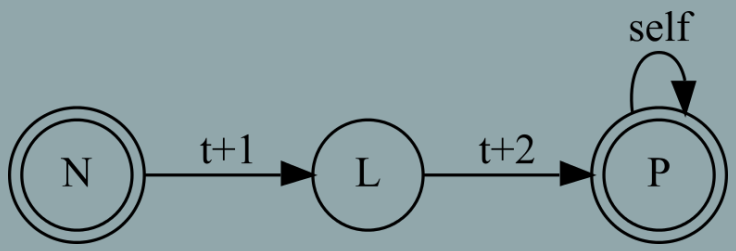
LING-362

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

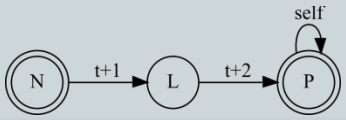
Syntactic parsing II



Exercise: results

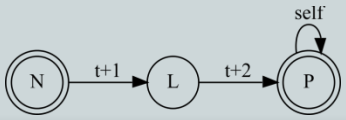
- For 13 sentences we can use:
  - 159 rules (118 unique)
  - 70 lexical + 89 phrasal

  ➢ lexical rules would quickly become much bigger if we continue!



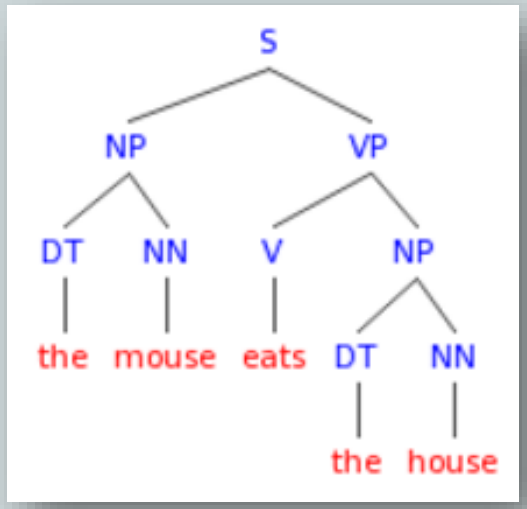
Exercise sentences

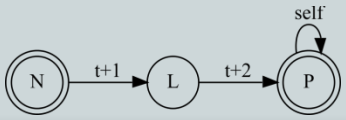
- Top 10 rules:
  - S > NP VP 11 (incl. with ".")
  - . > . 7
  - NP > PRP 7
  - DT > the 6
  - NP > DT NN 5
  - PRP > I 3
  - CC > and 2
  - RB > not 3
  - VB > be 2
  - VP > VBD 2



Can we build a grammar from trees?

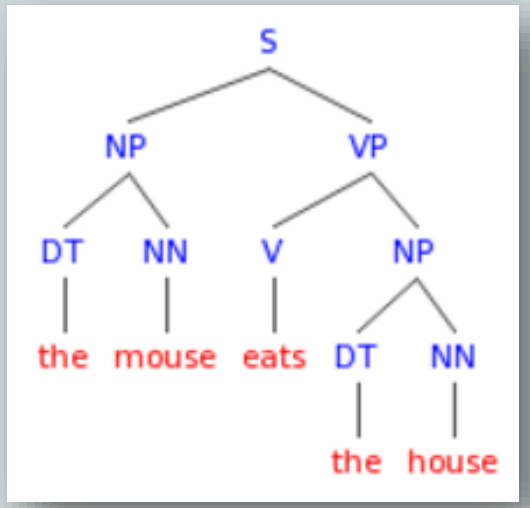
- Grammars can be induced from annotated data just like in our exercise
- In some ways, a corpus of syntax trees – a Treebank – **is** a grammar
- How often do we need the rules inherent in:

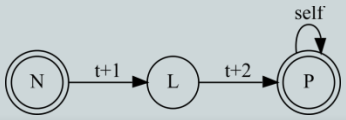




Answer

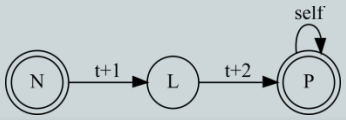
- S > NP VP (once)
- VP > V NP (once)
- NP > DT NN (twice)
- DT > the (twice)
- NN > mouse (once)
- NN > house (once)





Saving probabilities

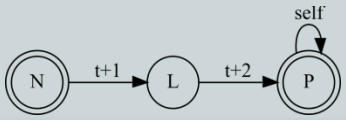
- It's easy to note how often each rule occurred,
  - Saving this data gives us an idea of how likely each decomposition is
  - Maybe we do need a rule for:
    - VP > V PP (overindulge in you)
    - V > over+V (overindulge)
  - But it's very rare/unlikely
- Saving the probabilities gives us a **Probabilistic Context Free Grammar** (PCFG)



How many rules?

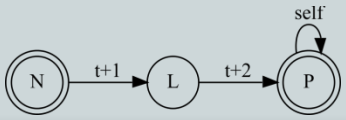
- Would we get a lot more rules than we could come up with by hand?
- The Penn Treebank Wall Street Journal corpus contains about 1 million tokens
- How many distinct **non-lexical** transition rules does it contain (incl. POS)?

- ~17,500 (Jurafsky & Martin 2008:408)



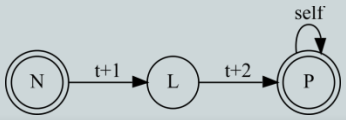
Very nice, but…

- So far we can only **generate**
  - Make all possible utterances using rules from a grammar or treebank
  - We could build a ‘tree-chatbot’ ☺
- NLP often less interested in generating
  - Random sentences are nice
  - But we want to **process** actual sentences generated by humans
  - Gateway to Natural Language **Understanding** (NLU)



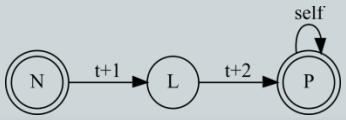
How can we recognize a parse?

- Is this an English sentence?
  - ***The cat the dog the mouse licked bit ran***
- Two ways to check:
  - **Top down**, we generate all possible sentences:
    - S > NP S VP (twice)
    - S > NP VP (once)
    - …
    - V > ran



How can we recognize a parse?

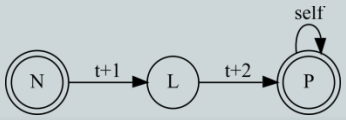
- Is this an English sentence?
  - ***The cat the dog the mouse licked bit ran***
- Two ways to check:
  - **Bottom up**, we try to build phrases from words:
    - DT > The : means we might have a DT here
    - …
    - NP > DT NN : means we might have an NP
    - …
    - S > NP VP : OK, we've reached start symbol, all good!



Efficiency

- In principle it's possible to use either approach
- In practice, it quickly becomes slow/impossible to apply so many rules so often for each sentence – recursion means infinite sentences!

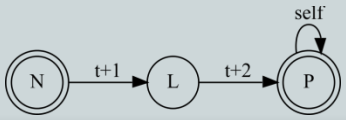
- We'd like to check all possible analyses at each position just once
- Example: The Cocke-Kasami-Younger algorithm (CKY)



Dynamic programming – CKY

- Much like in the Viterbi case, it's possible to 'keep track' of a path over all possible choices
- We conceptualize parses as a table:

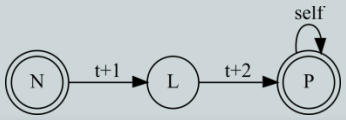
The	boys	shop	daily



Dynamic programming – CKY

- We'll only need the top diagonal half

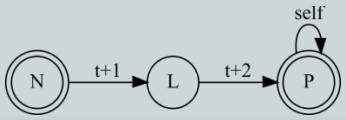
The	boys	shop	daily



Dynamic programming – CKY

- The CKY algorithm keeps track of possible parses by noting if any cell in the table is the right hand side of a rule

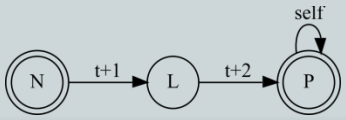
The	boys	shop	daily
DT			



Dynamic programming – CKY

- The CKY algorithm keeps track of possible parses by noting if any cell in the table is the right hand side of a rule

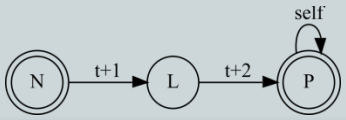
The	boys	shop	daily
DT			
	NNS		



Dynamic programming – CKY

- The CKY algorithm keeps track of possible parses by noting if any cell in the table is the right hand side of a rule

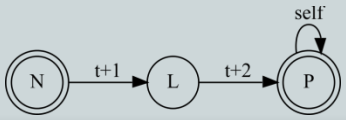
The	boys	shop	daily
DT	NP		
	NNS		



Dynamic programming – CKY

- The CKY algorithm keeps track of possible parses by noting if any cell in the table is the right hand side of a rule

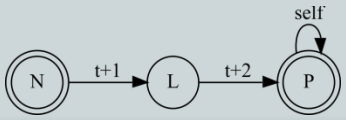
The	boys	shop	daily
DT	NP		
	NNS		
		VBP | NN	



Dynamic programming – CKY

- The CKY algorithm keeps track of possible parses by noting if any cell in the table is the right hand side of a rule

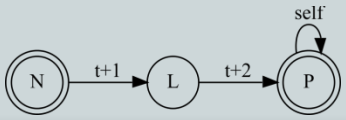
The	boys	shop	daily
DT	NP		
	NNS		
		VBP | NN	VP
			RB



Dynamic programming – CKY

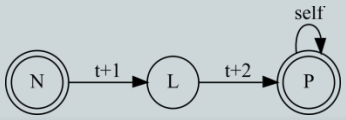
- The CKY algorithm keeps track of possible parses by noting if any cell in the table is the right hand side of a rule

The	boys	shop	daily
DT	NP		S
	NNS		
		VBP	VP
			RB



Some caveats

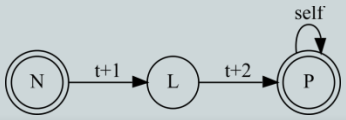
- Note that we only get to look at the previous column at each level
  - No way of checking two previous items
  - **Grammar must be in CNF**
- And we don't really know which parses worked
  - We only know that an S is somewhere in there
  - We would need to **keep track** of derivations belonging together
  - Allow multiple identical entries (several S also possible)



Dynamic programming – CKY

- Indexing multiple solutions

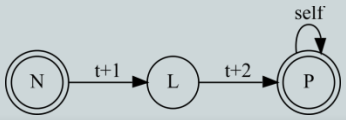
The	boys	shop	daily
DT	NP		S
	NNS	NP	
		VBP | NN	VP
			RB



Dynamic programming – CKY

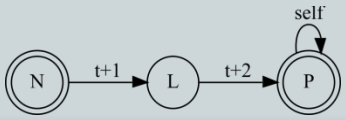
- Indexing multiple solutions

The	boys	shop	daily
DT	NP		S | S
	NNS	NP	
		VBP | NN | VBP	VP | VP
			RB | NN



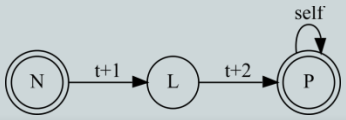
Parsing with NLTK – part 1

- Let's try to parse using the grammar we made together!
  - Example script: ***nltk_parsing.py***
- NLTK includes implementations of a variety of parsing algorithms
  - It is possible to use CKY, but we would need to generate and revert **CNF** style trees (dummy nodes)
  - We want to keep our n-ary trees
  - We will use NLTK's 'chart parser' (a related algorithm)



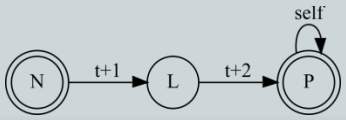
Compiling the grammar

```
import nltk
from nltk import CFG

grammar_string = """
S -> ADJP NP VP
S -> NP VP
S -> NP VP ADVP
S -> NP VP SENT
... """

exercise_grammar = CFG.fromstring(grammar_string)
```



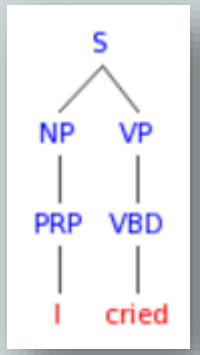
Parsing

```
test_sent = "I cried"

tokenized = nltk.word_tokenize(test_sent)

# Make a parser object, initialized with a grammar argument
parser = nltk.ChartParser(exercise_grammar)
# Parse tokenized data
trees = parser.parse(tokenized)
# Iterate through possible trees
for tree in trees:
    print(tree)
```

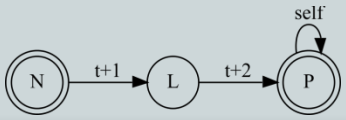


Output

```
(S
      (NP (PRP I))
      (VP (VBD cried))
)
```

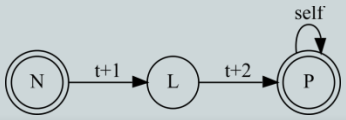


Great, an unambiguous parse!



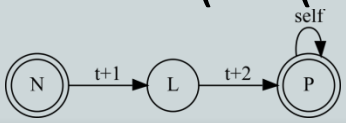
Parsing – ambiguity

```
test_sent = "It should never be the watery pizza"

tokenized = nltk.word_tokenize(test_sent)

parser = nltk.ChartParser(exercise_grammar)
trees = parser.parse(tokenized)

# Iterate through possible trees
for tree in trees:
    print(tree)
```



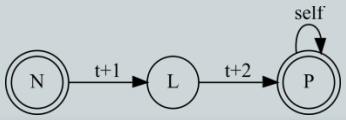
Output

```
(S
  (NP (PRP It))
  (VP
    (MD should)
    (RB never)
    (VB be)
    (NP (DT the) (ADJP (JJ watery)) (NN pizza))))
(S
  (NP (PRP It))
  (VP
    (MD should)
    (RB never)
    (VB be)
    (NP (DT the) (JJ watery) (NN pizza))))
```



Output

```
(S
  (NP (PRP It))
  (VP
    (MD should)
    (VP
      (RB never)
      (VP (VB be) (NP (DT the) (ADJP (JJ watery)) (NN pizza))))))
(S
  (NP (PRP It))
  (VP
    (MD should)
    (VP (RB never) (VP (VB be) (NP (DT the) (JJ watery) (NN
pizza))))))
```



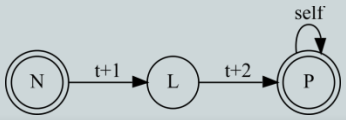
Homework – for Wednesday, Nov. 24

- Add rules so that the following sentences are parsed:
  - ***It is not their home***
  - ***They are very watery***
  - ***I was calling on the phone***

  Transition rules could be missing and words could be missing!

- Prevent the variable VP and ADJ problems so the 'pizza' sentence is unambiguous like this:

```
(S
  (NP (PRP It))
  (VP
    (MD should)
    (VP
      (RB never)
      (VP (VB be) (NP (DT the) (ADJP (JJ watery)) (NN pizza))))))
```

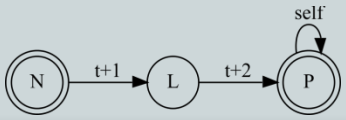


Homework

- How to submit:
  - Edit the script to add your rules directly
  - You can make life easier for us by doing:
    - grammar_string == """
      …
      VB -> "go"
      SENT -> "."
      """

      grammar_string += "S -> SENT“

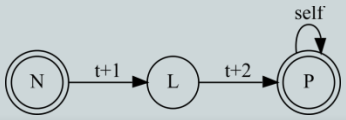
- **Note:** NLTK will not accept “$” in a category symbol, so use PRPS for “their”



Homework

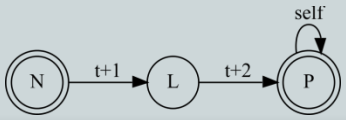
- How to submit:
  - Add test sentences and repeat this code chunk for each sentence you should parse – **test your code**!

```
…
test_sent3 = "I was calling on the phone"

tokenized = nltk.word_tokenize(test_sent3)

trees = parser.parse(tokenized)

for tree in trees:
    print(tree)
```



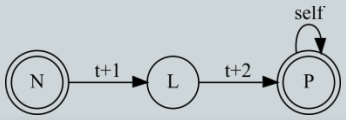
Parsing from scratch

- NLTK's rule based parser is nice
- But:
  - We want to be able to **learn** grammars from corpora
  - We want to know **how** this works!
- Let's take a look at the CKY algorithm!
  - Download and debug *code/parsing/cky_simple.py*



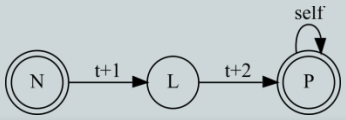
CKY algorithm

```
token_table = []
# Make an empty table
for i in range(len(tokens)):
    # Add a row
    token_table.append(["0"])
    for j in range(len(tokens)):
        # Add columns
        token_table[i].append("--")

for i in range(len(tokens)):
    token_table[i][i+1] = tagged[i][1]
```



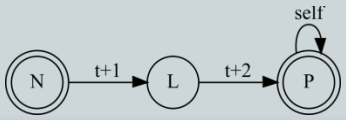
CKY algorithm

```
text = "the cat ate the mouse"
tokens = word_tokenize(text)
tagged = pos_tag(tokens)

rev_grammar = {}
rev_grammar[("DT","NN")] = "NP"
rev_grammar[("VB","NP")] = "VP"
rev_grammar[("NP","VP")] = "S"

full_table = cky(token_table, tokens, rev_grammar)

for row in full_table:
    print(row)
```



CKY algorithm

```
def cky(table, tokens, rev_grammar):
    for span in range(2, len(tokens)+1):
        for start in range(len(tokens)+1-span): # Go down diagonal
            end = start + span
            for mid in range(start+1, end):
                node1, node2 = table[start][mid], table[mid][end]
                if (node1, node2) in rev_grammar:
                    table[start][end] = rev_grammar[(node1, node2)]
    return table
```



Choosing the right parse

- All an algorithm like CKY does for us so far is check for **possible** positions to **split into phrases**
  - Useful in many contexts (CKY can be applied to Chinese word segmentation! Qian & Liu 2012)
  - Genome mapping (see Poptsova 2014)
  - …
- How can we decide which parse is right?