Intro to the NLTK syntax and semantics resources

Adrian Brasoveanu*

March 3, 2014

Contents

*Based on the NLTK book (Bird et al. 2009) and created with the PythonTeX package (Poore 2013).

1 Lexical Semantics – Wordnet

[**py1**]
```
>>> from __future__ import division
>>> import nltk, re, pprint
```

WordNet is a semantically-oriented dictionary of English, similar to a traditional thesaurus but with a richer structure.

- 155,287 words and 117,659 synonym sets

Consider (1) and (2) – they differ only with respect to the fact that the word *motorcar* in (1) is replaced by *automobile* in (2).

(1) Benz is credited with the invention of the motorcar.

(2) Benz is credited with the invention of the automobile.

The meaning of the sentences is pretty much the same. Given that everything else in the sentences is the same, we can conclude that the words 'motorcar' and 'automobile' have the same meaning, i.e., they are *synonyms*.

We can explore these words with the help of WordNet.

[**py2**]
```
>>> from nltk.corpus import wordnet as wn
>>> wn.synsets('motorcar')
[Synset('car.n.01')]
```

- *motorcar* has just one possible meaning and it is identified as `car.n.01` – the first noun sense of *car*

The entity `car.n.01` is called a synset, or "synonym set": a collection of synonymous words (or lemmas).

[**py3**]
```
>>> wn.synset('car.n.01').lemma_names
['car', 'auto', 'automobile', 'machine', 'motorcar']
```

Each word of a synset can have several meanings, e.g., *car* can also signify a train carriage, a gondola or an elevator car.

[**py4**]
```
>>> for synset in wn.synsets('car'):
...     print synset.lemma_names
...
['car', 'auto', 'automobile', 'machine', 'motorcar']
['car', 'railcar', 'railway_car', 'railroad_car']
['car', 'gondola']
['car', 'elevator_car']
['cable_car', 'car']
>>> for synset in wn.synsets('machine'):
...     print synset.lemma_names
...
['machine']
```

```
['machine']
['machine']
['machine', 'simple_machine']
['machine', 'political_machine']
['car', 'auto', 'automobile', 'machine', 'motorcar']
['machine']
['machine']
```

But we are only interested in the single meaning that is common to all words of the synset `car.n.01`.

```
[py5] >>> wn.synset('car.n.01').lemma_names
      ['car', 'auto', 'automobile', 'machine', 'motorcar']
```

Synsets also come with a prose definition and some example sentences:

```
[py6] >>> wn.synset('car.n.01').definition
      'a motor vehicle with four wheels; usually propelled by an internal combustion engine'
      >>> wn.synset('car.n.01').examples
      ['he needs a car to get to work']
```

Although definitions help humans to understand the intended meaning of a synset, the words of the synset are often more useful for programs. That is, we identify the various meanings of a single lexical item by the items it is synonymous with.

- *motorcar* has only one synset / meaning

```
[py7] >>> for synset in wn.synsets('motorcar'):
      ...     print synset.lemma_names
      ...
      ['car', 'auto', 'automobile', 'machine', 'motorcar']
```

- *auto* has only one synset / meaning – the same as *motorcar*

```
[py8] >>> for synset in wn.synsets('auto'):
      ...     print synset.lemma_names
      ...
      ['car', 'auto', 'automobile', 'machine', 'motorcar']
```

- *car* has 5 synsets / meanings

```
[py9] >>> for synset in wn.synsets('car'):
      ...     print synset.lemma_names, '\n'
      ...
      ['car', 'auto', 'automobile', 'machine', 'motorcar']

      ['car', 'railcar', 'railway_car', 'railroad_car']

      ['car', 'gondola']
```

```
['car', 'elevator_car']

['cable_car', 'car']
```

- *machine* has 8 synsets / meanings

[**py10**]
```
>>> for synset in wn.synsets('machine'):
...     print synset.lemma_names, '\n'
...
['machine']

['machine']

['machine']

['machine', 'simple_machine']

['machine', 'political_machine']

['car', 'auto', 'automobile', 'machine', 'motorcar']

['machine']

['machine']
```

[**py11**]
```
>>> for synset in wn.synsets('machine'):
...     print synset.lemma_names, synset.definition, '\n'
...
['machine'] any mechanical or electrical device that transmits or modifies energy to p

['machine'] an efficient person

['machine'] an intricate organization that accomplishes its goals efficiently

['machine', 'simple_machine'] a device for overcoming resistance at one point by apply

['machine', 'political_machine'] a group that controls the activities of a political p

['car', 'auto', 'automobile', 'machine', 'motorcar'] a motor vehicle with four wheels;

['machine'] turn, shape, mold, or otherwise finish by machinery

['machine'] make by machinery
```

More examples: *dish*, *murder*, *newspaper*

```
[py12] >>> for synset in wn.synsets('dish'):
       ...     print synset.lemma_names
       ...
       ['dish']
       ['dish']
       ['dish', 'dishful']
       ['smasher', 'stunner', 'knockout', 'beauty', 'ravisher', 'sweetheart', 'peach', 'lulu'
       ['dish', 'dish_aerial', 'dish_antenna', 'saucer']
       ['cup_of_tea', 'bag', 'dish']
       ['serve', 'serve_up', 'dish_out', 'dish_up', 'dish']
       ['dish']
       >>> for synset in wn.synsets('dish'):
       ...     print synset.lemma_names, synset.definition, "\n", synset.examples, "\n"
       ...
       ['dish'] a piece of dishware normally used as a container for holding or serving food
       ['we gave them a set of dishes for a wedding present']

       ['dish'] a particular item of prepared food
       ['she prepared a special dish for dinner']

       ['dish', 'dishful'] the quantity that a dish will hold
       ['they served me a dish of rice']

       ['smasher', 'stunner', 'knockout', 'beauty', 'ravisher', 'sweetheart', 'peach', 'lulu'
       []

       ['dish', 'dish_aerial', 'dish_antenna', 'saucer'] directional antenna consisting of a
       []

       ['cup_of_tea', 'bag', 'dish'] an activity that you like or at which you are superior
       ['chemistry is not my cup of tea', 'his bag now is learning to play golf', 'marriage w

       ['serve', 'serve_up', 'dish_out', 'dish_up', 'dish'] provide (usually but not necessar
       ['We serve meals for the homeless', 'She dished out the soup at 8 P.M.', 'The entertai

       ['dish'] make concave; shape like a dish
       []

       >>> for synset in wn.synsets('murder'):
       ...     print synset.lemma_names, synset.definition, "\n", synset.examples, "\n"
       ...
       ['murder', 'slaying', 'execution'] unlawful premeditated killing of a human being by a
       []

       ['murder', 'slay', 'hit', 'dispatch', 'bump_off', 'off', 'polish_off', 'remove'] kill
       ['The mafia boss ordered his enemies murdered']
```

```
['mangle', 'mutilate', 'murder'] alter so as to make unrecognizable
['The tourists murdered the French language']

>>> for synset in wn.synsets('politics'):
...     print synset.lemma_names, synset.definition, "\n", synset.examples, "\n"
...
['politics', 'political_relation'] social relations involving intrigue to gain authori
['office politics is often counterproductive']

['politics', 'political_science', 'government'] the study of government of states and
[]

['politics'] the profession devoted to governing and to political affairs
[]

['politics', 'political_sympathies'] the opinion you hold with respect to political qu
[]

['politics'] the activities and affairs involved in managing a state or a government
['unemployment dominated the politics of the inter-war years', 'government agencies mu

>>> for synset in wn.synsets('up'):
...     print synset.lemma_names, synset.definition, "\n", synset.examples, "\n"
...
['up'] raise
['up the ante']

['up'] being or moving higher in position or greater in some value; being above a form
['the anchor is up', 'the sun is up', 'he lay face up', 'he is up by a pawn', 'the mar

['astir', 'up'] out of bed
['are they astir yet?', 'up by seven each morning']

['improving', 'up'] getting higher or more vigorous
['its an up market', 'an improving economy']

['up', 'upward'] extending or moving toward a higher place
['the up staircase', 'a general upward movement of fish']

['up'] (usually followed by `on' or `for') in readiness
['he was up on his homework', 'had to be up for the game']

['up'] open
['the windows are up']

['up'] (used of computers) operating properly
['how soon will the computers be up?']
```

```
['up'] used up
['time is up']

['up', 'upwards', 'upward', 'upwardly'] spatially or metaphorically from a lower to a
['look up!', 'the music surged up', 'the fragments flew upwards', 'prices soared upwar

['up'] to a higher intensity
['he turned up the volume']

['up'] nearer to the speaker
['he walked up and grabbed my lapels']

['up'] to a more central or a more northerly place
['was transferred up to headquarters', 'up to Canada for a vacation']

['up', 'upwards', 'upward'] to a later time
['they moved the meeting date up', 'from childhood upward']

>>> for synset in wn.synsets('newspaper'):
...     print synset.lemma_names, synset.definition, "\n", synset.examples, "\n"
...
['newspaper', 'paper'] a daily or weekly publication on folded sheets; contains news a
['he read his newspaper at breakfast']

['newspaper', 'paper', 'newspaper_publisher'] a business firm that publishes newspaper
['Murdoch owns many newspapers']

['newspaper', 'paper'] the physical object that is the product of a newspaper publishe
['when it began to rain he covered his head with a newspaper']

['newspaper', 'newsprint'] cheap paper made from wood pulp and used for printing newsp
['they used bales of newspaper every day']

>>> len(wn.synsets('set'))
45
>>> for synset in wn.synsets('set'):
...     print synset.lemma_names, synset.definition, "\n", synset.examples, "\n"
...
['set'] a group of things of the same kind that belong together and are so used
['a set of books', 'a set of golf clubs', 'a set of teeth']

['set'] (mathematics) an abstract collection of numbers or symbols
['the set of prime numbers is infinite']

['set', 'exercise_set'] several exercises intended to be done in series
['he did four sets of the incline bench press']
```

['stage_set', 'set'] representation consisting of the scenery and other properties use
['the sets were meticulously authentic']

['set', 'circle', 'band', 'lot'] an unofficial association of people or groups
['the smart set goes there', 'they were an angry lot']

['bent', 'set'] a relatively permanent inclination to react in a particular way
['the set of his mind was obvious']

['set'] the act of putting something in position
['he gave a final set to his hat']

['set'] a unit of play in tennis or squash
['they played two sets of tennis after dinner']

['hardening', 'solidifying', 'solidification', 'set', 'curing'] the process of becomin
['the hardening of concrete', 'he tested the set of the glue']

['Set', 'Seth'] evil Egyptian god with the head of a beast that has high square ears a
[]

['set'] the descent of a heavenly body below the horizon
['before the set of sun']

['set', 'readiness'] (psychology) being temporarily ready to respond in a particular w
["the subjects' set led them to solve problems the familiar way and to overlook the si

['set'] any electronic equipment that receives or transmits radio or tv signals
['the early sets ran on storage batteries']

['put', 'set', 'place', 'pose', 'position', 'lay'] put into a certain place or abstrac
['Put your things here', 'Set the tray down', 'Set the dogs on the scent of the missin

['determine', 'set'] fix conclusively or authoritatively
['set the rules']

['specify', 'set', 'determine', 'define', 'fix', 'limit'] decide upon or fix definitel
['fix the variables', 'specify the parameters']

['set', 'mark'] establish as the highest level or best performance
['set a record']

['set'] put into a certain state; cause to be in a certain state
['set the house afire']

['set'] fix in a border

['The goldsmith set the diamond']

['fix', 'prepare', 'set_up', 'ready', 'gear_up', 'set'] make ready or suitable or equi
['Get the children ready for school!', 'prepare for war', 'I was fixing to leave town

['set'] set to a certain position or cause to operate correctly
['set clocks or instruments']

['set', 'localize', 'localise', 'place'] locate
['The film is set in Africa']

['set', 'go_down', 'go_under'] disappear beyond the horizon
['the sun sets early these days']

['arrange', 'set'] adapt for performance in a different way
['set this poem to music']

['plant', 'set'] put or set (seeds, seedlings, or plants) into the ground
["Let's plant flowers in the garden"]

['set'] apply or start
['set fire to a building']

['jell', 'set', 'congeal'] become gelatinous
['the liquid jelled after we added the enzyme']

['typeset', 'set'] set in type
['My book will be typeset nicely', 'set these words in italics']

['set'] put into a position that will restore a normal state
['set a broken bone']

['set', 'countersink'] insert (a nail or screw below the surface, as into a countersin
[]

['set'] give a fine, sharp edge to a knife or razor
[]

['sic', 'set'] urge to attack someone
['The owner sicked his dogs on the intruders', 'the shaman sics sorcerers on the evil

['place', 'put', 'set'] estimate
['We put the time of arrival at 8 P.M.']

['rig', 'set', 'set_up'] equip with sails or masts
['rig a ship']

```
['set_up', 'lay_out', 'set'] get ready for a particular purpose or event
['set up an experiment', 'set the table', 'lay out the tools for the surgery']

['adjust', 'set', 'correct'] alter or regulate so as to achieve accuracy or conform to
['Adjust the clock, please', 'correct the alignment of the front wheels']

['fructify', 'set'] bear fruit
['the apple trees fructify']

['dress', 'arrange', 'set', 'do', 'coif', 'coiffe', 'coiffure'] arrange attractively
['dress my hair for the wedding']

['fit', 'primed', 'set'] (usually followed by `to' or `for') on the point of or strong
['in no fit state to continue', 'fit to drop', 'laughing fit to burst', 'she was fit t

['fixed', 'set', 'rigid'] fixed and unmoving
['with eyes set in a fixed glassy stare', 'his bearded face already has a set hollow l

['located', 'placed', 'set', 'situated'] situated in a particular spot or position
['valuable centrally located urban land', 'strategically placed artillery', 'a house s

['laid', 'set'] set down according to a plan:"a carefully laid table with places set f
['stones laid in a pattern']

['set'] being below the horizon
['the moon is set']

['determined', 'dictated', 'set'] determined or decided upon as by an authority
['date and place are already determined', 'the dictated terms of surrender', 'the time

['hardened', 'set'] converted to solid form (as concrete)
[]
```

WordNet synsets correspond to abstract concepts, and they don't always have corresponding words in English.

These concepts are linked together in a hierarchy. Some concepts are very general, such as *Entity*, *State*, *Event* – called unique beginners or root synsets. Others, such as *gas guzzler* and *hatchback*, are much more specific.

For example, given a concept like *motorcar*, we can look at the concepts that are more specific – the (immediate) hyponyms:

[**py13**]
```
>>> motorcar = wn.synset('car.n.01')
>>> motorcar.hyponyms()
[Synset('stanley_steamer.n.01'), Synset('hardtop.n.01'), Synset('loaner.n.02'), Synset
```

Or at the concepts that are less specific – the (immediate) hypernyms:

[**py14**]
```
>>> motorcar.hypernyms()
[Synset('motor_vehicle.n.01')]
```

. . . and further hypernyms:

[**py15**]
```
>>> wn.synset('motor_vehicle.n.01').hypernyms()
[Synset('self-propelled_vehicle.n.01')]
>>> wn.synset('self-propelled_vehicle.n.01').hypernyms()
[Synset('wheeled_vehicle.n.01')]
>>> wn.synset('wheeled_vehicle.n.01').hypernyms()
[Synset('container.n.01'), Synset('vehicle.n.01')]
```

This better explored by tracking the full path or paths from a synset to a root hypernym:

[**py16**]
```
>>> motorcar.root_hypernyms()
[Synset('entity.n.01')]

>>> paths = motorcar.hypernym_paths()
>>> len(paths)
2
>>> [synset.name for synset in paths[0]]
['entity.n.01', 'physical_entity.n.01', 'object.n.01', 'whole.n.02', 'artifact.n.01',
>>> [synset.name for synset in paths[1]]
['entity.n.01', 'physical_entity.n.01', 'object.n.01', 'whole.n.02', 'artifact.n.01',

>>> def simple_path(path):
...     return [s.lemmas[0].name for s in path]
...
>>> for path in paths:
...     print simple_path(path)
...
['entity', 'physical_entity', 'object', 'whole', 'artifact', 'instrumentality', 'conta
['entity', 'physical_entity', 'object', 'whole', 'artifact', 'instrumentality', 'conve
```

[**py17**]
```
>>> for synset in wn.synsets('freedom'):
...     print synset.lemma_names, synset.definition, "\n", synset.examples, "\n"
...
['freedom'] the condition of being free; the power to act or speak or think without ex
[]

['exemption', 'freedom'] immunity from an obligation or duty
[]

>>> wn.synsets('freedom')
[Synset('freedom.n.01'), Synset('exemption.n.01')]
>>> freedom = wn.synset('freedom.n.01')
>>> paths = freedom.hypernym_paths()
>>> len(paths)
1
>>> [synset.name for synset in paths[0]]
['entity.n.01', 'abstraction.n.06', 'attribute.n.02', 'state.n.02', 'freedom.n.01']
```

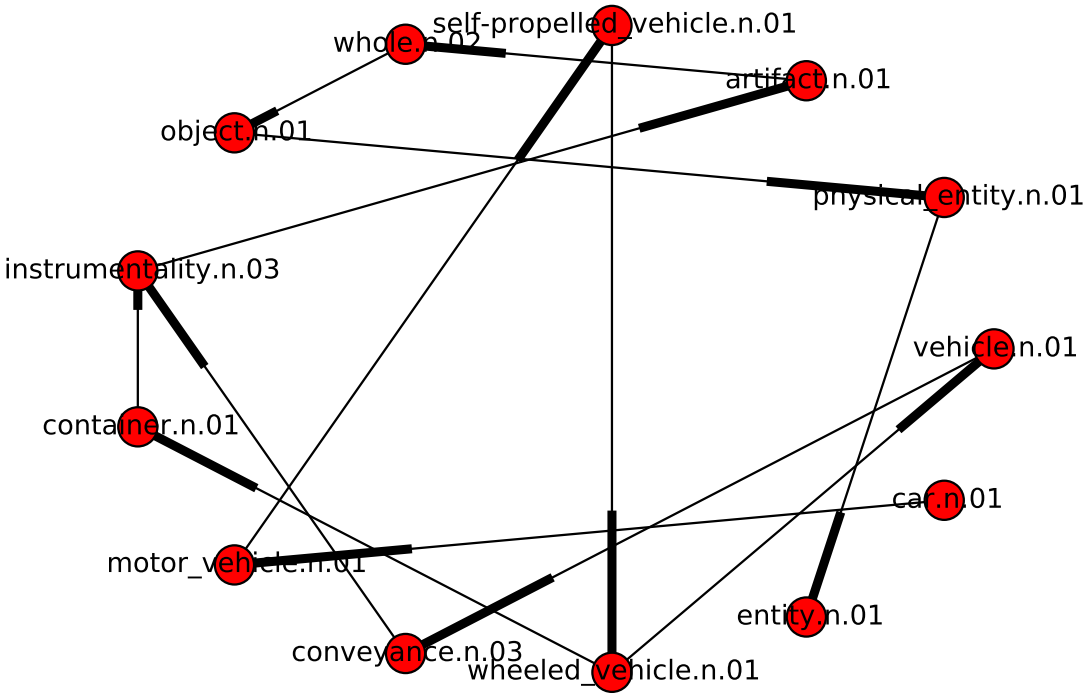
```
>>> for path in paths:
...     print simple_path(path)
...
['entity', 'abstraction', 'attribute', 'state', 'freedom']
```

We can display these graphs as follows.[1]

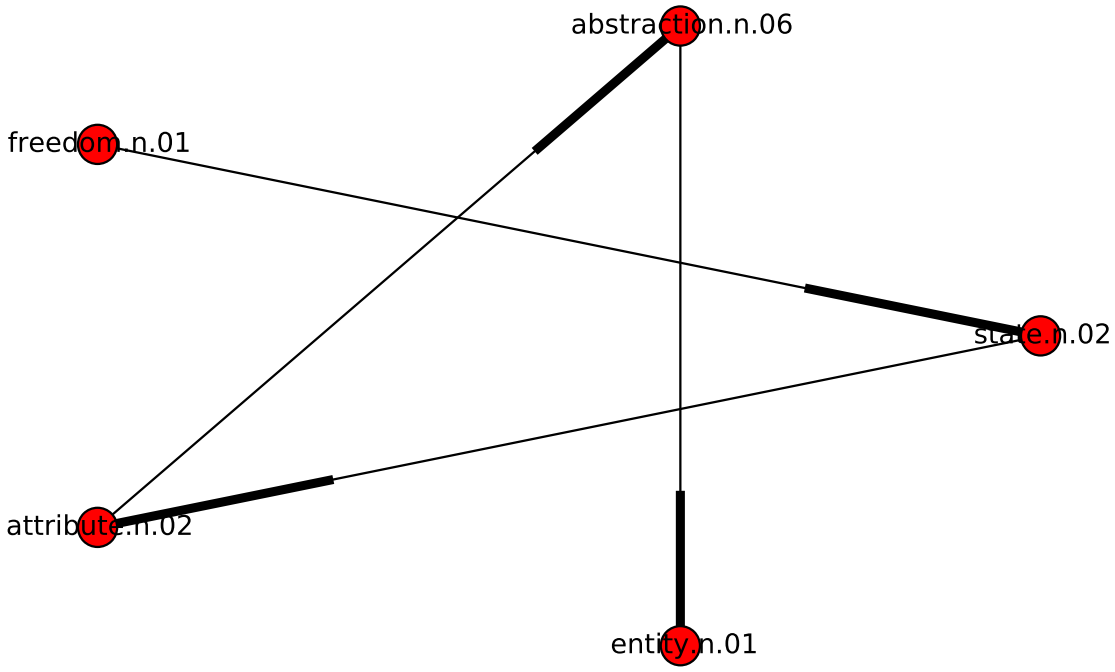
[**py18**]
```
>>> import networkx as nx

>>> def closure_graph(synset, fn):
...     seen = set()
...     graph = nx.DiGraph()
...     def recurse(s):
...         if not s in seen:
...             seen.add(s)
...             graph.add_node(s.name)
...             for s1 in fn(s):
...                 graph.add_node(s1.name)
...                 graph.add_edge(s.name, s1.name)
...                 recurse(s1)
...     recurse(synset)
...     return graph
...
```

[**py19**]
```
>>> motorcar = wn.synset('car.n.01')

>>> graph1 = closure_graph(motorcar, lambda s: s.hypernyms())
>>> nx.draw_shell(graph1, node_size=300, font_size=12)
>>> #nx.draw_spectral(graph1)
>>> #nx.draw(graph1)

>>> import matplotlib.pyplot as plt
>>> plt.savefig("graph1.pdf", dpi=600)
>>> plt.close()
```

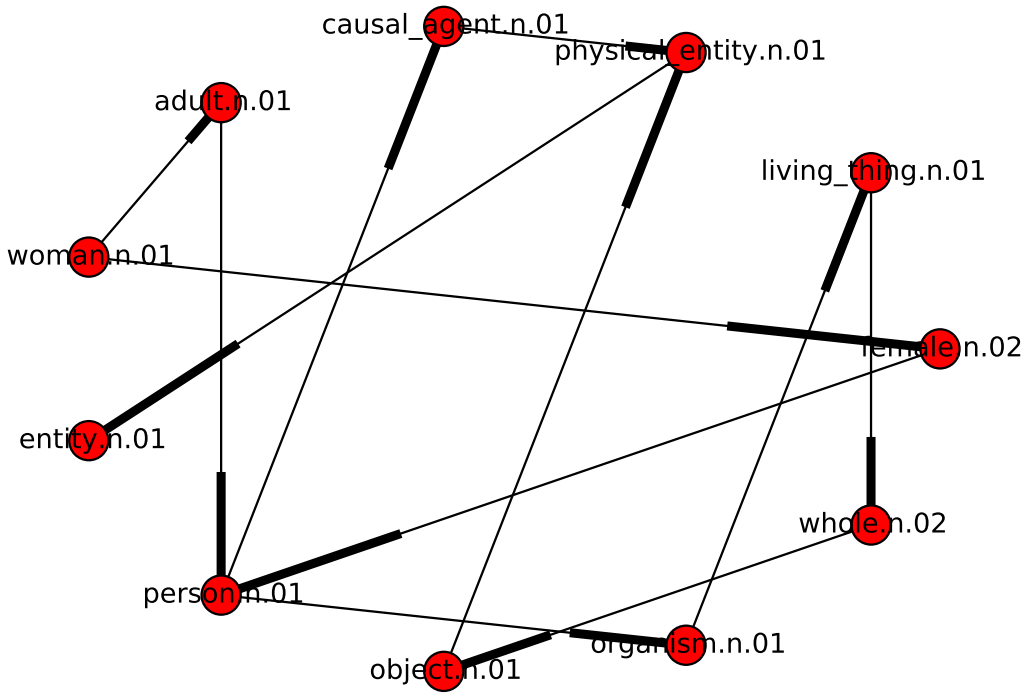
[1] Code based on this blog post: http://www.randomhacks.net/articles/2009/12/29/visualizing-wordnet-relationships-as-graphs.



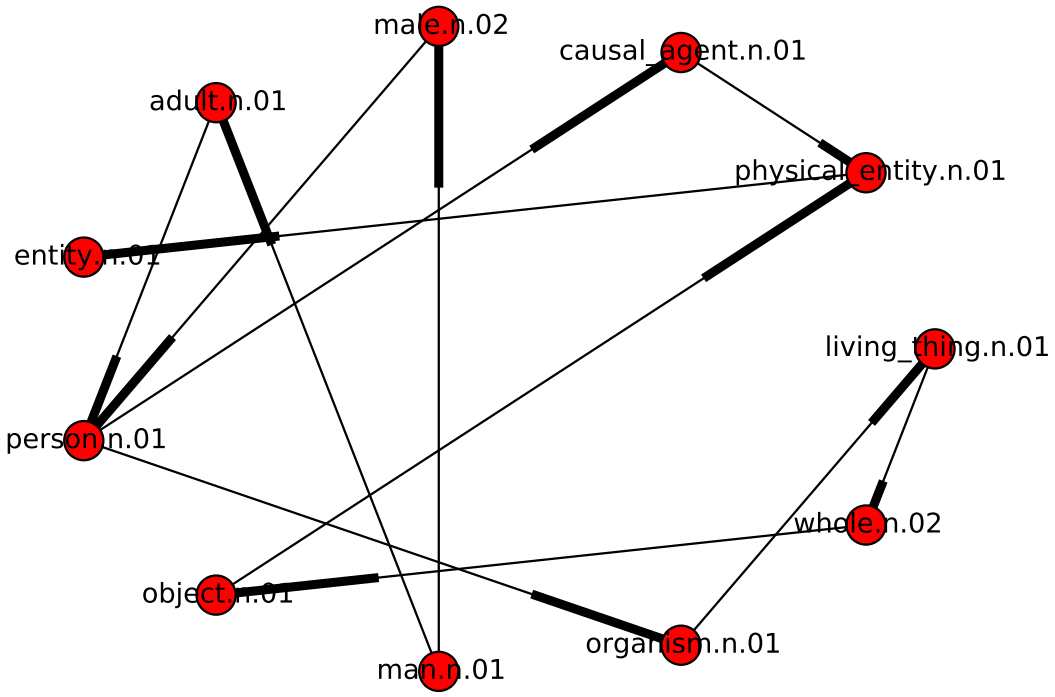
[**py20**]
```
>>> freedom = wn.synset('freedom.n.01')
>>> graph2 = closure_graph(freedom, lambda s: s.hypernyms())
>>> nx.draw_shell(graph2, node_size=300, font_size=12)
>>> plt.savefig("graph2.pdf", dpi=600)
>>> plt.close()
```




```
[py21] >>> woman = wn.synset('woman.n.01')
       >>> graph3 = closure_graph(woman, lambda s: s.hypernyms())
       >>> nx.draw_shell(graph3, node_size=300, font_size=12)
       >>> plt.savefig("graph3.pdf", dpi=600)
       >>> plt.close()
```



[**py22**]
```
>>> man = wn.synset('man.n.01')
>>> graph4 = closure_graph(man, lambda s: s.hypernyms())
>>> nx.draw_shell(graph4, node_size=300, font_size=12)
>>> plt.savefig("graph4.pdf", dpi=600)
>>> plt.close()
```



1.1 More lexical relations

Hypernyms and hyponyms are called lexical relations because they relate one synset to another.

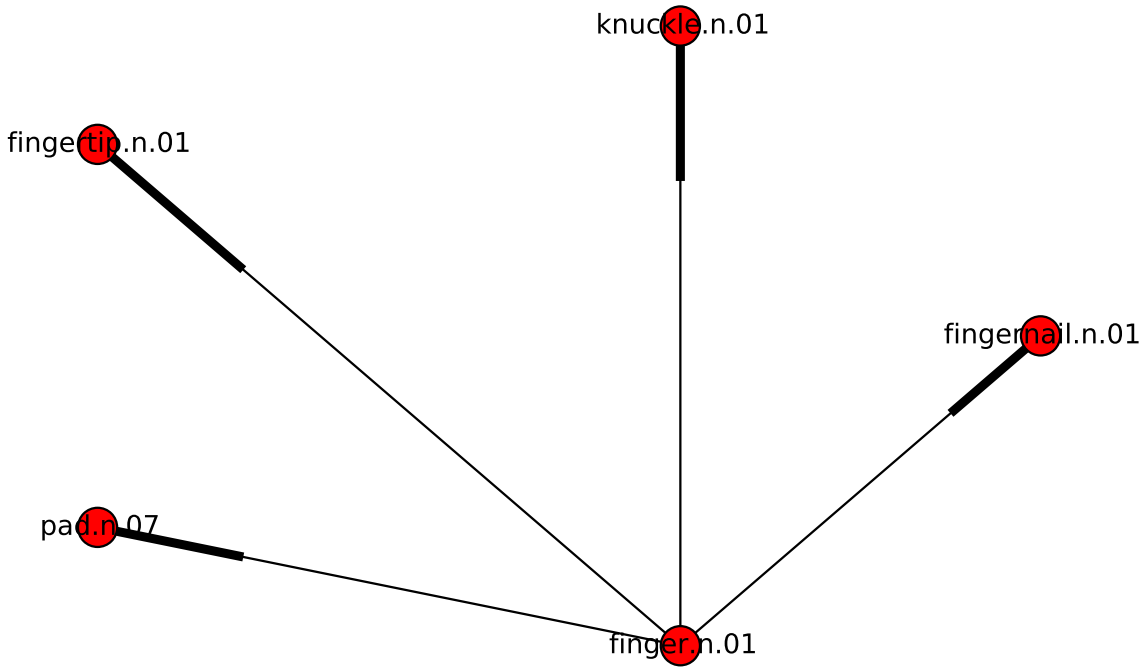
- these two relations navigate up and down the "is-a" hierarchy

Other important ways to navigate the WordNet network:

- from items to their components, i.e., their meronyms

[**py23**]
```
>>> wn.synset('tree.n.01').part_meronyms()
[Synset('burl.n.02'), Synset('crown.n.07'), Synset('stump.n.01'), Synset('trunk.n.01')
>>> wn.synset('tree.n.01').substance_meronyms()
[Synset('heartwood.n.01'), Synset('sapwood.n.01')]
```

[**py24**]
```
>>> finger = wn.synset('finger.n.01')
>>> graph5 = closure_graph(finger, lambda s: s.part_meronyms())
>>> nx.draw_shell(graph5, node_size=300, font_size=12)
>>> plt.savefig("graph5.pdf", dpi=600)
>>> plt.close()
```



- from items to the things they are contained in, i.e., their holonyms

[**py25**]
```
>>> wn.synset('tree.n.01').member_holonyms()
[Synset('forest.n.01')]
```

Consider for example the word *mint*.

[**py26**]
```
>>> for synset in wn.synsets('mint', wn.NOUN):
...     print synset.name + ':', synset.definition + "\n"
...
batch.n.02: (often followed by `of') a large number or amount or extent

mint.n.02: any north temperate plant of the genus Mentha with aromatic leaves and smal

mint.n.03: any member of the mint family of plants

mint.n.04: the leaves of a mint plant used fresh or candied

mint.n.05: a candy that is flavored with a mint oil

mint.n.06: a plant where money is coined by authority of the government
```

```
>>> wn.synset('mint.n.04').part_holonyms()
[Synset('mint.n.02')]
>>> wn.synset('mint.n.04').substance_holonyms()
[Synset('mint.n.05')]
```

There are also relationships between verbs. For example, the act of walking involves the act of stepping, so walking entails stepping. Some verbs have multiple entailments:

[**py27**]
```
>>> wn.synset('walk.v.01').entailments()
[Synset('step.v.01')]
>>> wn.synset('eat.v.01').entailments()
[Synset('swallow.v.01'), Synset('chew.v.01')]
>>> wn.synset('tease.v.03').entailments()
[Synset('arouse.v.07'), Synset('disappoint.v.01')]
```

Antonymy is defined as a relation between words / lemmas:

[**py28**]
```
>>> wn.lemma('supply.n.02.supply').antonyms()
[Lemma('demand.n.02.demand')]
>>> wn.lemma('rush.v.01.rush').antonyms()
[Lemma('linger.v.04.linger')]
>>> wn.lemma('horizontal.a.01.horizontal').antonyms()
[Lemma('vertical.a.01.vertical'), Lemma('inclined.a.02.inclined')]
```

2 Syntax (Context Free Grammars)

2.1 Syntactic / structural ambiguity

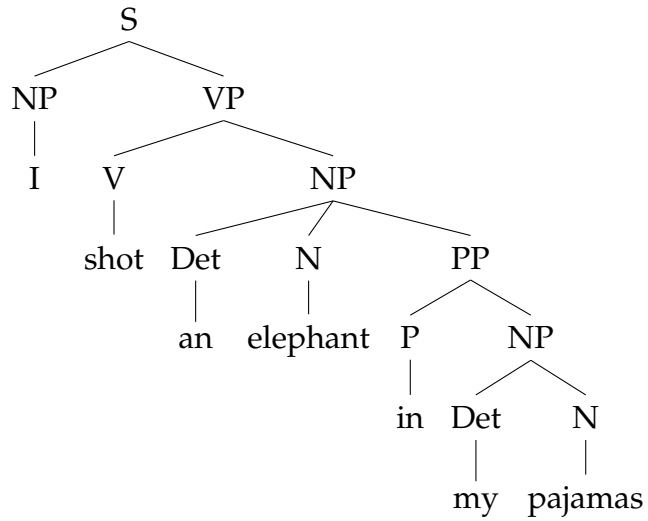
Consider the sentence:

(3) I shot an elephant in my pajamas.

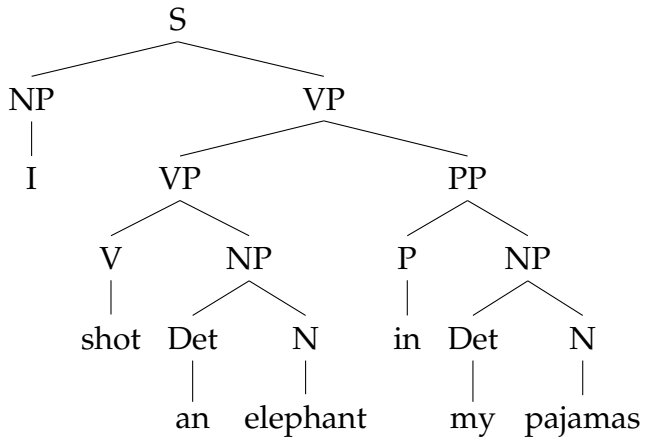
[**py29**]
```
>>> groucho_grammar = nltk.parse_cfg("""
...     S -> NP VP
...     PP -> P NP
...     NP -> Det N | Det N PP | 'I'
...     VP -> V NP | VP PP
...     Det -> 'an' | 'my'
...     N -> 'elephant' | 'pajamas'
...     V -> 'shot'
...     P -> 'in'
... """)

>>> sent = ['I', 'shot', 'an', 'elephant', 'in', 'my', 'pajamas']
>>> parser = nltk.ChartParser(groucho_grammar)
>>> trees = parser.nbest_parse(sent)
```

```
>>> for tree in trees:
...     print tree, "\n\n"
...
(S
  (NP I)
  (VP
    (V shot)
    (NP (Det an) (N elephant) (PP (P in) (NP (Det my) (N pajamas))))))


(S
  (NP I)
  (VP
    (VP (V shot) (NP (Det an) (N elephant)))
    (PP (P in) (NP (Det my) (N pajamas)))))


>>> with open("tree0.txt", "w") as file:
...     file.write(trees[0].pprint_latex_qtree())
...
>>> #trees[0].draw()
>>> with open("tree1.txt", "w") as file:
...     file.write(trees[1].pprint_latex_qtree())
...
>>> #trees[1].draw()
```

(4)



(5)



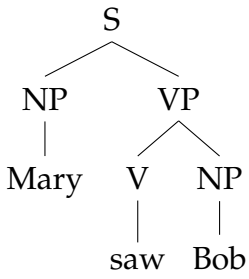
2.2 A simple CFG

[**py30**]
```
>>> grammar1 = nltk.parse_cfg("""
...     S -> NP VP
...     VP -> V NP | V NP PP
...     PP -> P NP
...     V -> "saw" | "ate" | "walked"
...     NP -> "John" | "Mary" | "Bob" | "I" | Det N | Det N PP
...     Det -> "a" | "an" | "the" | "my"
...     N -> "man" | "dog" | "cat" | "telescope" | "park"
...     P -> "in" | "on" | "by" | "with"
... """)
```

A production like `VP -> V NP | V NP PP` is an abbreviation for the two productions `VP -> V NP` and `VP -> V NP PP`.

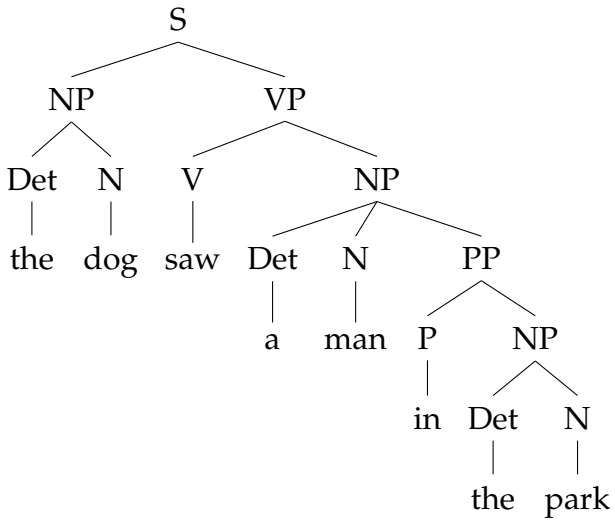
[**py31**]
```
>>> cp = nltk.ChartParser(grammar1)

>>> sent = "Mary saw Bob".split()
>>> trees = cp.nbest_parse(sent)
>>> len(trees)
1
>>> for tree in trees:
...     print tree
...
(S (NP Mary) (VP (V saw) (NP Bob)))
>>> with open("tree2.txt", "w") as file:
...     file.write(trees[0].pprint_latex_qtree())
...
```

(6)

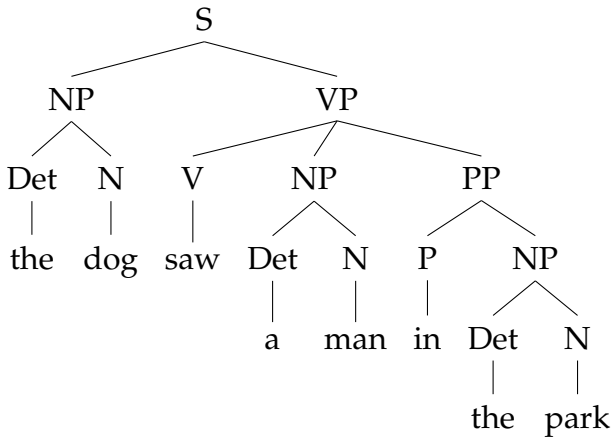


[**py32**]
```
>>> sent = "the dog saw a man in the park".split()
>>> trees = cp.nbest_parse(sent)
>>> len(trees)
2
>>> for tree in trees:
...     print tree
...
(S
  (NP (Det the) (N dog))
  (VP
    (V saw)
    (NP (Det a) (N man) (PP (P in) (NP (Det the) (N park))))))
(S
  (NP (Det the) (N dog))
  (VP
    (V saw)
    (NP (Det a) (N man))
    (PP (P in) (NP (Det the) (N park)))))
>>> with open("tree3.txt", "w") as file:
...     file.write(trees[0].pprint_latex_qtree())
...
>>> with open("tree4.txt", "w") as file:
...     file.write(trees[1].pprint_latex_qtree())
...
```

(7)

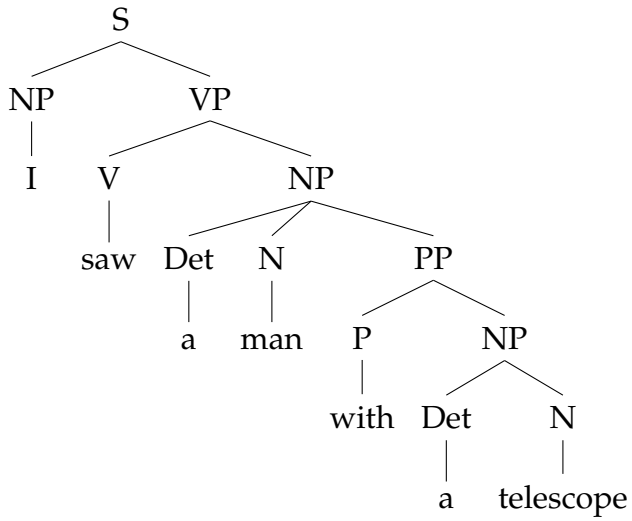


(8)

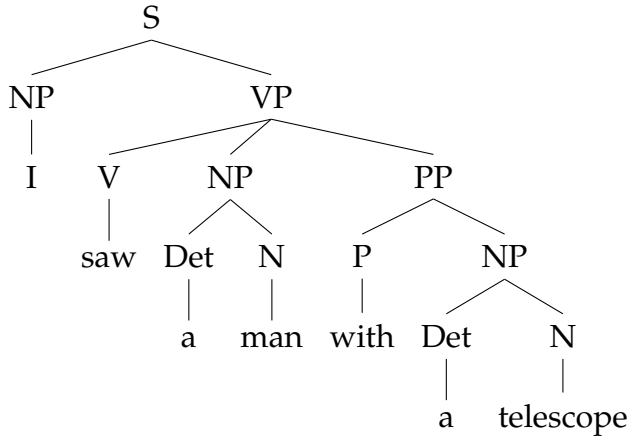


[**py33**]
```
>>> sent = "I saw a man with a telescope".split()
>>> trees = cp.nbest_parse(sent)
>>> len(trees)
2
>>> for tree in trees:
...     print tree
...
(S
  (NP I)
  (VP
    (V saw)
    (NP (Det a) (N man) (PP (P with) (NP (Det a) (N telescope))))))
(S
  (NP I)
  (VP
    (V saw)
    (NP (Det a) (N man))
    (PP (P with) (NP (Det a) (N telescope)))))
>>> with open("tree5.txt", "w") as file:
...     file.write(trees[0].pprint_latex_qtree())
...
>>> with open("tree6.txt", "w") as file:
...     file.write(trees[1].pprint_latex_qtree())
...
```

(9)



(10)

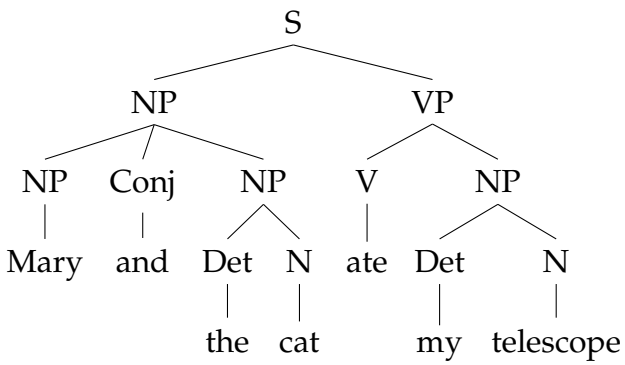


[**py34**]
```
>>> grammar1_1 = nltk.parse_cfg("""
...     S -> NP VP
...     VP -> V NP | V NP PP
...     PP -> P NP
...     V -> "saw" | "ate" | "walked"
...     Conj -> "and"
...     NP -> "John" | "Mary" | "Bob" | "I" | Det N | Det N PP | NP Conj NP
...     Det -> "a" | "an" | "the" | "my"
...     N -> "man" | "dog" | "cat" | "telescope" | "park"
...     P -> "in" | "on" | "by" | "with"
... """)
>>> cp1 = nltk.ChartParser(grammar1_1)
```

[**py35**]
```
>>> sent = "Mary and the cat ate my telescope".split()
>>> trees = cp1.nbest_parse(sent)
>>> len(trees)
1
>>> for tree in trees:
...     print tree
...
(S
```

```
  (NP (NP Mary) (Conj and) (NP (Det the) (N cat)))
  (VP (V ate) (NP (Det my) (N telescope))))
>>> with open("tree7.txt", "w") as file:
...     file.write(trees[0].pprint_latex_qtree())
...
```

(11)



[**py36**]
```
>>> import os
>>> import sys
>>> sys.path.append(os.getcwd())
>>> grammar2 = nltk.data.load('file:mygrammar.cfg')
```

(12)
```
S -> NP VP
VP -> V NP | V NP PP
PP -> P NP
V -> "saw" | "ate" | "walked"
NP -> "John" | "Mary" | "Bob" | "I" | Det N | Det N PP
Det -> "a" | "an" | "the" | "my"
N -> "man" | "dog" | "cat" | "telescope" | "park"
P -> "in" | "on" | "by" | "with"
```

- make sure that you put a `.cfg` suffix on the filename, and that there are no spaces in the string `'file:mygrammar.cfg'`
- you can check what productions are currently in the grammar as follows

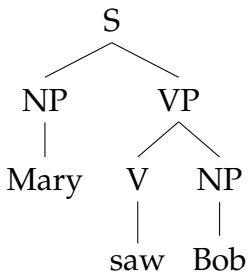
[**py37**]
```
>>> for p in grammar2.productions():
...     print p
...
S -> NP VP
VP -> V NP
VP -> V NP PP
PP -> P NP
V -> 'saw'
V -> 'ate'
V -> 'walked'
NP -> 'John'
NP -> 'Mary'
NP -> 'Bob'
```

```
NP -> 'I'
NP -> Det N
NP -> Det N PP
Det -> 'a'
Det -> 'an'
Det -> 'the'
Det -> 'my'
N -> 'man'
N -> 'dog'
N -> 'cat'
N -> 'telescope'
N -> 'park'
P -> 'in'
P -> 'on'
P -> 'by'
P -> 'with'
```

- if the command `print tree` produces no output, this is probably because your sentence `sent` is not admitted by your grammar

[**py38**]
```
>>> cp = nltk.ChartParser(grammar2)

>>> sent = "Mary saw Bob".split()
>>> trees = cp.nbest_parse(sent)
>>> len(trees)
1
>>> for tree in trees:
...     print tree
...
(S (NP Mary) (VP (V saw) (NP Bob)))
>>> with open("tree8.txt", "w") as file:
...     file.write(trees[0].pprint_latex_qtree())
...
```

(13)



- grammatical categories cannot be combined with lexical items on the righthand side of the same production, e.g., a production such as `PP -> 'of' NP` is disallowed
- multi-word lexical items are not allowed on the righthand side of a production, e.g., instead of writing `NP -> 'New York'`, we need to write something like `NP -> 'New_York'`

2.3 The L_{0E} grammar (Dowty et al. 1981)

We load the grammar:

[**py39**]
```
>>> grammar3 = nltk.data.load('file:Dowty_et_al.cfg')
```

This is the content of the file:

(14)
```
% start S
# ############################
# Phrase Structure Rules
# ############################

S -> S conj S
S -> neg S
S -> N VP
VP -> Vi
VP -> Vt N

# ############################
# Lexical Rules
# ############################

N -> 'Liz' | 'Sadie' | 'Hank'
Vi -> 'sleeps' | 'snores' | 'is-boring'
Vt -> 'loves' | 'hates' | 'is-taller-than'
conj -> 'and' | 'or'
neg -> 'it-is-not-the-case-that'
```

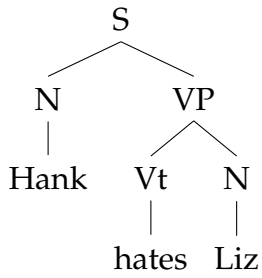
We can print the grammar productions. The list explicitly unpacks all the disjunctive ('|') rules:

[**py40**]
```
>>> for p in grammar3.productions():
...     print p
...
S -> S conj S
S -> neg S
S -> N VP
VP -> Vi
VP -> Vt N
N -> 'Liz'
N -> 'Sadie'
N -> 'Hank'
Vi -> 'sleeps'
Vi -> 'snores'
Vi -> 'is-boring'
Vt -> 'loves'
Vt -> 'hates'
Vt -> 'is-taller-than'
```

```
conj -> 'and'
conj -> 'or'
neg -> 'it-is-not-the-case-that'
```

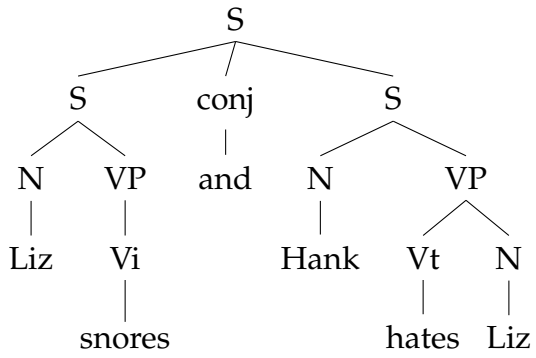
[**py41**]
```
>>> cp = nltk.ChartParser(grammar3)

>>> sent = "Hank hates Liz".split()
>>> sent
['Hank', 'hates', 'Liz']
>>> trees = cp.nbest_parse(sent)
>>> len(trees)
1
>>> print trees[0]
(S (N Hank) (VP (Vt hates) (N Liz)))
>>> with open("tree9.txt", "w") as file:
...     file.write(trees[0].pprint_latex_qtree())
...
```

(15)



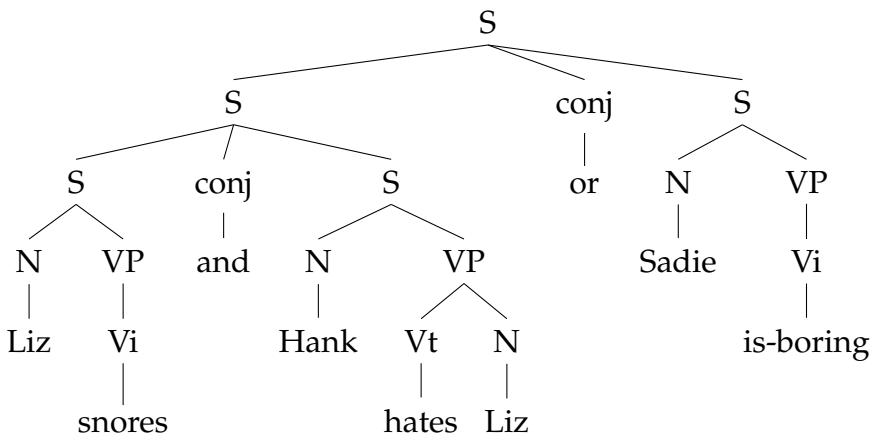
[**py42**]
```
>>> sent = "Liz snores and Hank hates Liz".split()
>>> trees = cp.nbest_parse(sent)
>>> len(trees)
1
>>> print trees[0]
(S
  (S (N Liz) (VP (Vi snores)))
  (conj and)
  (S (N Hank) (VP (Vt hates) (N Liz))))
>>> with open("tree10.txt", "w") as file:
...     file.write(trees[0].pprint_latex_qtree())
...
```

(16)

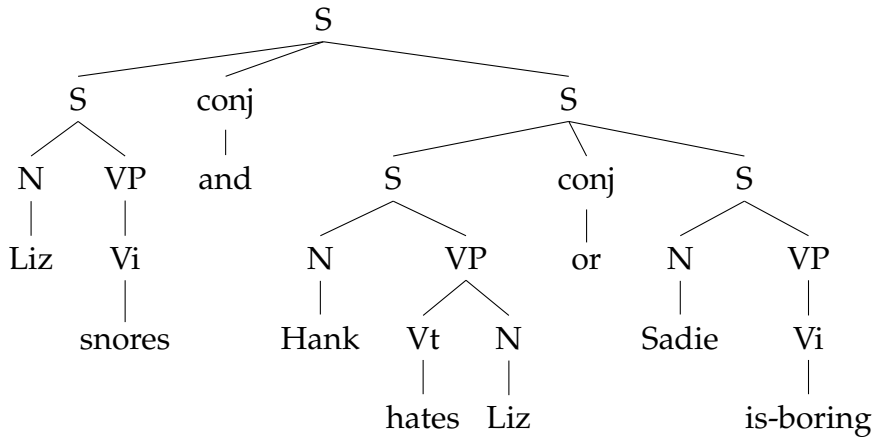


[**py43**]
```
>>> sent = "Liz snores and Hank hates Liz or Sadie is-boring".split()
>>> trees = cp.nbest_parse(sent)
>>> len(trees)
2
>>> print trees[0]
(S
  (S
    (S (N Liz) (VP (Vi snores)))
    (conj and)
    (S (N Hank) (VP (Vt hates) (N Liz))))
  (conj or)
  (S (N Sadie) (VP (Vi is-boring))))
>>> print trees[1]
(S
  (S (N Liz) (VP (Vi snores)))
  (conj and)
  (S
    (S (N Hank) (VP (Vt hates) (N Liz)))
    (conj or)
    (S (N Sadie) (VP (Vi is-boring)))))
>>> with open("tree11.txt", "w") as file:
...     file.write(trees[0].pprint_latex_qtree())
...
>>> with open("tree12.txt", "w") as file:
...     file.write(trees[1].pprint_latex_qtree())
...
```

(17)



(18)



2.4 Scaling Up

We have only considered 'toy' grammars. Can the approach be scaled up to cover large fragments of natural languages? How hard would it be to construct such a set of productions by hand? In general, the answer is: very hard.

Even if we allow ourselves to use various formal devices that give much more succinct representations of grammar productions, it is still extremely difficult to keep control of the complex interactions between the many productions required to cover the major constructions of a language.

That is, it is hard to modularize grammars so that one portion can be developed independently of the other parts.

This in turn means that it is difficult to distribute the task of grammar writing across a team of linguists.

Another difficulty is that as the grammar expands to cover a wider and wider range of constructions, there is a corresponding increase in the number of analyses which are admitted for any one sentence. *Ambiguity increases with coverage*.

Despite these problems, some large collaborative projects have achieved interesting and impressive results in developing rule-based grammars for several languages, e.g.:

- the Lexical Functional Grammar (LFG) Pargram project
- the Head-Driven Phrase Structure Grammar (HPSG) LinGO Matrix framework
- the Lexicalized Tree Adjoining Grammar XTAG Project

2.5 Treebanks and grammars

The usual way to develop broad coverage grammars is not by manually designing a grammar, but by using syntactically annotated corpora (tree banks) to induce grammars, i.e., obtain grammatical rules and various other parameters from the corpus and then generalize them in various ways.

The Penn Treebank corpus is probably the most well-known corpus providing syntactic annotation and other structural information for a fairly substantial sample of naturally-occuring text. In particular, the Penn Treebank provides:

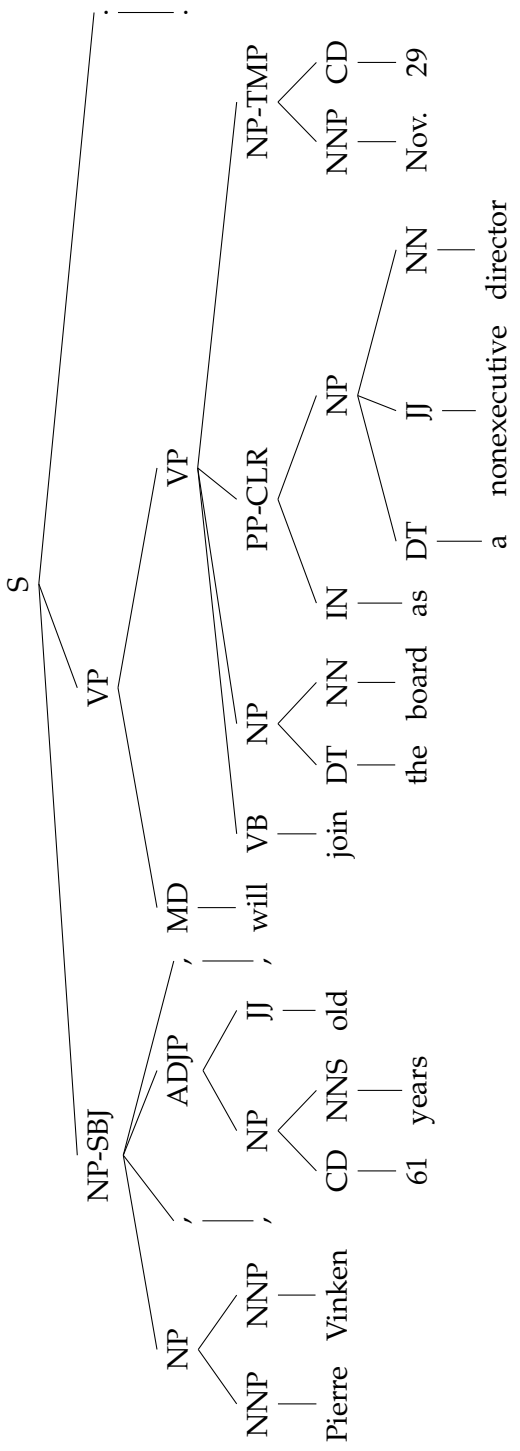
- skeletal parses showing rough syntactic and semantic information – a bank of linguistic trees
- part-of-speech tags
- for the Switchboard corpus of telephone conversations, dysfluency annotation

NLTK comes with 10% of the Penn Treebank corpus:

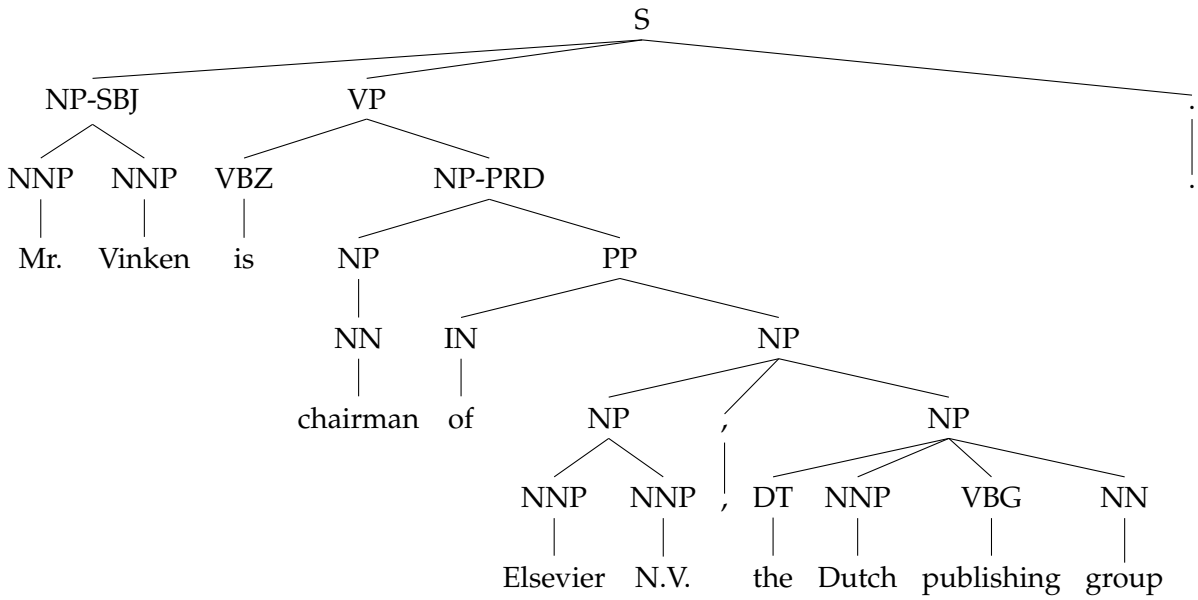
[**py44**]
```
>>> from nltk.corpus import treebank

>>> treebank.fileids()
['wsj_0001.mrg', 'wsj_0002.mrg', 'wsj_0003.mrg', 'wsj_0004.mrg', 'wsj_0005.mrg', 'wsj_
>>> len(treebank.fileids())
199
```

[**py45**]
```
>>> trees = treebank.parsed_sents('wsj_0001.mrg')
>>> len(trees)
2
>>> with open("tree13.txt", "w") as file:
...     file.write(trees[0].pprint_latex_qtree())
...
>>> with open("tree14.txt", "w") as file:
...     file.write(trees[1].pprint_latex_qtree())
...
```

(19)

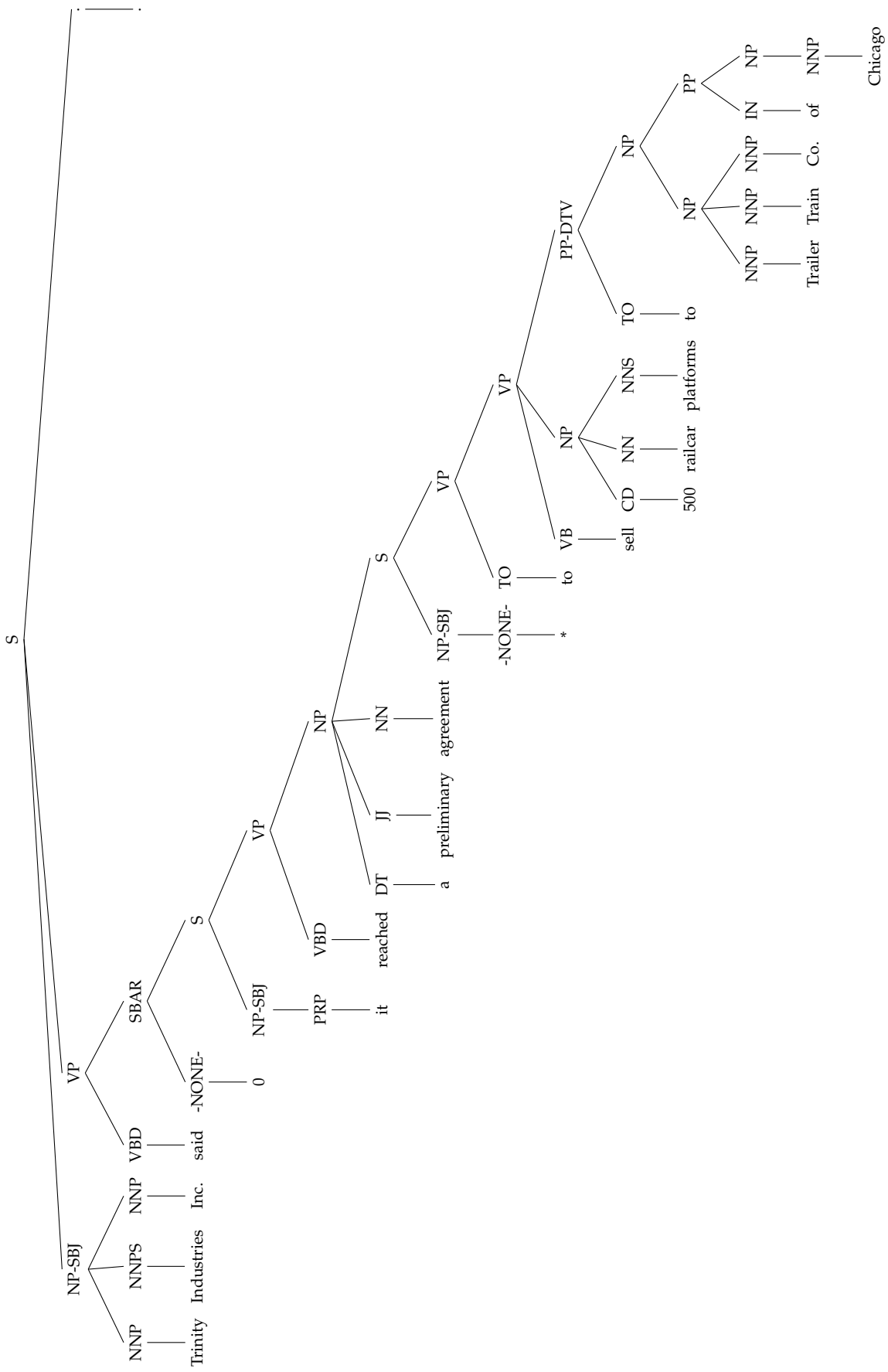

S
NP-SBJ
NP
NNP
Pierre
NNP
Vinken
,
ADJP
NP
CD
61
NNS
years
JJ
old
,
VP
MD
will
VP
VB
join
NP
DT
the
NN
board
PP-CLR
IN
as
NP
DT
a
JJ
nonexecutive
NN
director
NP-TMP
NNP
Nov.
CD
29
.
.

(20)

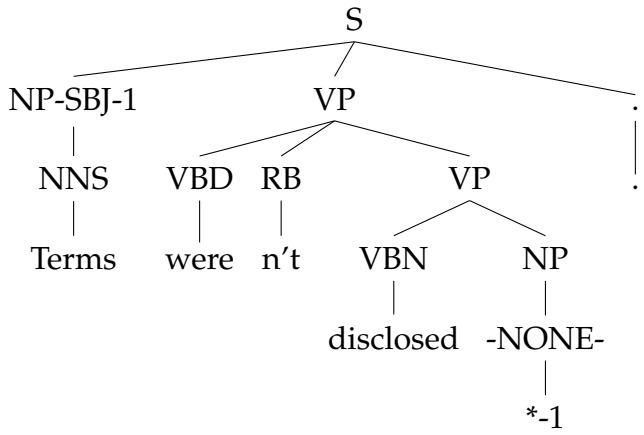


[**py46**]
```
>>> trees = treebank.parsed_sents('wsj_0199.mrg')
>>> len(trees)
3
>>> with open("tree15.txt", "w") as file:
...     file.write(trees[0].pprint_latex_qtree())
...
>>> with open("tree16.txt", "w") as file:
...     file.write(trees[1].pprint_latex_qtree())
...
```

(21)


S
NP-SBJ
NNP Trinity
NNPS Industries
NNP Inc.
VP
VBD said
SBAR
-NONE- 0
S
NP-SBJ
PRP it
VP
VBD reached
NP
DT a
JJ preliminary
NN agreement
S
NP-SBJ
-NONE- *
VP
TO to
VP
VB sell
PP-DTV
NP
CD 500
NN railcar
NNS platforms
TO to
NP
NP
NNP Trailer
NNP Train
NNP Co.
PP
IN of
NP
NNP Chicago
.

(22)



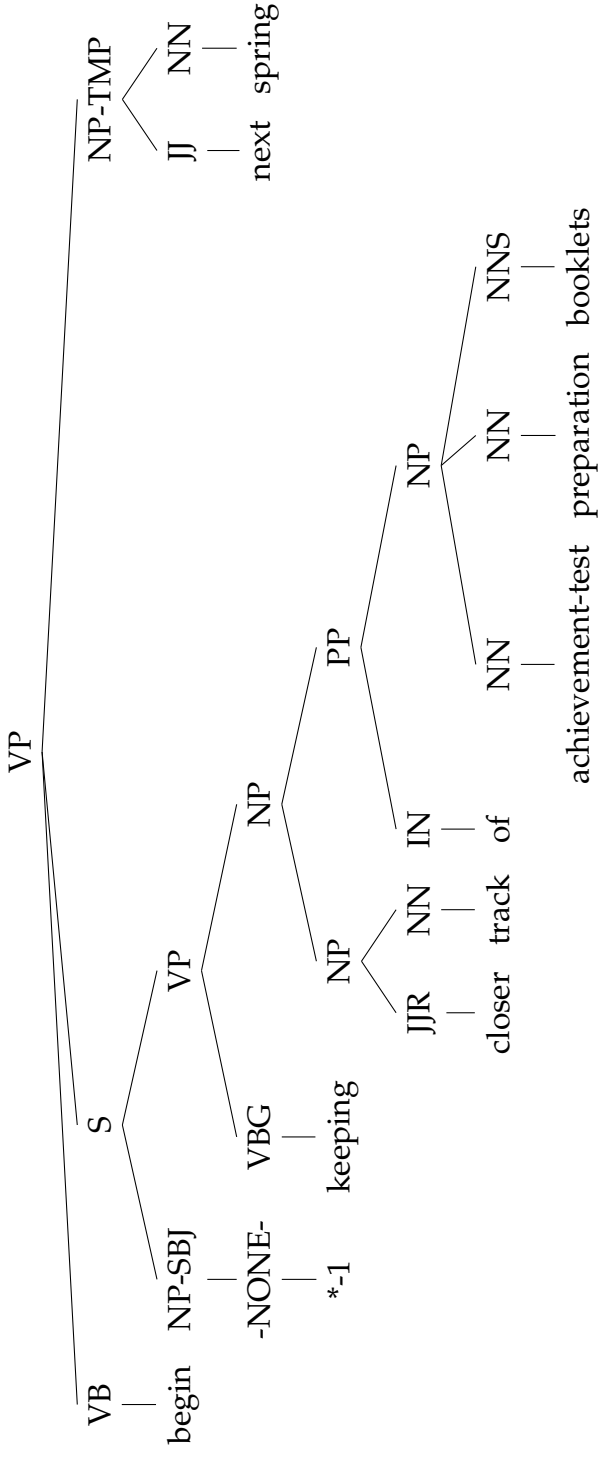
We can use this data to help develop a grammar.

- for example, we can use a simple filter to find verbs that take sentential complements
- assuming we already have a production of the form `VP -> Vs S`, this information enables us to identify particular verbs that would be included in the expansion of `Vs`.

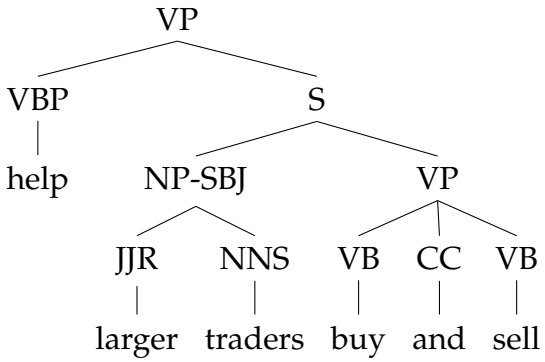
[**py47**]
```
>>> def filter(tree):
...     child_nodes = [child.node for child in tree \
...                                if isinstance(child, nltk.Tree)]
...     return (tree.node == 'VP') and ('S' in child_nodes)
...
>>> trees = [subtree for tree in treebank.parsed_sents() \
...                  for subtree in tree.subtrees(filter)]
>>> len(trees)
1111
```

[**py48**]
```
>>> with open("tree17.txt", "w") as file:
...     file.write(trees[201].pprint_latex_qtree())
...
>>> with open("tree18.txt", "w") as file:
...     file.write(trees[782].pprint_latex_qtree())
...
>>> with open("tree19.txt", "w") as file:
...     file.write(trees[1007].pprint_latex_qtree())
...
```

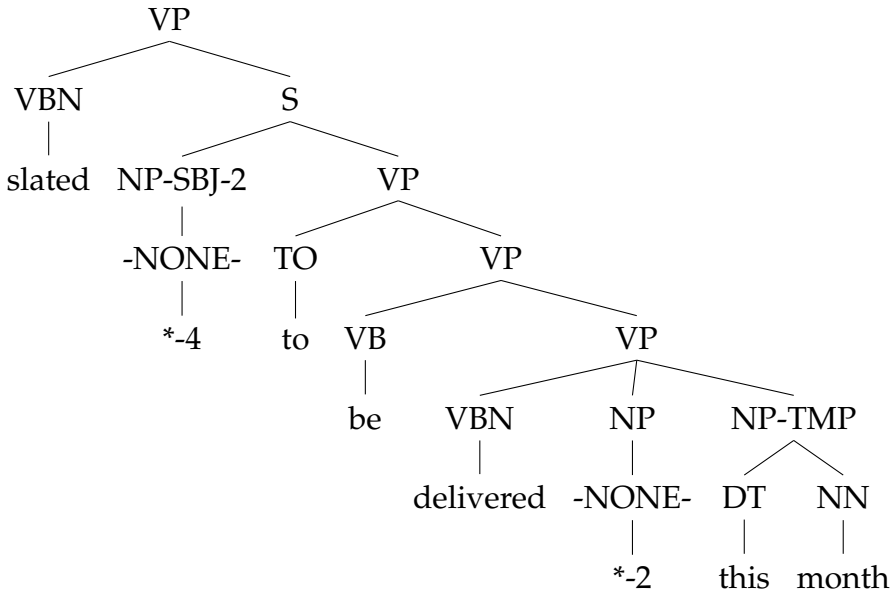
(23)


VP
VB
begin
S
NP-SBJ
-NONE-
*-1
VP
VBG
keeping
NP
NP
JJR
closer
NN
track
PP
IN
of
NP
NN
achievement-test
NN
preparation
NNS
booklets
NP-TMP
JJ
next
NN
spring

(24)



(25)



2.6 Pernicious Ambiguity

Unfortunately, as the coverage of the grammar increases and the length of the input sentences grows, the number of parse trees grows rapidly.

To see this, consider a simple example.

- the word *fish* is both a noun and a verb
- we can make up the sentence *fish fish fish*
- this sentence means that fish like to fish for other fish (try the same sentence with *police* instead of *fish*)

Here is a toy grammar for *fish* sentences.

[**py49**]
```
>>> grammar4 = nltk.parse_cfg("""
...     S -> N V N
...     N -> N Sbar
...     Sbar -> N V
...     N -> 'fish'
...     V -> 'fish'
... """)
```

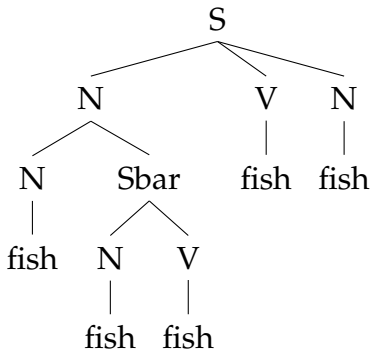
We can try parsing a longer sentence:

- *fish fish fish fish fish*
- among other things, this sentence means that fish that other fish fish are in the habit of fishing fish themselves

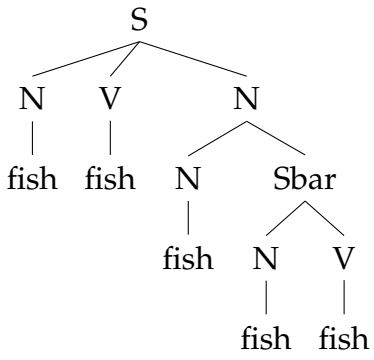
Let's see how many possible trees we have.

[**py50**]
```
>>> cp = nltk.ChartParser(grammar4)

>>> sent = "fish fish fish fish fish".split()
>>> trees = cp.nbest_parse(sent)
>>> len(trees)
2
>>> with open("tree20.txt", "w") as file:
...     file.write(trees[0].pprint_latex_qtree())
...
>>> with open("tree21.txt", "w") as file:
...     file.write(trees[1].pprint_latex_qtree())
...
```

(26)



(27)



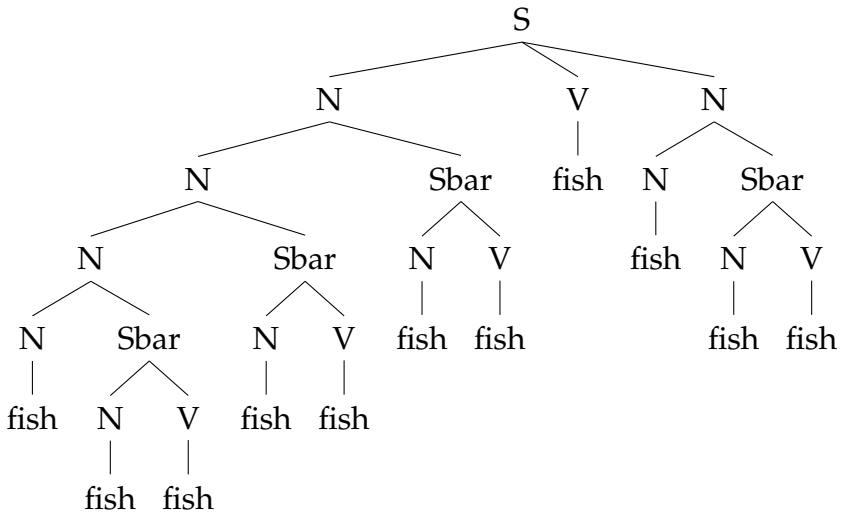
As the length of this sentence goes up $(3, 5, 7, 9, 11, \ldots)$ we get the following numbers of parse trees: $1, 2, 5, 14, 42 \ldots$ That is, the number of parse trees increases dramatically.

Let's test this for the *fish* sentence of length 11.

[**py51**]
```
>>> sent = "fish fish fish fish fish fish fish fish fish fish fish".split()
>>> sent
```

```
['fish', 'fish', 'fish', 'fish', 'fish', 'fish', 'fish', 'fish', 'fish', 'fish', 'fish
>>> len(sent)
11
>>> trees = cp.nbest_parse(sent)
>>> len(trees)
42
>>> with open("tree22.txt", "w") as file:
...     file.write(trees[21].pprint_latex_qtree())
...
```

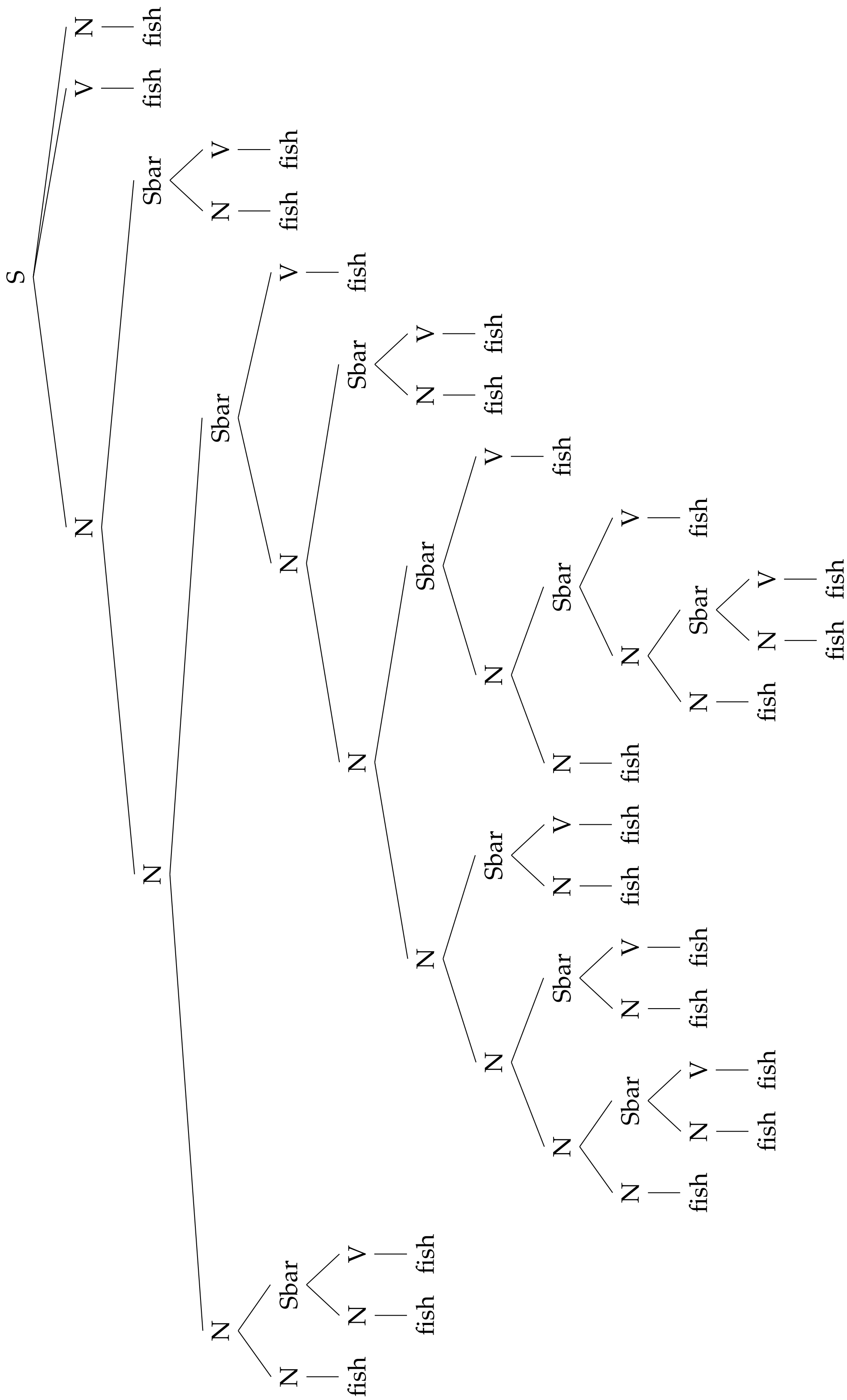
(28)



The average length of sentences in the Wall Street Journal section of Penn Treebank is 23. How many possible trees do you think there are for a *fish* sentence of length 23?

[**py52**]
```
>>> sent = ["fish"] * 23
>>> trees = cp.nbest_parse(sent)
>>> len(trees)
58786
>>> with open("tree23.txt", "w") as file:
...     file.write(trees[20222].pprint_latex_qtree())
...
```

(29)



So much for structural ambiguity. How about lexical ambiguity?

As soon as we try to construct a broad coverage grammar, we are forced to make lexical entries highly ambiguous for their part of speech.

- in a toy grammar, *a* is only a determiner, *dog* is only a noun, and *runs* is only a verb

- in a broad coverage grammar, *a* is also a noun (*part a*), *dog* is also a verb (meaning to follow closely), and *runs* is also a noun (*ski runs*)

So a parser for a broad-coverage grammar will be overwhelmed with ambiguity.

3 Probabilistic Context Free Grammars (PCFGs) and probabilistic parsing

Weighted grammars and probabilistic parsing algorithms have provided an effective solution to these problems.

3.1 Why gradient grammaticality?

Why would we think that the notion of grammaticality could be gradient?

Consider the verb *give*:

- it requires both a direct object (the thing being given) and an indirect object (the recipient)
- these complements can be given in either order

(30) a. Kim gave a bone to the dog. PREPOSITIONAL DATIVE
b. Kim gave the dog a bone. DOUBLE OBJECT

- PREPOSITIONAL DATIVE form: the direct object appears first, followed by a prepositional phrase containing the indirect object
- DOUBLE OBJECT form: the indirect object appears first, followed by the direct object

In the above example, either order is acceptable. However, if the indirect object is a pronoun, there is a strong preference for the double object construction:

(31) a. *Kim gives the heebie-jeebies to me. PREPOSITIONAL DATIVE
b. Kim gives me the heebie-jeebies. DOUBLE OBJECT

We can examine all the instances of prepositional dative and double object constructions involving *give* in the Penn Treebank sample:

[**py53**]
```
>>> def give(t):
...     return t.node == 'VP' and \
...     len(t) > 2 and t[1].node == 'NP' and \
...     (t[2].node == 'PP-DTV' or t[2].node == 'NP') and \
...     ('give' in t[0].leaves() or 'gave' in t[0].leaves())
...
>>> def sent(t):
...     return ' '.join(token for token in t.leaves() \
...                             if token[0] not in '*-0')
...
>>> def print_node(t, width):
```

```
...     output = "%s %s: %s / %s: %s" % \
...                 (sent(t[0]), t[1].node, sent(t[1]), t[2].node, sent(t[2]))
...     if len(output) > width:
...         output = output[:width] + "..."
...     print output
...
>>> for tree in nltk.corpus.treebank.parsed_sents():
...     for t in tree.subtrees(give):
...         print_node(t, 72)
...
gave NP: the chefs / NP: a standing ovation
give NP: advertisers / NP: discounts for maintaining or increasing ad sp...
give NP: it / PP-DTV: to the politicians
gave NP: them / NP: similar help
give NP: them / NP:
give NP: only French history questions / PP-DTV: to students in a Europe...
give NP: federal judges / NP: a raise
give NP: consumers / NP: the straight scoop on the U.S. waste crisis
gave NP: Mitsui / NP: access to a high-tech medical product
give NP: Mitsubishi / NP: a window on the U.S. glass industry
give NP: much thought / PP-DTV: to the rates she was receiving , nor to ...
give NP: your Foster Savings Institution / NP: the gift of hope and free...
give NP: market operators / NP: the authority to suspend trading in futu...
gave NP: quick approval / PP-DTV: to $ 3.18 billion in supplemental appr...
give NP: the Transportation Department / NP: up to 50 days to review any...
give NP: the president / NP: such power
give NP: me / NP: the heebie-jeebies
give NP: holders / NP: the right , but not the obligation , to buy a cal...
gave NP: Mr. Thomas / NP: only a `` qualified '' rating , rather than ``...
give NP: the president / NP: line-item veto power
```

We observe a strong tendency for the shortest complement to appear first. However, this does not account for a form like *give NP: federal judges / NP: a raise*, where animacy may play a role.

3.2 Defining and parsing with PCFGs

A probabilistic context free grammar (PCFG) is a context free grammar that associates a probability with each of its productions. It generates the same set of parses for a text that the corresponding context free grammar does, but in addition it assigns a probability to each parse.

The probability of a parse generated by a PCFG is simply the product of the probabilities of the productions used to generate it.

The simplest way to define a PCFG is as a sequence of weighted productions, with the weights appearing in square brackets.

[**py54**]
```
>>> grammar5 = nltk.parse_pcfg("""
...     S -> N VP [1.0]
```

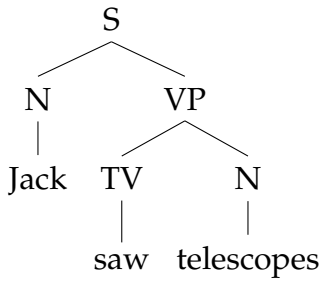
```
...     VP -> TV N [0.4]
...     VP -> IV [0.3]
...     VP -> DatV N N [0.3]
...     TV -> 'saw' [1.0]
...     IV -> 'ate' [1.0]
...     DatV -> 'gave' [1.0]
...     N -> 'telescopes' [0.8]
...     N -> 'Jack' [0.2]
... """)

>>> print grammar5
Grammar with 9 productions (start state = S)
    S -> N VP [1.0]
    VP -> TV N [0.4]
    VP -> IV [0.3]
    VP -> DatV N N [0.3]
    TV -> 'saw' [1.0]
    IV -> 'ate' [1.0]
    DatV -> 'gave' [1.0]
    N -> 'telescopes' [0.8]
    N -> 'Jack' [0.2]
```

- it is sometimes convenient to combine multiple productions into a single line, e.g., `VP -> TV N [0.4] | IV [0.3] | DatV N N [0.3]`
- this makes it easier to check that all productions with a given left-hand side have probabilities that sum to 1
- PCFG grammars impose this constraint so that the trees generated by the grammar form a probability distribution

The parse trees includes probabilities now:

[**py55**]
```
>>> vp = nltk.ViterbiParser(grammar5)

>>> sent = 'Jack saw telescopes'.split()
>>> tree = vp.parse(sent)
>>> print tree
(S (N Jack) (VP (TV saw) (N telescopes))) (p=0.064)
>>> with open("tree24.txt", "w") as file:
...     file.write(tree.pprint_latex_qtree())
...
```

(32)



Since parse trees are assigned probabilities, it no longer matters that there may be a huge number of possible parses for a given sentence. A parser will be responsible for finding *the most likely parse or parses*, and not all the parses.

4 Propositional logic

Here are the sentential connectives, a.k.a. boolean operators, defined in NLTK:

4.1 Syntax of propositional logic

[**py56**]
```
>>> nltk.boolean_ops()
negation           -
conjunction        &
disjunction        |
implication        ->
equivalence        <->

>>> lp = nltk.LogicParser()
```

[**py57**]
```
>>> formula = lp.parse('-(Liz_snores & Hank_hates_Liz)')
>>> formula
<NegatedExpression -(Liz_snores & Hank_hates_Liz)>
>>> print formula
-(Liz_snores & Hank_hates_Liz)

>>> lp.parse('Liz_snores & Hank_hates_Liz')
<AndExpression (Liz_snores & Hank_hates_Liz)>
>>> lp.parse('Liz_snores | (Hank_hates_Liz -> Liz_hates_Hank)')
<OrExpression (Liz_snores | (Hank_hates_Liz -> Liz_hates_Hank))>
>>> lp.parse('Liz_snores <-> --Liz_snores')
<IffExpression (Liz_snores <-> --Liz_snores)>
```

4.2 Interpretation, a.k.a. valuation

[**py58**]
```
>>> val = nltk.Valuation([('Liz_snores', True), \
...                       ('Hank_hates_Liz', True), \
...                       ('Liz_hates_Hank', False)])
```

```
>>> val['Hank_hates_Liz']
True

>>> dom = set([])
>>> g = nltk.Assignment(dom)

>>> m = nltk.Model(dom, val)
```

Every model comes with an `evaluate()` method, which will determine the semantic value of logical expressions.

[**py59**]
```
>>> print m.evaluate('Hank_hates_Liz', g)
True
>>> print m.evaluate('Liz_snores & Hank_hates_Liz', g)
True
>>> print m.evaluate('Liz_snores & Liz_hates_Hank', g)
False
>>> print m.evaluate('Hank_hates_Liz -> Liz_hates_Hank', g)
False
>>> print m.evaluate('Liz_snores | (Hank_hates_Liz -> Liz_hates_Hank)', g)
True
```

5 First-order logic

5.1 Syntax of first-order logic

We break sentences into predicates and arguments / terms.

- terms: individual variables (basically, pronouns) and individual constants (basically, names)
- predicates: *walk* is a unary / 1-place predicate, *see* is a binary / 2-place predicate etc.

It is often helpful to inspect the syntactic structure of expressions of first-order logic and the usual way of doing this is to assign types to expressions.

Following the Montagovian tradition, we use two basic types:

- e is the type of expressions like (proper) names that denote entities
- t is the type of formulas / sentences, i.e., expressions that denote truth values

Given these two basic types, we can form complex types for function expressions:

- $\langle e, t\rangle$ is the type of expressions that denote functions from entities to truth values, namely 1-place predicates
- $\langle e, \langle e, t\rangle\rangle$ is the type of expressions that denote functions from entities to functions from entities truth values, namely 2-place predicates

The LogicParser can be invoked so that it carries out type checking:

[**py60**]
```
>>> import nltk
>>> tlp = nltk.LogicParser(type_check=True)
>>> parsedSent = tlp.parse('walk(angus)')
>>> parsedSent.argument
<ConstantExpression angus>
>>> parsedSent.argument.type
e
>>> parsedSent.function
<ConstantExpression walk>
>>> parsedSent.function.type
<e,?>
```

Why do we see $\langle e, ? \rangle$ at the end of this example?

- the type-checker will try to infer as many types as possible
- but in this case, it has not managed to fully specify the type of 'walk' because its result type is unknown

We want *walk* to be of type $\langle e, t \rangle$ – so we need to specify this. This specification is sometimes called a *signature*:

[**py61**]
```
>>> sig = {'walk': '<e,t>'}
>>> parsed = tlp.parse('walk(angus)', sig)
>>> parsed.function.type
<e,t>
```

A binary predicate has type $\langle e, \langle e, t \rangle \rangle$, i.e., it combines first with an argument of type e to make a unary predicate. Then it combines with another argument to form a sentence.

But for readability, we represent binary predicates as combining with their two arguments simultaneously, e.g., *see(angus, cyril)* and not *see(cyril)(angus)*.

In first-order logic, arguments of predicates can also be individual variables x, y, z etc.

- individual variables are similar to personal pronouns like *he*, *she* and *it*, in that we need to know about the context of use in order to figure out their denotation

Consider the sentence below:

(33) He disappeared.

- one way of interpreting the pronoun in (33) is by pointing to a relevant individual in the local context
- another way is to supply a textual antecedent for the pronoun, for example by uttering (34) prior to (33)

(34) Cyril is a bad dog.
[He disappeared.]

- we say that *he* is coreferential with the NP *Cyril*

- in this case, (33) semantically equivalent to (35) below:

(35) Cyril disappeared.

Variables in first-order logic correspond to pronouns:

(36) a. He is a dog and he disappeared.
b. $dog(x)$ & $disappear(x)$

By placing an existential quantifier *existsx* / $\exists x$ (*for somex*), we can form new sentences in which these variables are interpreted as dependent pronouns

(37) *exists* x. ($dog(x)$ & $disappear(x)$), i.e.,
$\exists x(dog(x) \wedge disappear(x))$

(38) a. There is a dog and he disappeared.
b. There is a dog that disappeared.
c. A dog disappeared.

In addition to the existential quantifier, we have a universal quantifier *allx* / $\forall x$

(39) *all* x. ($dog(x)$ -> $disappear(x)$), i.e.,
$\forall x(dog(x) \rightarrow disappear(x))$

(40) a. For any entity x, if it is a dog, it disappeared.
b. All entities that are dogs disappeared.
c. Every dog disappeared.

The `parse()` method of NLTK's LogicParser returns objects of class `Expression`. For each expression, we can determine the set of variables that are free in it.

[**py62**]
```
>>> lp = nltk.LogicParser()
>>> lp.parse('dog(cyril)').free()
set([])
>>> lp.parse('dog(x)').free()
set([Variable('x')])
>>> lp.parse('own(angus, cyril)').free()
set([])
>>> lp.parse('exists x.dog(x)').free()
set([])
>>> lp.parse('exists x.own(x, y)').free()
set([Variable('y')])
>>> lp.parse('all y. (dog(y) -> exists x. (person(x) & own(x, y)))').free()
set([])
```

5.2 Semantics of first-order logic

Given a first-order logic language L, a model $\mathcal{M}$ for L is a pair $\langle D, Val \rangle$:

- D is a nonempty set called the domain of the model; it consists of the entities in our model

- *Val* is a function called the valuation function which assigns appropriate semantic values to the basic expressions of *L*

[**py63**]
```
>>> dom = set(['b', 'o', 'c'])
>>> v = """
...     bertie => b
...     olive => o
...     cyril => c
...     boy => {b}
...     girl => {o}
...     dog => {c}
...     walk => {o, c}
...     see => {(b, o), (c, b), (o, c)}
... """
>>> val = nltk.parse_valuation(v)
>>> print val
{'bertie': 'b',
 'boy': set([('b',)]),
 'cyril': 'c',
 'dog': set([('c',)]),
 'girl': set([('o',)]),
 'olive': 'o',
 'see': set([('b', 'o'), ('c', 'b'), ('o', 'c')]),
 'walk': set([('c',), ('o',)])}
```

Optional hw assignment: draw a picture for this valuation / model like we did in class.

The first-order logic counterpart of a context of use is a *variable assignment*: a mapping from individual variables to entities in the domain.

[**py64**]
```
>>> g = nltk.Assignment(dom, [('x', 'o'), ('y', 'c')])
>>> g
{'y': 'c', 'x': 'o'}
>>> print g
g[c/y][o/x]
```

We can evaluate atomic formulas:

- we create a model
- we call the `evaluate()` method to compute the truth value

[**py65**]
```
>>> m = nltk.Model(dom, val)
>>> m.evaluate('see(cyril, bertie)', g)
True
>>> m.evaluate('see(olive, y)', g)
True
>>> m.evaluate('see(y, x)', g)
False
```

```
>>> m.evaluate('see(bertie, olive) & boy(bertie) & -walk(bertie)', g)
True

>>> g
{'y': 'c', 'x': 'o'}
>>> g.purge()
>>> g
{}

>>> m.evaluate('exists x.(girl(x) & walk(x))', g)
True
>>> m.evaluate('girl(x) & walk(x)', g.add('x', 'o'))
True
```

We can identify *satisfiers* for a first-order formula with respect to a free variable:

[**py66**]
```
>>> fmla1 = lp.parse('girl(x) | boy(x)')
>>> m.satisfiers(fmla1, 'x', g)
set(['b', 'o'])
>>> fmla2 = lp.parse('girl(x) -> walk(x)')
>>> m.satisfiers(fmla2, 'x', g)
set(['c', 'b', 'o'])
>>> fmla3 = lp.parse('walk(x) -> girl(x)')
>>> m.satisfiers(fmla3, 'x', g)
set(['b', 'o'])

>>> m.evaluate('all x.(girl(x) -> walk(x))', g)
True
>>> m.satisfiers(fmla2, 'x', g)
set(['c', 'b', 'o'])

>>> m.evaluate('all x.(girl(x) | boy(x))', g)
False
>>> m.satisfiers(fmla1, 'x', g)
set(['b', 'o'])
```

6 A little bit of lambda calculus

λ, symbolized as \ in NLTK, is a variable-binding operator – just as the first-order logic quantifiers are.

- if we have an open formula such as (41a), then we can bind the variable x with the lambda operator, as shown in (41b)
- informally, the resulting expression is interpreted as shown in (41c) and (41d)

(41) a. `(walk(x) & chew_gum(x))`

b. `\x.(walk(x) & chew_gum(x))`
c. the set of entities x such that x walks and x chews gum
d. the entities that both walk and chew gum

[**py67**]
```
>>> lp = nltk.LogicParser()
>>> e = lp.parse(r'\x.(walk(x) & chew_gum(x))')
>>> e
<LambdaExpression \x.(walk(x) & chew_gum(x))>
>>> e.free()
set([])
>>> print lp.parse(r'\x.(walk(x) & chew_gum(y))')
\x.(walk(x) & chew_gum(y))
```

Another example:

(42) a. `\x.(walk(x) & chew_gum(x)) (gerald)`
b. gerald is one of the entities x that both walk and chew gum
c. gerald both walks and chews gum
d. `walk(gerald) & chew_gum(gerald)`

[**py68**]
```
>>> e = lp.parse(r'\x.(walk(x) & chew_gum(x)) (gerald)')
>>> print e
\x.(walk(x) & chew_gum(x))(gerald)
>>> print e.simplify()
(walk(gerald) & chew_gum(gerald))
```

More examples:

(43) a. `\x.(dog(x) & own(y, x))`
b. you give a paraphrase!

(44) a. `\x.\y.(dog(x) & own(y, x))`
b. you give a paraphrase!

[**py69**]
```
>>> print lp.parse(r'\x.\y.(dog(x) & own(y, x))(cyril)').simplify()
\y.(dog(cyril) & own(y,cyril))

>>> print lp.parse(r'\x.\y.(dog(x) & own(y, x))(cyril, angus)')
((\x y.(dog(x) & own(y,x)))(cyril))(angus)
>>> print lp.parse(r'\x.\y.(dog(x) & own(y, x))(cyril, angus)').simplify()
(dog(cyril) & own(angus,cyril))
```

Alpha conversion (a.k.a. renaming bound variables):

[**py70**]
```
>>> e1 = lp.parse('exists x.P(x)')
>>> print e1
exists x.P(x)
```

```
>>> e2 = e1.alpha_convert(nltk.sem.logic.Variable('z'))
>>> print e2
exists z.P(z)
>>> e1 == e2
True

>>> e3 = e2.alpha_convert(nltk.sem.logic.Variable('y17'))
>>> print e3
exists y17.P(y17)
>>> e2 == e3
True
```

6.1 Quantified NPs

Consider the simple sentences in (45a) and (46a) below:

(45) a. A dog barks.
b. `exists x.(dog(x) & bark(x))`

(46) a. No dog barks.
b. `-(exists x.(dog(x) & bark(x)))`

- we want to assign a separate meaning to the NPs *a dog* and *no dog* in much the same way in which we assign a separate meaning to *Cyril* in *Cyril barks*
- furthermore, whatever meaning we assign to these NPs, we want the final truth conditions for the English sentences in (45a) and (46a) to be the ones assigned to the first-order formulas in (45b) and (46b), respectively

We do this by lambda abstraction over higher-order variables – in particular, over property variables P, P' etc.

- *a dog*

[**py71**]
```
>>> lp.parse(r'(\P.exists x.(dog(x) & P(x)))')
<LambdaExpression \P.exists x.(dog(x) & P(x))>
```

- *barks*

[**py72**]
```
>>> lp.parse(r'\y.(bark(y))')
<LambdaExpression \y.bark(y)>
```

- *a dog barks*

[**py73**]
```
>>> S1 = lp.parse('(\P.exists x.(dog(x) & P(x)))(\y.(bark(y)))')
>>> print S1
(\P.exists x.(dog(x) & P(x)))(\y.bark(y))
>>> print S1.simplify()
exists x.(dog(x) & bark(x))
```

- *no dog*

[**py74**]
```
>>> lp.parse(r'(\P.-(exists x.(dog(x) & P(x))))')
<LambdaExpression \P.-exists x.(dog(x) & P(x))>
```

- *barks*

[**py75**]
```
>>> lp.parse(r'\y.(bark(y))')
<LambdaExpression \y.bark(y)>
```

- *no dog barks*

[**py76**]
```
>>> S2 = lp.parse('(\P.-(exists x.(dog(x) & P(x))))(\y.(bark(y)))')
>>> print S2
(\P.-exists x.(dog(x) & P(x)))(\y.bark(y))
>>> print S2.simplify()
-exists x.(dog(x) & bark(x))
```

We can provide a translation for the determiners *a* and *no* by adding another lambda abstraction. We exemplify with *no* only.

- *no*

[**py77**]
```
>>> lp.parse(r'(\P2.\P.-(exists x.(P2(x) & P(x))))')
<LambdaExpression \P2 P.-exists x.(P2(x) & P(x))>
```

- *dog*

[**py78**]
```
>>> lp.parse(r'\z.(dog(z))')
<LambdaExpression \z.dog(z)>
```

- *no dog*

[**py79**]
```
>>> NP1 = lp.parse(r'(\P2.\P.-(exists x.(P2(x) & P(x))))(\z.(dog(z)))')
>>> print NP1
(\P2 P.-exists x.(P2(x) & P(x)))(\z.dog(z))
>>> print NP1.simplify()
\P.-exists x.(dog(x) & P(x))
```

- *barks*

[**py80**]
```
>>> lp.parse(r'\y.(bark(y))')
<LambdaExpression \y.bark(y)>
```

- *no dog barks*

[**py81**]
```
>>> S3 = lp.parse( \
...      r'((\P2.\P.-(exists x.(P2(x) & P(x))))(\z.(dog(z)))(\y.(bark(y))))')
>>> print S3
((\P2 P.-exists x.(P2(x) & P(x)))(\z.dog(z)))(\y.bark(y))
>>> print S3.simplify()
-exists x.(dog(x) & bark(x))
```

6.2 Transitive verbs with quantified NPs as direct objects

This is the simplest translation for the transitive verb *see*:

[**py82**]
```
>>> lp.parse(r'\x2.\x1.see(x1,x2)')
<LambdaExpression \x2 x1.see(x1,x2)>
```

It works great for direct objects that are proper names, e.g., *Cyril saw Angus*.

[**py83**]
```
>>> S4 = lp.parse(r'(\x2.\x1.see(x1,x2)(angus))(cyril)')
>>> print S4
((\x2 x1.see(x1,x2))(angus))(cyril)
>>> print S4.simplify()
see(cyril,angus)
```

But it doesn't work for direct objects that are quantified NPs, e.g., *Cyril saw no boy*:

- the final formula for this sentence should be as follows

[**py84**]
```
>>> print lp.parse(r'-(exists x.(boy(x) & see(cyril,x)))')
-exists x.(boy(x) & see(cyril,x))
```

- but instead, we get this

[**py85**]
```
>>> S5 = lp.parse( \
...       r'(\x2.\x1.see(x1,x2)((\P.-(exists x.(boy(x) & P(x))))))(cyril)')
>>> print S5
((\x2 x1.see(x1,x2))(\P.-exists x.(boy(x) & P(x))))(cyril)
>>> print S5.simplify()
see(cyril,\P.-exists x.(boy(x) & P(x)))
```

We can give a more general translation in which a transitive verb lambda abstracts over a higher-order variable that has the same type as quantified NPs:

- *saw*

[**py86**]
```
>>> TV1 = lp.parse(r'(\Q.\x1.Q(\x2.see(x1,x2)))')
>>> print TV1
\Q x1.Q(\x2.see(x1,x2))
```

- *no boy*

[**py87**]
```
>>> lp.parse(r'(\P.-(exists x.(boy(x) & P(x))))')
<LambdaExpression \P.-exists x.(boy(x) & P(x))>
```

- *saw no boy*

[py88]
```
>>> VP1 = lp.parse(
...       r'((\Q.\x1.Q(\x2.see(x1,x2)))(\P.-(exists x.(boy(x) & P(x)))))')
>>> print VP1
(\Q x1.Q(\x2.see(x1,x2)))(\P.-exists x.(boy(x) & P(x)))
>>> print VP1.simplify()
\x1.-exists x.(boy(x) & see(x1,x))
```

- *Cyril saw no boy*

[py89]
```
>>> S6 = lp.parse(
...      r'((\Q.\x1.Q(\x2.see(x1,x2)))(\P.-(exists x.(boy(x) & P(x)))))(cyril)')
>>> print S6
((\Q x1.Q(\x2.see(x1,x2)))(\P.-exists x.(boy(x) & P(x))))(cyril)
>>> print S6.simplify()
-exists x.(boy(x) & see(cyril,x))
```

7 Discourse Representation Theory (DRT)

A discourse is a sequence of sentences. Often, the interpretation of a sentence in a discourse depends on the preceding sentences.

For example, pronouns like *he*, *she* and *it* can be anaphoric to (i.e., take their reference from) previous indefinite NPs. For example:

(47) a. Angus owns a dog.
b. It bit Irene.

The second sentence in (47) is interpreted as: the previously mentioned dog (owned by Angus) bit Irene.

The first-order formula below does not capture this interpretation (why?):

[py90]
```
>>> print lp.parse(r'exists x.(dog(x) & own(angus,x)) & bite(x,irene)')
(exists x.(dog(x) & own(angus,x)) & bite(x,irene))
```

The approach to quantification in first-order logic is limited to single sentences. But (47) seems to be a case in which the scope of a quantifier can extend over two sentences. In particular, the first-order formula below captures the correct interpretation of (47) (why exactly?):

[py91]
```
>>> print lp.parse(r'exists x.(dog(x) & own(angus,x) & bite(x,irene))')
exists x.(dog(x) & own(angus,x) & bite(x,irene))
```

Discourse Representation Theory (DRT) was developed with the specific goal of providing a means for handling this and other semantic phenomena that seem to be characteristic of discourse.

A discourse representation structure (DRS) presents the meaning of discourse in terms of a list of discourse referents (i.e., variables) and a list of conditions.

- the discourse referents are the things under discussion in the discourse; they correspond to the individual variables of first-order logic

- the DRS conditions constrain the value of those discourse referents; they correspond to (atomic) open formulas of first-order logic

The first sentence (47a) is represented in DRT as follows.

[**py92**]
```
>>> dp = nltk.DrtParser()
>>> drs1 = dp.parse('([x, y], [angus(x), dog(y), own(x, y)])')
>>> print drs1
([x,y],[angus(x), dog(y), own(x,y)])
>>> drs1.pprint()
 ----------
| x y      |
|----------|
| angus(x) |
| dog(y)   |
| own(x,y) |
|__________|
>>> #drs1.draw()
```

7.1 DRT to FOL translation

Every DRS can be translated into a formula of first-order logic:

[**py93**]
```
>>> print drs1.fol()
exists x y.(angus(x) & dog(y) & own(x,y))
```

DRT expressions have a DRS-concatenation operator, represented as '+' (this is really just dynamic conjunction ';')

- the concatenation of two DRSs is a single DRS containing the merged discourse referents and the conditions from both arguments
- DRS-concatenation automatically alpha-converts bound variables to avoid name-clashes

[**py94**]
```
>>> drs2 = dp.parse('([z],[irene(z),bite(y,z)])')
>>> print drs2
([z],[irene(z), bite(y,z)])
>>> drs2.pprint()
 -----------
| z         |
|-----------|
| irene(z)  |
| bite(y,z) |
|___________|

>>> drs3 = dp.parse('([x, y], [angus(x), dog(y), own(x, y)]) + ([z],[irene(z),bite(y,z
>>> print drs3
(([x,y],[angus(x), dog(y), own(x,y)]) + ([z],[irene(z), bite(y,z)]))
```

```
>>> drs3.pprint()
  ----------     -----------
 | x y      |   | z         |
(|----------| + |-----------|)
 | angus(x) |   | irene(z)  |
 | dog(y)   |   | bite(y,z) |
 | own(x,y) |   |___________|
 |__________|
>>> drs3s = drs3.simplify()
>>> print drs3s
([x,y,z],[angus(x), dog(y), own(x,y), irene(z), bite(y,z)])
>>> drs3s.pprint()
 -----------
| x y z     |
|-----------|
| angus(x)  |
| dog(y)    |
| own(x,y)  |
| irene(z)  |
| bite(y,z) |
|___________|
```

Note that the first-order translation of the simplified `drs3s` provides the intuitively correct truth conditions for discourse (47):

[**py95**]
```
>>> print drs3s.fol()
exists x y z.(angus(x) & dog(y) & own(x,y) & irene(z) & bite(y,z))
```

7.2 Embedded DRSs

It is possible to embed one DRS within another. This is how universal quantification is handled.

[**py96**]
```
>>> drs5 = dp.parse('([], [(([x], [dog(x)]) -> ([y],[ankle(y), bite(x, y)]))])')
>>> print drs5
([],[(([x],[dog(x)]) -> ([y],[ankle(y), bite(x,y)]))])
>>> drs5.pprint()
 ---------------------------------
|                                 |
|---------------------------------|
|   --------      -----------   |
|  | x      |    | y         |  |
| (|--------| -> |-----------|) |
|  | dog(x) |    | ankle(y)  |  |
|  |________|    | bite(x,y) |  |
|               |___________|  |
|_________________________________|
>>> print drs5.fol()
all x.(dog(x) -> exists y.(ankle(y) & bite(x,y)))
```

We capture basic examples of donkey anaphora. Both sentences below are represented in the same way and the correct truth conditions (for the strong reading) are derived.

(48) Every farmer who owns a donkey beats it.

(49) If a farmer owns a donkey, he beats it.

[**py97**]
```
>>> drs6 = dp.parse('([], [(([x,y], [farmer(x),donkey(y),own(x,y)]) -> ([],[beat(x,y)]
>>> print drs6
([],[(([x,y],[farmer(x), donkey(y), own(x,y)]) -> ([],[beat(x,y)]))])
>>> drs6.pprint()
 ___________________________________
|                                   |
|___________________________________|
|   ___________       ___________   |
|  | x y       |     |           |  |
| (|-----------| ->  |-----------|) |
|  | farmer(x) |     | beat(x,y) |  |
|  | donkey(y) |     |___________|  |
|  | own(x,y)  |                    |
|  |___________|                    |
|___________________________________|
>>> print drs6.fol()
all x y.((farmer(x) & donkey(y) & own(x,y)) -> beat(x,y))
```

7.3 Anaphora resolution in more detail

DRT is designed to allow anaphoric pronouns to be interpreted by linking to existing discourse referents.

When we represented (47) above, we automatically resolved the pronoun in the second sentence.

But we want an explicit step of anaphora resolution because resolving anaphors is in general not as straightforward as it is in (47).

DRT sets constraints on which discourse referents are *accessible* as possible antecedents, but does not intended to explain how a particular antecedent is chosen from the set of candidates.

The module `nltk.sem.drt_resolve_anaphora` adopts a similarly conservative strategy: if a DRS contains a condition of the form `PRO(x)`, the method `resolve_anaphora()` replaces this with a condition of the form `x = [...]`, where `[...]` is a list of possible antecedents

[**py98**]
```
>>> drs7 = dp.parse('([x, y], [angus(x), dog(y), own(x, y)])')
>>> drs8 = dp.parse('([u, z], [PRO(u), irene(z), bite(u, z)])')
>>> drs9 = drs7 + drs8
>>> drs9s = drs9.simplify()
>>> print drs9s
([u,x,y,z],[angus(x), dog(y), own(x,y), PRO(u), irene(z), bite(u,z)])
>>> drs9s.pprint()
 ___________
| u x y z   |
```

```
|-----------|
| angus(x)  |
| dog(y)    |
| own(x,y)  |
| PRO(u)    |
| irene(z)  |
| bite(u,z) |
|___________|
>>> print drs9s.resolve_anaphora()
([u,x,y,z],[angus(x), dog(y), own(x,y), (u = [x,y,z]), irene(z), bite(u,z)])
>>> drs9s.resolve_anaphora().pprint()
 ---------------
| u x y z       |
|---------------|
| angus(x)      |
| dog(y)        |
| own(x,y)      |
| (u = [x,y,z]) |
| irene(z)      |
| bite(u,z)     |
|_______________|
```

Since the algorithm for anaphora resolution has been separated into its own module, this facilitates swapping in alternative procedures which try to make more intelligent guesses about the correct antecedent.

References

Bird, Steven et al. (2009). *Natural Language Processing with Python*. O'Reilly Media.

Dowty, D.R. et al. (1981). *Introduction to Montague Semantics*. Studies in Linguistics and Philosophy. Springer.

Poore, Geoffrey M. (2013). "Reproducible Documents with PythonTeX". In: *Proceedings of the 12th Python in Science Conference*. Ed. by Stéfan van der Walt et al., pp. 78–84.