ(q,i)}
- Where:
  - Q is a set of possible states qi… qn
  - q0 is the starting state within Q
  - F is a subset of end states within Q
  - Σ is the input alphabet
  - Δ is the output alphabet
  - δ(q,i) is a set of allowable transitions from state q given input i, mapping to some states in Q
  - σ(q,i) is a set of allowable outputs given state q and input i



Transducers as graphs

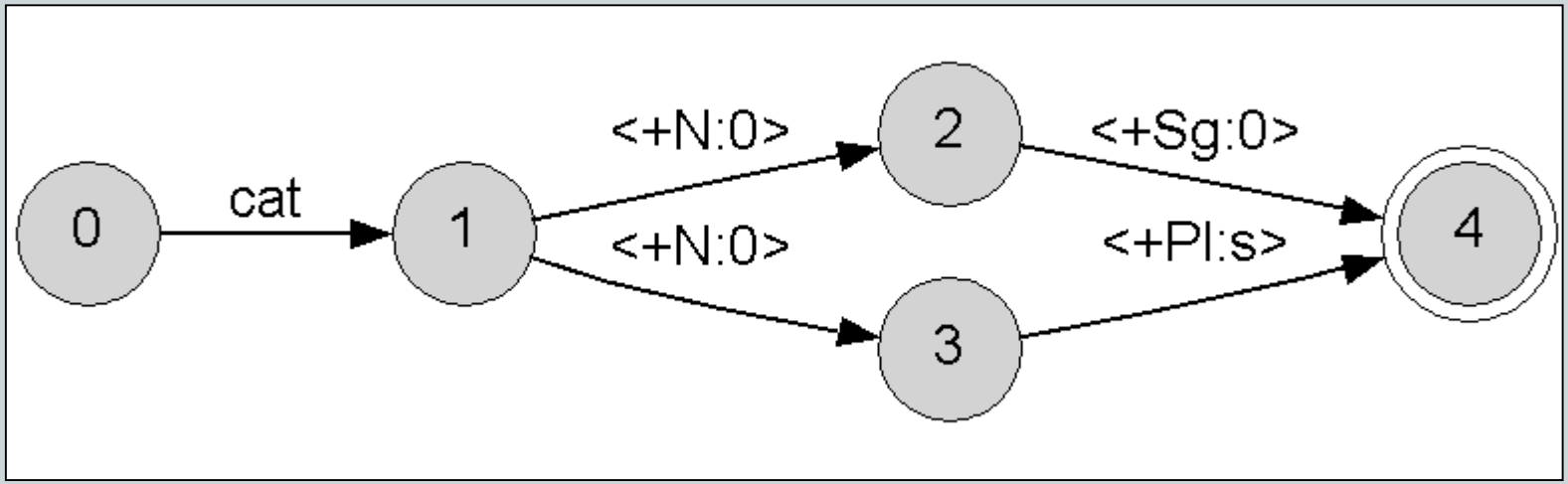
- FSTs can be expressed as graphs with symbol **translations** on the transitions
- Suppose we wanted to translate Spanish ***gato*** into English ***cat***:

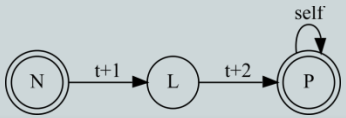




Transducers as analyzers

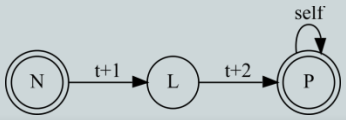
- The same idea holds for morphological analysis
- Translate a word into an analysis:





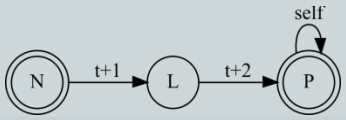
Regular Relations

- A regular relation describes:
  - for every state change in a regular automaton
  - a **finite set** of possible outputs
- Regular relations are like bilingual dictionaries for two regular languages
  - They allow **inversion** (we can go from L2 <> L1)
  - Allow **composition** (L1 > L2, L2 > L3 → L1 > L3)



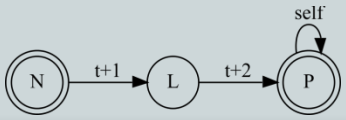
What are the alphabets for each tape?

- On the one side we have real words:
  - *cats*
  - *panicked*
  - *tries*
- On the other side?
  - …
- Where do we store all these words and symbols?



Can we do this with re.sub?

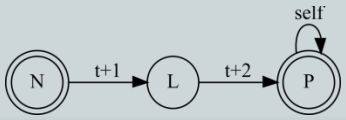
- Maybe, but…
  - For real-world systems we will not want to write 10K character regex substitutions
  - More convenient to store words and categories in a machine readable **lexicon**
  - re.sub is a bit different from textbooks FSTs:
    - Allows **capturing groups**
    - Inversion property not guaranteed
  - Python **re** is also not actually that efficient… (esp. generation from nulls or ‘epsilons’)



Enter the .lexc format!

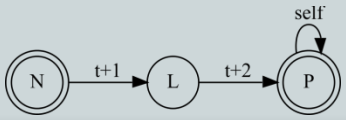
- Developed by Xerox/AT&T for XFST
- De-facto standard in most commercial FSMs
- Used for morphological analysis, date/other pattern recognizers, template generation…

- Basic idea:
  - Define symbols and categories
  - Cascade through a set of continuation categories
  - Output analyses as we go along
  - Invert for generation



Enter the .lexc format!

```
Multichar_Symbols +N +Sg +Pl

LEXICON Root

Noun ;

LEXICON Noun

cat   Ninf;

LEXICON Ninf

+N+Sg:0   #;
+N+Pl:s  #;
```

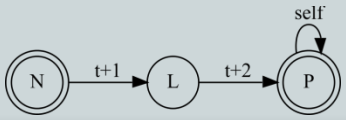


Now we can "translate"

- Download from Canvas (Code > fsm):
  - ***cat.lexc***
  - ***run_cat_lexc.py, fst.py***

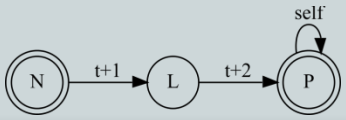
> python run_cat_lexc.py -h

```
usage: run_cat_lexc.py [-h] lexc inputfile

positional arguments:
  lexc          the .lexc file
  inputfile     an input text file consisting of the words
                to analyze, one per line

optional arguments:
  -h, --help  show this help message and exit
```



Now we can “translate”

```
p = argparse.ArgumentParser()
p.add_argument("lexc", help="the .lexc file")
p.add_argument("inputfile", help="an input text file")
options = p.parse_args()

# compile transducer
transducer = generate_table(options.lexc)
fst = FST(transducer)
fst.invert()  # Analysis, not generation

with open(options.inputfile, 'r', encoding="utf-8") as f:
        for line in f:
                print(fst.transduce(line.strip()))
```



Now we can “translate”

- Inverted (analysis)

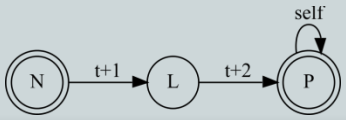
cat
->cat+N+Sg

cats
->cat+N+Pl

- Uninverted (generation)

cat+N+Sg
->cat

cat+N+Pl
->cats



A more complex lexicon

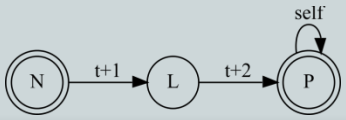
- Download ***english1.lexc*** from Canvas
- Some more words to play with:

Multichar_Symbols +N +V +PastPart +Past +PresPart +3P +Sg +Pl

LEXICON Root

Noun ;

Verb ;



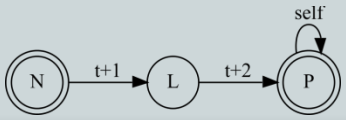
A more complex lexicon

LEXICON Noun

cat Ninf;
city Ninf;
fox Ninf;
panic Ninf;
try Ninf;
watch Ninf;

LEXICON Verb

beg Vinf;
fox Vinf;
make Vinf;
panic Vinf;
try Vinf;
watch Vinf;



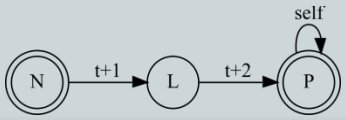
A more complex lexicon

LEXICON Ninf

+N+Sg:0 #;
+N+Pl:^s #;

LEXICON Vinf

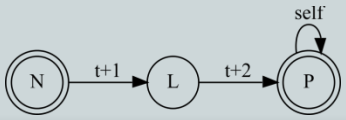
+V:0 #;
+V+3P+Sg:^s #;
+V+Past:^ed #;
+V+PastPart:^ed #;
+V+PresPart:^ing #;



Generating words on either tape

```
transducer = generate_table(options.lexc)
fst = FST(transducer)

print(fst.lower_words(n=3))
```

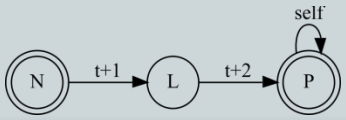
try+N+Pl
try+N+Pl
fox+N+Sg



Generating words on either tape

```
transducer = generate_table(options.lexc)
fst = FST(transducer)

print(fst.upper_words(n=3))
```

watch^s
try^ed
make^ing



Some problems

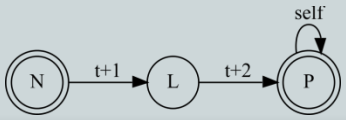
- We'll need some adjustment rules to fix these:

  watch^s

  try^ed

  make^ing

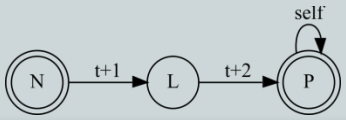
- These rules should apply at the morpheme juncture (^ symbol used in our lexicon)



Replacement rules

- We can use re.sub to clean up our outputs in a separate function:

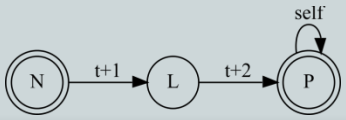
```
def clean_word(word):

    # e-deletion: make^ing -> mak^ing
    cleaned = re.sub(r'e\^(ed|ing)', r'^\1', word)

    # e-insertion: watch^s -> watche^s
    cleaned = re.sub(r'([szx]|ch|sh)\^s', r'\1^s', cleaned)

    # Remove remaining "^"
    cleaned = re.sub(r'\^', '', cleaned)

    return cleaned
```



Combining everything

```
# generate clean words from analyses
print("\nGenerating clean word forms:\n" + "="*20)
to_generate = open(options.inputfile, encoding="utf-8").read()
to_generate = to_generate.strip().split("\n")
for analysis in to_generate:
        generated = fst.transduce(analysis,with_input=False)
        generated = clean_word(generated)
        print(generated)
```

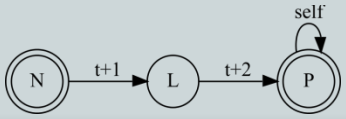
cats
watches
making



Epsilon insertion

- Note that analysis doesn't quite work, since we expect inputs like "cat^s"
- Pure C++ FSMs can consider producing such symbols from empty input, also called 'epsilon'
- For our pure Python code we can do this:

```
analyzed = fst.transduce(word,with_input=False)

# re.sub cannot invert caret deletion (epsilon insertion)
if analyzed=="" and re.search(r'(s|ed|ing)$',word) is not None:
        with_caret = re.sub(r'(s|ed|ing)$',r'^\1',word)
        analyzed = fst.transduce(with_caret)
```

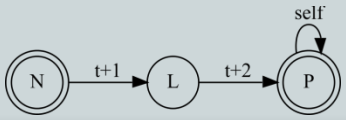


Exercise 1

- Add two y-replacement rules to fix these outputs:

  try+V+Past -> *tryed*

  city+N+Pl -> *citys*

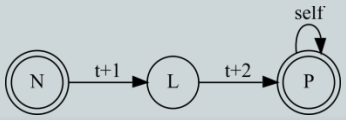


Exercise 2

- Add a K insertion rule to fix these outputs:

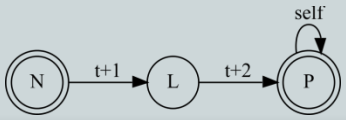
  panicing

  paniced



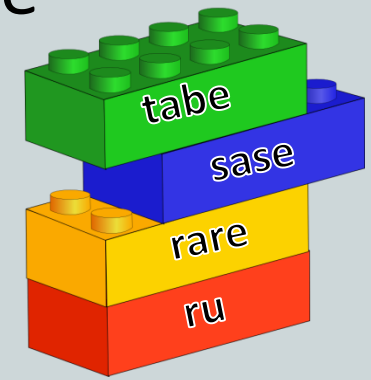
Home work – Japanese verbs

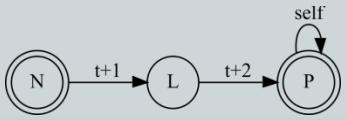
- For next **Wednesday** we will write a .lexc file and a python script for Japanese verb forms
- We will practice on four verbs from the two major conjugation classes:
  - -eru/-iru verbs: *taberu* 'eat', *nobiru* 'stretch'
  - -u verbs: *yomu* 'read', *hanasu* 'speak'



Home work – Japanese verbs

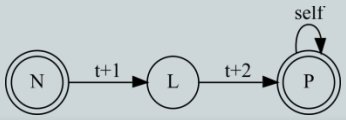
- We will model the causative and passive inflections:
  - -iru/-eru verbs:
    - Drop 'ru'
    - Add **saseru** (causative) or **rareru** (passive)
    - or both: **saserareru** (be made to do something)
    - ***tabesaseru***: make someone eat; ***nobirareru***: be stretched
  - -u verbs:
    - Drop 'u'
    - Add **aseru** (causative) or **areru** (passive)
    - or both: **aserareru**
    - ***yomaserareru***: *be made to read*





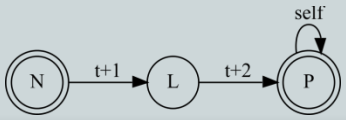
Home work – Japanese verbs

- Produce a .lexc file that:
  - Defines the verb stems in each class (you will need two paths of verbal endings)
  - Defines the necessary suffixes (which differ in each class)
  - Combines the suffixes correctly with each verb type
- In a separate **python** script, use the fst's **transduce** command to get analyses for the provided Japanese words file (submit both **.lexc** and **.py** files!)
- You should get all 1+3 possible inflected forms for all 4 verbs (16 forms):
  - ***taberu, tabesaseru, taberareru, tabesaserareru*** (be made to eat)
  - …



Home work – Japanese verbs

- Bonus: [1pt each, total 2pts]
  - Add a gloss to each word in the .lexc file so your analysis also outputs a **translation:**
    - yomaseru -> read+V+Caus
  - Add the honorific suffix -masu to the base form according to this list, and the test forms to the .txt file:
    - Type 1: (use a symbol +Hon)
      - yomimasu
      - hanashimasu <- note: **si** is pronounced **shi** in Japanese - use re.sub!
    - Type 2: (use a symbol +Hon)
      - tabemasu
      - nobimasu



FSM: Going further

- More on XFST syntax: in **Canvas**
- XTAG English Morphology
  - Upenn project for a large coverage English grammar (in TAG, backed by FSM)
  - http://www.cis.upenn.edu/~xtag/swrelease.html
- EMOR (and SMOR for German):
  - http://www.cis.uni-muenchen.de/~schmid/tools/SFST/
- PCKIMMO – English FSM (and Japanese, Finnish)
  - http://www.sil.org/pckimmo
- morpha/morphg – English grammar
  - http://users.sussex.ac.uk/~johnca/morph.html
  - Version ported to Java: https://github.com/knowitall/morpha