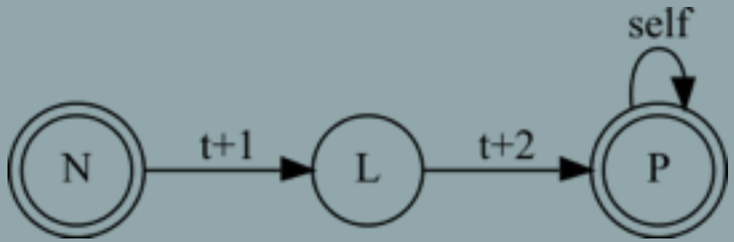
Algorithms for NLP

Finite State Morphology

Amir Zeldes
amir.zeldes@georgetown.edu

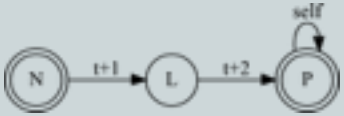
21 September 2017



Regular Expressions – a language?

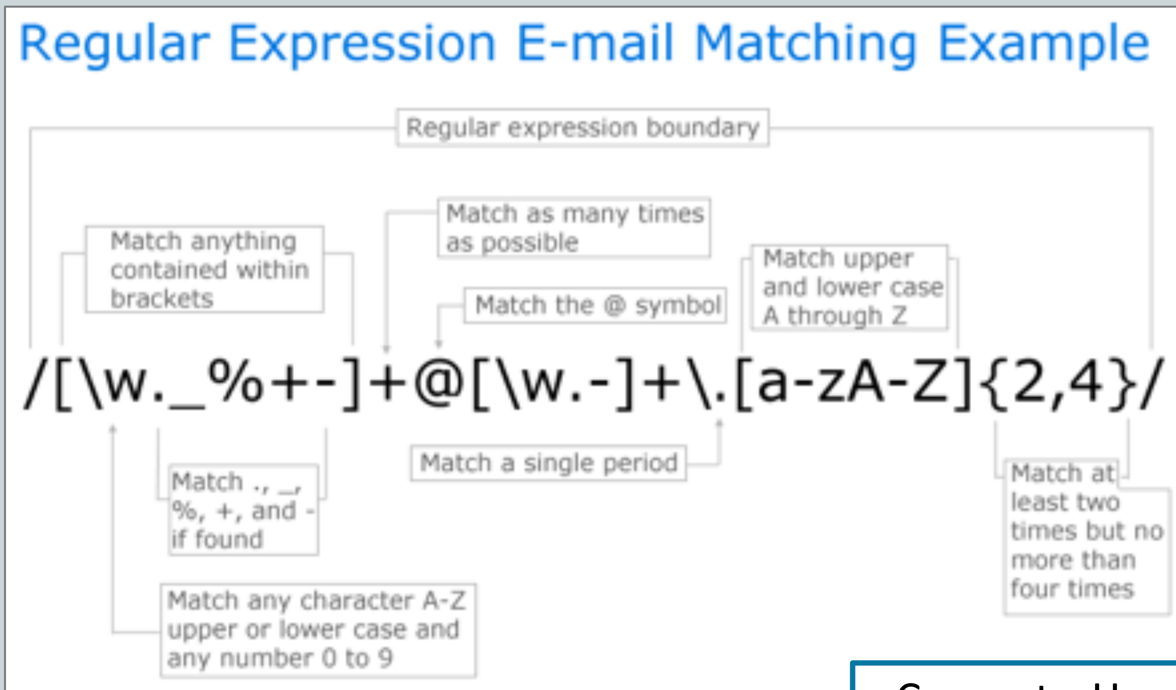
- By now we've all seen regular expressions
- Very useful for finding phone numbers, URLs
- Or potentially catching killers:

XKCD

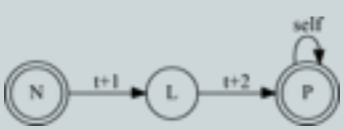


Regular Expressions – a language?

- Can regex capture the grammar of a language?
- What is the grammatical structure of e-mail?



ComputerHope



From regex to natural language

- Regular expressions describe a simple grammar
  - For example, you could think of a regex:
    /DA*N/
  - As describing a Noun Phrase:

Determiner	(Adj)*	Noun
The	*quick brown*	*fox*
a		*house*
my	*lovely*	*cat*

  - Just replace each noun with N, each Adj with A…
  - We can now recognize noun phrases!
    (why should we?)

From regex to natural language

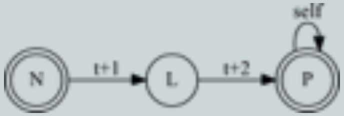
- In fact, syntax is more complex than what we can express with regex:

pick the kids up: /VDNP/

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

sleep the kids up

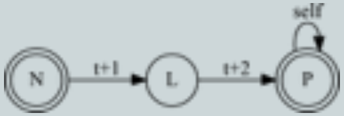
pick the kids over



From regex to natural language

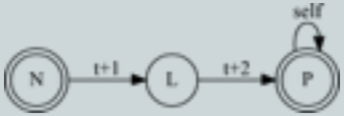
- But for **morphology**, word formation is often describable using something like regex:
  - Super anti adverbs: /(super)?(anti)?ADJ(ly)?/
    super-anti-ingenious-ly
  - Noun compounds: /N+N/
    nightgown
- But what is ADJ? or N?
- Can we do regex with a different 'alphabet'?

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



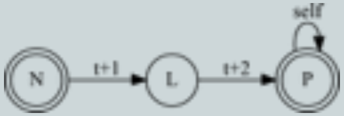
Regular languages

- In fact we can create a regular language grammar using:
  - 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



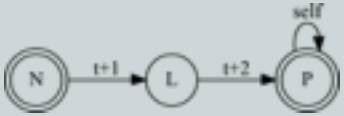
Finite State Methods

- Another way to look at regular grammars is thinking of a reader head
- Moving from character to character on a ribbon
- /ba+$/ matches like this:
  - Initial state: read till you see a **b**
  - The letter **b** is reached -> change to state 2
  - Move right on the ribbon – look for **a**
    - If **a** is seen -> stay in the same state, keep going
    - Else if non-**a** is seen -> match fails
    - Else if input runs out -> done (successful match)



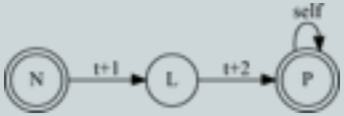
Finite state automaton - FSA

- This type of computing characterizes an **FSA:**
  - Finite number of states, including start and end
  - Transitions depend on input
- More formally:
  - 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



Finite State Morphology

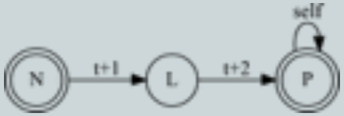
- Among most successful applications of FSA
- Popular for agglutinative languages (Turkish, Japanese), and highly inflected concatenative ones (e.g. Slavic)
- Some approaches to non-concatenative morphologies (Arabic, Hebrew)
- Basic tasks:
  - Morphological parsing
  - Generation



From NL input to states

- Famous Turkish example (Jurafsky & Martin 2008, after Kemal Oflazer):
  - UygarlaştIramadIklarImIzdanmIşsInizcasIna
    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…



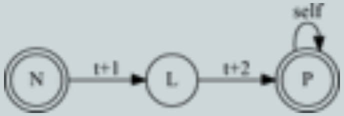
Morphological parsing

- The task:
  - Given some word in language X as input:
  - Output lexicon forms of constituent parts (“morphemes”)
  - 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

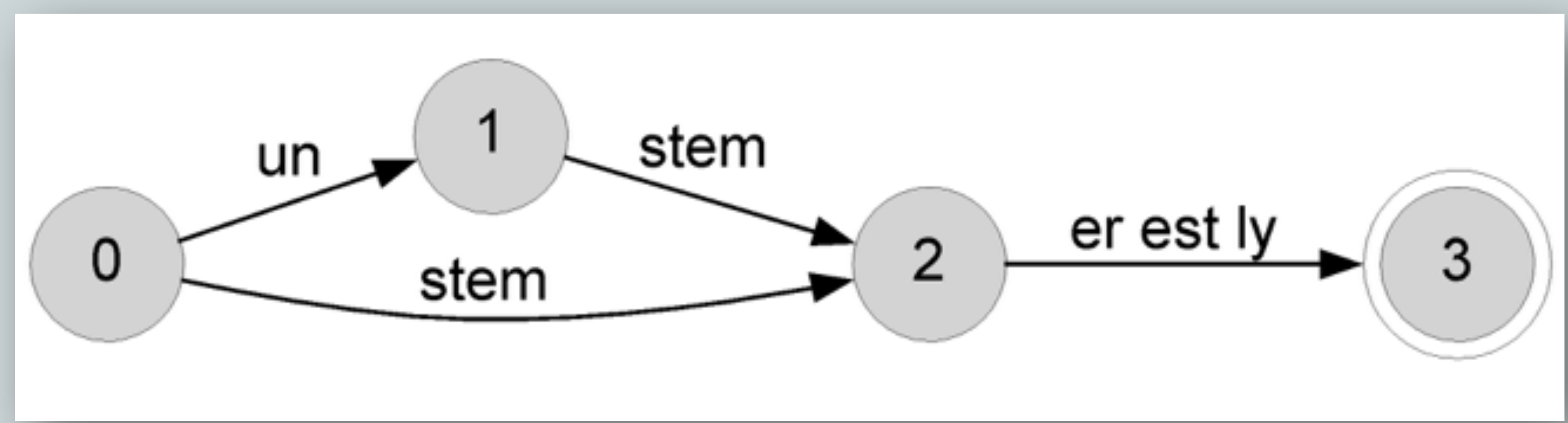
English adjectives

- 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?

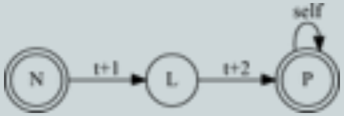


First approximation

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



- ➢Problems?

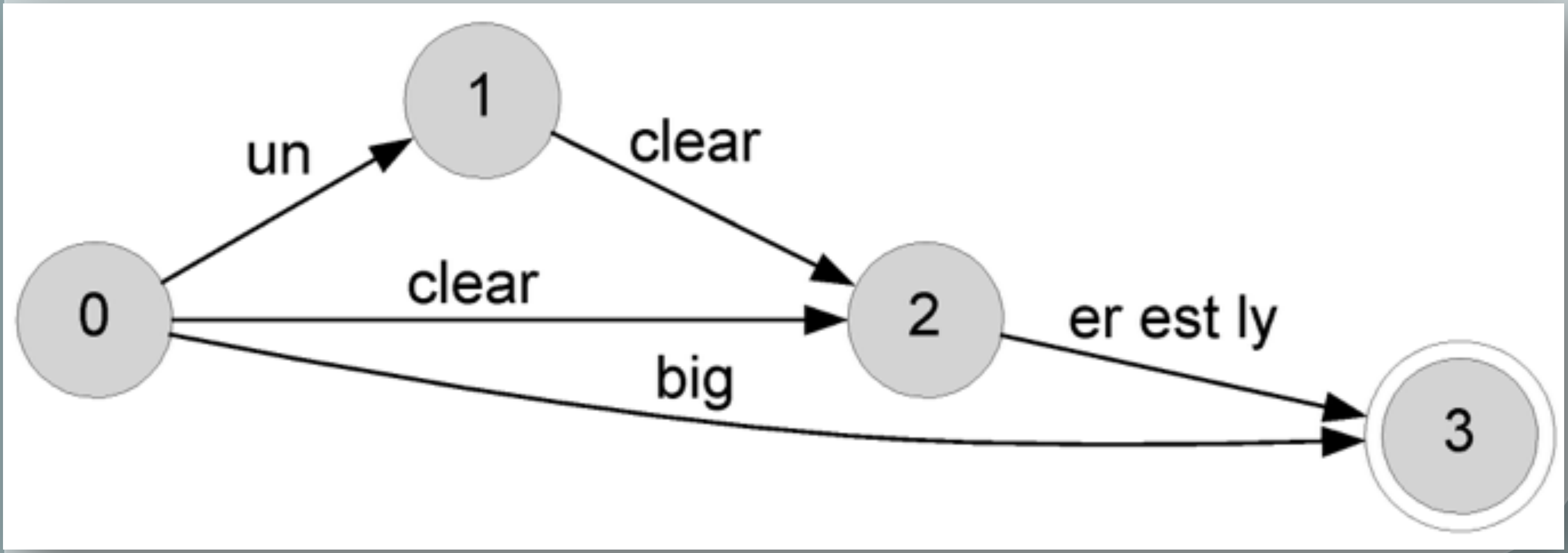


Problems

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

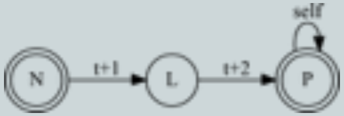
Solutions

- 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 (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 work with **Foma** today (platform independent)



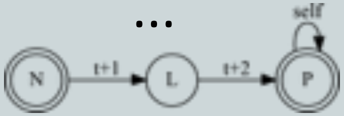
Running Foma

- You should have Foma for your OS from here:
  - https://code.google.com/p/foma/
- To run it, open a terminal window
  - Windows: Window key -> cmd
  - Mac: run Terminal
  - Navigate to directory: cd YOUR_PATH
  - Run foma: foma (or foma.exe)

Running Foma

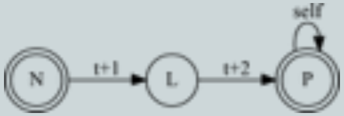
- Interactive mode is like an interpreter (e.g. in Python):

  foma[0]: **define** ***Consonant*** [p|t|k|b|d|g|l|m|n|r|s|z];
  defined Consonant: 543 bytes. 2 states, 12 arcs, 12 paths.
  foma[0]: **define** ***Vowel*** [a|e|i|o|u];
  defined Vowel: 333 bytes. 2 states, 5 arcs, 5 paths.
  foma[0]: **regex** ***Consonant Vowel+***;
  757 bytes. 3 states, 22 arcs, Cyclic.
  foma[1]: **words** # Outputs possible inputs which terminate
  pa
  paa
  pae
  pe
  pea
  ...



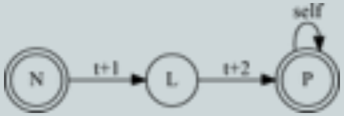
Some regex differences to POSIX

- The regex syntax used in Python is called POSIX
- XFST / Foma syntax has some differences:
  - Use spaces to delimit symbols:
    - cat = a single symbol, called cat
    - c a t = three symbols, called c, a, t
    - {cat} = alternative spelling of c, a, t
  - The dot is replace by ?:
    - {c ? t} = cat, cot, cut …
  - Use brackets instead of ? for optional:
    - (UN) ADJ
  - % is the escape symbol (like \): %? = a real question mark
  - \ is negation: \a = not an a
  - The only disjunction (OR) is: [x|y] – no ranges like [abc]



Strings and symbol names

- Definitions are most of the work, look like this:
  - **define** ***NAME*** DEFINITION;
- Definitions can contain string literals in {…} or other names:
  - **define** ***first_name*** {Amir};
  - **define** ***last_name*** {Zeldes};
  - **define** ***full_name*** first_name last_name;



Strings and symbol names

- We can also allow alternative values:

  define ***first*** [{Bobby} | {Amir}];

  define ***last*** {Zeldes};

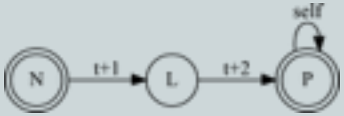
  define ***full*** first last;

  regex ***full***;

  words;

  BobbyZeldes

  AmirZeldes

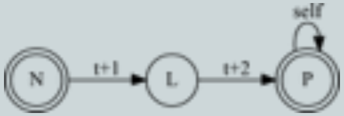


Let's try the English adjectives

- Suppose adjectives look like this:
  - Can start with ***un-***
  - Have a stem like ***big*** or ***clear***
  - Can end in ***-er***, ***-est***
- Use:
  - **define** SYMBOL1 [{string1}|{string2}|…];
  - **regex** (SYMBOL1) SYMBOL2 …;
  - **words**;

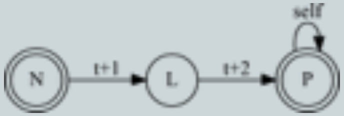
Solution

foma[0]: **define** ***UN*** {un};
defined UN: 212 bytes. 3 states, 2 arcs, 1 path.
foma[0]: **define** ***STEM*** [{big}|{clear}];
defined STEM: 423 bytes. 8 states, 8 arcs, 2 paths.
foma[0]: **define** ***SUFFIX*** [{er}|{est}];
defined SUFFIX: 303 bytes. 4 states, 4 arcs, 2 paths.
foma[0]: **regex** ***(UN) STEM (SUFFIX)***;
607 bytes. 13 states, 16 arcs, 12 paths.
foma[1]: **words**
unbig
unbiger
unbigest
unclear
unclearer
unclearest
…



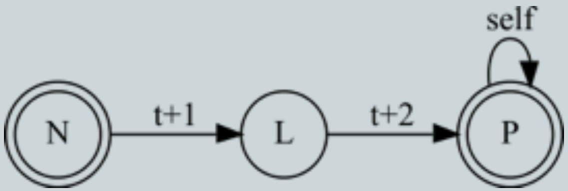
Visualizing automata

- It can be useful to visualize FSAs as graphs
- Generic graph visualization software:
  - http://www.graphviz.org/
  - Defines a format for graphs
  - Graphs are rendered using layouting algorithms
  - Several options come with Graphviz, notably **dot**
- You also need a viewer:
  - A simple cross-platform option: ZGR Viewer
  - http://zvtm.sourceforge.net/zgrviewer.html

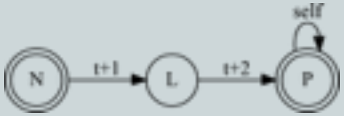


Visualizing automata

- Example: this little NLP logo:

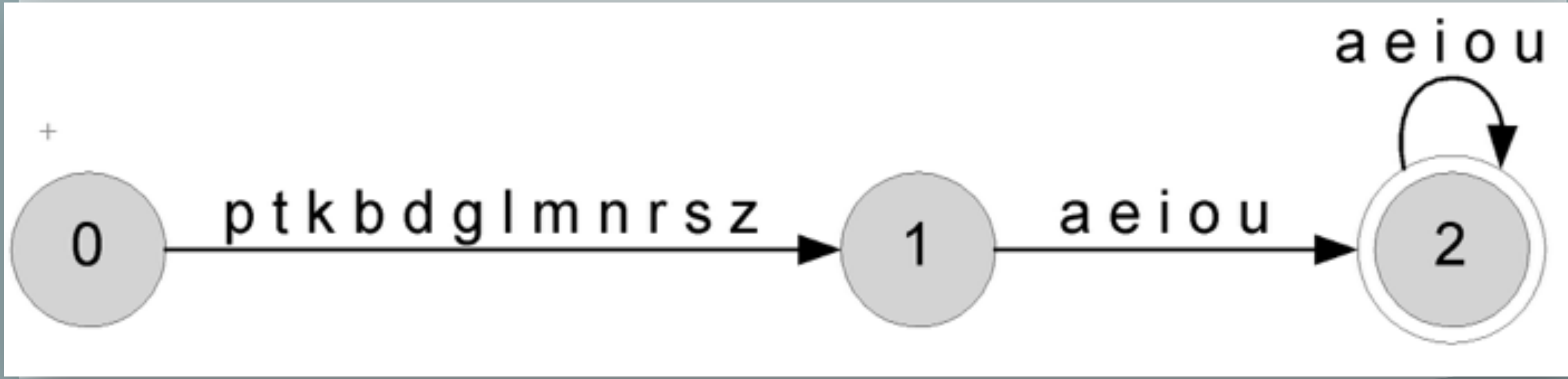



```
digraph finite_state_machine {
  rankdir=LR;
  size="8,5"
  node [shape = doublecircle]; N P;
  node [shape = circle];
  N -> L [ label = "t+1" ];
  L -> P [ label = "t+2" ];
  P-> P [ label = "self" ];
}
```



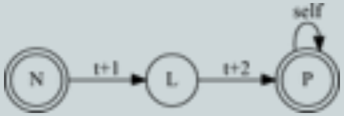
Our syllable example

- Foma [1]: print dot > some_file_name.dot
- Open with a GraphViz viewer (e.g. ZGRViewer)
  - Note the states and transitions
  - Start and end symbols (0 and double circle)



Script files

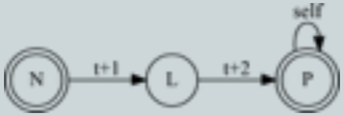
- More often we'll define an automaton in a separate file
- Let's look at a script describing English numerals:
  - Download this script by Lauri Karttunen:
    http://corpling.uis.georgetown.edu/amir/public/numeral.script



An example: English numerals

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

```
# Excerpt from Karttunen (2004)
# List the numbers from one to nine
define OneToNine [{one} | {two} | {three} | {four} | {five}
| {six} | {seven} | {eight} | {nine}];
```



An example: English numerals

- We need to deal with teens + multiples of ten
- What kinds of morphemes are there?
- Schematically:
  - thir + [{teen}|{ty}]
  - four + [{teen}|{ty}]
  - fif + [{teen}|{ty}]
  - …
- Ten, eleven, twelve and twenty are separate

An example: English numerals

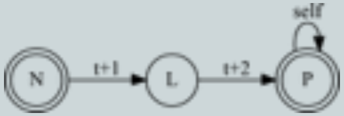
Rules for teens:

define *TeenTen* [{thir} | {fif} | {six} | {seven} | {eigh} | {nine}];

define *Teens* [{ten} | {eleven} | {twelve} |
[*TeenTen* | {four}] {teen}];

Now things that can be followed by -ty:

define *TenStem* [*TeenTen* | {twen} | {for}];



An example: English numerals

Finally, allow twenty one, twenty two…:

define *Tens* [*TenStem* {ty} ({-} *OneToNine*)];

Now all possible numbers are:

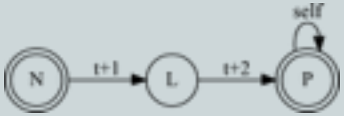
define *OneToNinetyNine* [*OneToNine* | *Teens* | *Tens*];

Push our automaton to the stack for use

regex *OneToNinetyNine;*

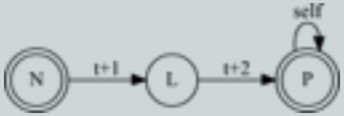
print random-words

exit



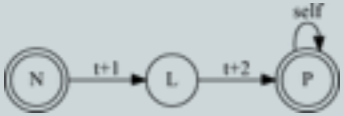
Result

foma[1]: **random-words**
[1] fifty-five
[1] seven
[2] fifty
[1] forty
[1] fifty-nine
[1] eleven
[1] twenty
[2] thirty
[1] ten
[1] nineteen
[1] two
[1] nine
[1] ninety-seven



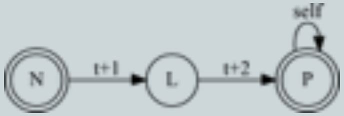
Running the script

- To run a script file from the terminal:
  - `foma -l numeral.script`



What next

- This has been a very shallow introduction: ‘regex with morphemes’
- FSMs can go much further
- Usually:
    - Two way translation between forms and analyses
    - Formally, Finite-State Transducers (FSTs)



What next

- More realistic examples:

  398 bytes. 7 states, 7 arcs, 2 paths.

  foma[1]: **up**

  apply up> cats

  cat+N+Pl

  foma[1]: **down**

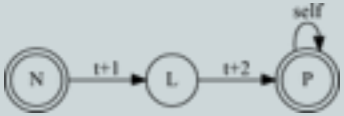
  apply down> cat+N+Pl

  cats

- The basis for generating inflected forms in NLG, analysis in morphologically rich NLU

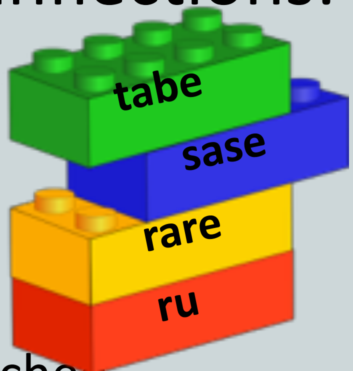
Exercise for home – Japanese verbs

- You can practice writing finite-state morphologies on a language you are less familiar with
- Try modeling four verbs from the two major conjugation classes in Japanese:
  - -eru/-iru verbs: *taberu* 'eat', *nobiru* 'stretch'
  - -u verbs: *yomu* 'read', *kaku* 'write'



Exercise for home – Japanese verbs

- We can 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*

Exercise for home – Japanese verbs

- Produce a script that:
  - Defines the verb stems in each class
  - Defines the necessary suffixes
  - Combines the suffixes correctly with each class
- Using the **words** command, you should get all 3 possible inflected forms for all 4 verbs (12 forms):
  - *tabesaseru, taberareu, tabesaserareru* (be made to eat)
  - …
- If you want to learn more, send me an e-mail and stop by LING-362 in two weeks!