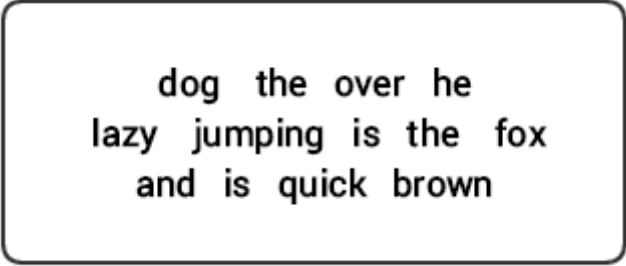

By [Dipanjan Sarkar](#), Data Science Lead at Applied Materials.

 [comments](#)

For any language, syntax and structure usually go hand in hand, where a set of specific rules, conventions, and principles govern the way words are combined into phrases; phrases get combined into clauses; and clauses get combined into sentences. We will be talking specifically about the English language syntax and structure in this section. **In English, words usually combine together to form other constituent units. These constituents include words, phrases, clauses, and sentences.** Considering a sentence, “*The brown fox is quick and he is jumping over the lazy dog*”, it is made of a bunch of words and just looking at the words by themselves don’t tell us much.



dog the over he
lazy jumping is the fox
and is quick brown

A bunch of unordered words don’t convey much information

Knowledge about the structure and syntax of language is helpful in many areas like text processing, annotation, and parsing for further operations such as text classification or summarization. Typical parsing techniques for understanding text syntax are mentioned below.

- **Parts of Speech (POS) Tagging**
- **Shallow Parsing or Chunking**
- **Constituency Parsing**
- **Dependency Parsing**

We will be looking at all of these techniques in subsequent sections. Considering our previous example sentence “*The brown fox is quick and he is jumping over the lazy dog*”, if we were to annotate it using basic POS tags, it would look like the following figure.

DET	ADJ	N	V	ADJ	CONJ	PRON	V	V	ADV	DET	ADJ	N
The	brown	fox	is	quick	and	he	is	jumping	over	the	lazy	dog

POS tagging for a sentence

Thus, a sentence typically follows a hierarchical structure consisting the following components,

sentence → clauses → phrases → words

Tagging Parts of Speech

Parts of speech (POS) are specific lexical categories to which words are assigned, **based on their syntactic context and role.** Usually, words can fall into one of the following major categories.

- **N(oun):** This usually denotes words that depict some object or entity, which may be living or nonliving. Some examples would be fox , dog , book , and so on. The POS tag symbol for nouns is N.

- ***V(erb)***: Verbs are words that are used to describe certain actions, states, or occurrences. There are a wide variety of further subcategories, such as auxiliary, reflexive, and transitive verbs (and many more). Some typical examples of verbs would be running , jumping , read , and write . The POS tag symbol for verbs is **V**.
- ***Adj(ective)***: Adjectives are words used to describe or qualify other words, typically nouns and noun phrases. The phrase beautiful flower has the noun (N) flower which is described or qualified using the adjective (ADJ) beautiful . The POS tag symbol for adjectives is **ADJ** .
- ***Adv(erb)***: Adverbs usually act as modifiers for other words including nouns, adjectives, verbs, or other adverbs. The phrase very beautiful flower has the adverb (ADV) very , which modifies the adjective (ADJ) beautiful , indicating the degree to which the flower is beautiful. The POS tag symbol for adverbs is **ADV**.

Besides these four major categories of parts of speech , there are other categories that occur frequently in the English language. These include pronouns, prepositions, interjections, conjunctions, determiners, and many others. Furthermore, each POS tag like the *noun* (N) can be further subdivided into categories like *singular nouns* (NN), *singular proper nouns*(NNP), and *plural nouns* (NNS).

The process of classifying and labeling POS tags for words called *parts of speech tagging* or *POS tagging* . POS tags are used to annotate words and depict their POS, which is really helpful to perform specific analysis, such as narrowing down upon nouns and seeing which ones are the most prominent, word sense disambiguation, and grammar analysis. We will be leveraging both **nltk** and **spacy** which usually use the [Penn Treebank notation](#) for POS tagging.

```

1  # create a basic pre-processed corpus, don't lowercase to get POS context
2  corpus = normalize_corpus(news_df['full_text'], text_lower_case=False,
3                             text_lemmatization=False, special_char_removal=False)
4
5  # demo for POS tagging for sample news headline
6  sentence = str(news_df.iloc[1].news_headline)
7  sentence_nlp = nlp(sentence)
8
9  # POS tagging with Spacy
10 spacy_pos_tagged = [(word, word.tag_, word.pos_) for word in sentence_nlp]
11 pd.DataFrame(spacy_pos_tagged, columns=['Word', 'POS tag', 'Tag type'])
12
13 # POS tagging with nltk
14 nltk_pos_tagged = nltk.pos_tag(sentence.split())
15 pd.DataFrame(nltk_pos_tagged, columns=['Word', 'POS tag'])

```

nlp_strategy_10.py hosted with ❤ by GitHub

[view raw](#)

	Word	POS tag	Tag type		Word	POS tag
0	US	NNP	PROPN	0	US	NNP
1	unveils	VBZ	VERB	1	unveils	VBZ
2	world	NN	NOUN	2	world's	VBZ
3	's	POS	PART	3	most	RBS
4	most	RBS	ADV	4	powerful	JJ
5	powerful	JJ	ADJ	5	supercomputer,	JJ
6	supercomputer	NN	NOUN	6	beats	NNS
7	,	,	PUNCT	7	China	NNP
8	beats	VBZ	VERB			
9	China	NNP	PROPN			

SpaCy POS tagging

NLTK POS tagging

POS tagging a news headline

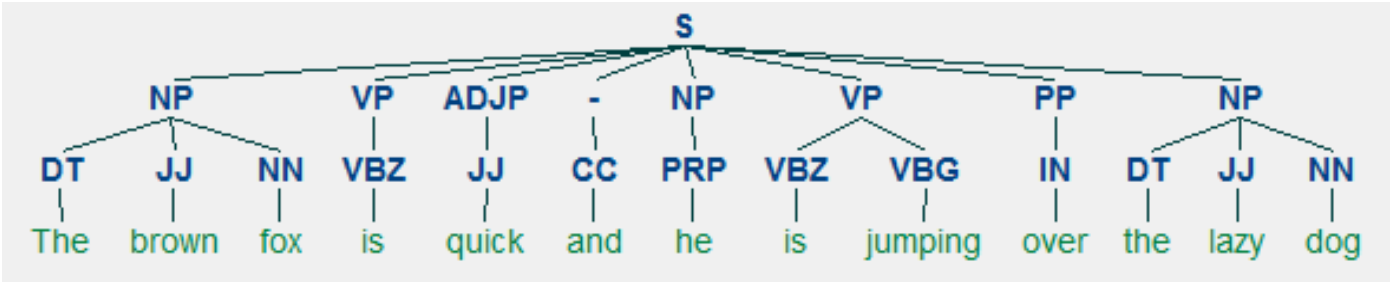
We can see that each of these libraries treat tokens in their own way and assign specific tags for them. Based on what we see, `spacy` seems to be doing slightly better than `nltk`.

Shallow Parsing or Chunking

Based on the hierarchy we depicted earlier, groups of words make up phrases. There are five major categories of phrases:

- **Noun phrase (NP):** These are phrases where a noun acts as the head word. Noun phrases act as a subject or object to a verb.
- **Verb phrase (VP):** These phrases are lexical units that have a verb acting as the head word. Usually, there are two forms of verb phrases. One form has the verb components as well as other entities such as nouns, adjectives, or adverbs as parts of the object.
- **Adjective phrase (ADJP):** These are phrases with an adjective as the head word. Their main role is to describe or qualify nouns and pronouns in a sentence, and they will be either placed before or after the noun or pronoun.
- **Adverb phrase (ADVP):** These phrases act like adverbs since the adverb acts as the head word in the phrase. Adverb phrases are used as modifiers for nouns, verbs, or adverbs themselves by providing further details that describe or qualify them.
- **Prepositional phrase (PP):** These phrases usually contain a preposition as the head word and other lexical components like nouns, pronouns, and so on. These act like an adjective or adverb describing other words or phrases.

Shallow parsing, also known as light parsing or chunking, is a popular natural language processing technique of analyzing the structure of a sentence to break it down into its smallest constituents (which are tokens such as words) and group them together into higher-level phrases. This includes POS tags as well as phrases from a sentence.



An example of shallow parsing depicting higher level phrase annotations

We will leverage the **conll2000** corpus for training our shallow parser model. This corpus is available in **nltk** with chunk annotations and we will be using around 10K records for training our model. A sample annotated sentence is depicted as follows.

```
1  from nltk.corpus import conll2000
2
3  data = conll2000.chunked_sents()
4  train_data = data[:10900]
5  test_data = data[10900:]
6
7  print(len(train_data), len(test_data))
8  print(train_data[1])
```

nlp_strategy_11.py hosted with ❤ by GitHub

[view raw](#)

```
10900 48
(S
  Chancellor/NNP
  (PP of/IN)
  (NP the/DT Exchequer/NNP)
  (NP Nigel/NNP Lawson/NNP)
  (NP 's/POS restated/VBN commitment/NN)
  (PP to/TO)
  (NP a/DT firm/NN monetary/JJ policy/NN)
  (VP has/VBZ helped/VBN to/TO prevent/VB)
  (NP a/DT freefall/NN)
  (PP in/IN)
  (NP sterling/NN)
  (PP over/IN)
  (NP the/DT past/JJ week/NN)
  ./.)
```

From the preceding output, you can see that our data points are sentences that are already annotated with phrases and POS tags metadata that will be useful in training our shallow parser model. We will leverage two chunking utility functions, `tree2conlltags`, to get triples of word, tag, and chunk tags for each token, and `conlltags2tree` to generate a parse tree from these token triples. We will be using these functions to train our parser. A sample is depicted below.

```
1  from nltk.chunk.util import tree2conlltags, conlltags2tree
2
3  wtc = tree2conlltags(train_data[1])
4  wtc
```

nlp_strategy_12.py hosted with ❤ by GitHub

[view raw](#)

```
[('Chancellor', 'NNP', 'O'),
 ('of', 'IN', 'B-PP'),
 ('the', 'DT', 'B-NP'),
 ('Exchequer', 'NNP', 'I-NP'),
 ('Nigel', 'NNP', 'B-NP'),
 ('Lawson', 'NNP', 'I-NP'),
 (''s', 'POS', 'B-NP'),
 ('restated', 'VBN', 'I-NP'),
```

```
( 'commitment', 'NN', 'I-NP'),
( 'to', 'TO', 'B-PP'),
( 'a', 'DT', 'B-NP'),
( 'firm', 'NN', 'I-NP'),
( 'monetary', 'JJ', 'I-NP'),
( 'policy', 'NN', 'I-NP'),
( 'has', 'VBZ', 'B-VP'),
( 'helped', 'VBN', 'I-VP'),
( 'to', 'TO', 'I-VP'),
( 'prevent', 'VB', 'I-VP'),
( 'a', 'DT', 'B-NP'),
( 'freefall', 'NN', 'I-NP'),
( 'in', 'IN', 'B-PP'),
( 'sterling', 'NN', 'B-NP'),
( 'over', 'IN', 'B-PP'),
( 'the', 'DT', 'B-NP'),
( 'past', 'JJ', 'I-NP'),
( 'week', 'NN', 'I-NP'),
( '.', '.', 'O')]
```

The chunk tags use the IOB format. This notation represents Inside, Outside, and Beginning. The B- prefix before a tag indicates it is the beginning of a chunk, and I- prefix indicates that it is inside a chunk. The O tag indicates that the token does not belong to any chunk. The B- tag is always used when there are subsequent tags of the same type following it without the presence of O tags between them.

We will now define a function `conll_tag_chunks()` to extract POS and chunk tags from sentences with chunked annotations and a function called `combined_taggers()` to train multiple taggers with backoff taggers (e.g. unigram and bigram taggers)

```
1 def conll_tag_chunks(chunk_sents):
2     tagged_sents = [tree2conlltags(tree) for tree in chunk_sents]
3     return [(t, c) for (w, t, c) in sent] for sent in tagged_sents]
4
5
6 def combined_tagger(train_data, taggers, backoff=None):
7     for tagger in taggers:
8         backoff = tagger(train_data, backoff=backoff)
9     return backoff
```

nlp_strategy_13.py hosted with ❤ by GitHub

[view raw](#)

We will now define a class `NGramTagChunker` that will take in tagged sentences as training input, get their (*word, POS tag, Chunk tag*) **WTC triples**, and train a `BigramTagger` with a `UnigramTagger` as the backoff tagger. We will also define a `parse()` function to perform shallow parsing on new sentences

The `UnigramTagger`, `BigramTagger`, and `TrigramTagger` are classes that inherit from the base class `NGramTagger`, which itself inherits from the `ContextTagger` class, which inherits from the `SequentialBackoffTagger` class.

We will use this class to train on the `conll2000` chunked `train_data` and evaluate the model performance on the `test_data`

```
1 from nltk.tag import UnigramTagger, BigramTagger
2 from nltk.chunk import ChunkParserI
3
```

```

4  # define the chunker class
5  class NGramTagChunker(ChunkParserI):
6
7      def __init__(self, train_sentences,
8                    tagger_classes=[UnigramTagger, BigramTagger]):
9          train_sent_tags = conll_tag_chunks(train_sentences)
10         self.chunk_tagger = combined_tagger(train_sent_tags, tagger_classes)
11
12     def parse(self, tagged_sentence):
13         if not tagged_sentence:
14             return None
15         pos_tags = [tag for word, tag in tagged_sentence]
16         chunk_pos_tags = self.chunk_tagger.tag(pos_tags)
17         chunk_tags = [chunk_tag for (pos_tag, chunk_tag) in chunk_pos_tags]
18         wpc_tags = [(word, pos_tag, chunk_tag) for ((word, pos_tag), chunk_tag)
19                    in zip(tagged_sentence, chunk_tags)]
20         return conlltags2tree(wpc_tags)
21
22     # train chunker model
23     ntc = NGramTagChunker(train_data)
24
25     # evaluate chunker model performance
26     print(ntc.evaluate(test_data))

```

nlp_strategy_14.py hosted with ❤ by GitHub

[view raw](#)

ChunkParse score:

IOB Accuracy:	90.0%
Precision:	82.1%
Recall:	86.3%
F-Measure:	84.1%

Our chunking model gets an accuracy of around 90% which is quite good! Let's now leverage this model to shallow parse and chunk our sample news article headline which we used earlier, ***"US unveils world's most powerful supercomputer, beats China"***.

```

chunk_tree = ntc.parse(nltk_pos_tagged)
print(chunk_tree)

```

Output:

```

(S
  (NP US/NNP)
  (VP unveils/VBZ world's/VBZ)
  (NP most/RBS powerful/JJ supercomputer,/JJ beats/NNS China/NNP))

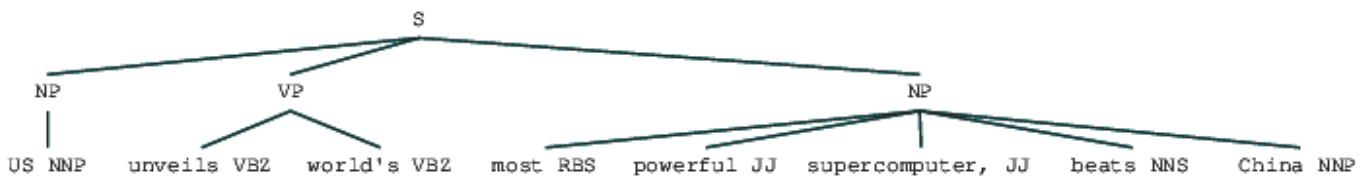
```

Thus you can see it has identified two noun phrases (NP) and one verb phrase (VP) in the news article. Each word's POS tags are also visible. We can also visualize this in the form of a tree as follows. You might need to install [ghostscript](#) in case `nltk` throws an error.

```
1 from IPython.display import display
2
3 ## download and install ghostscript from https://www.ghostscript.com/download/gsdnld.html
4
5 # often need to add to the path manually (for windows)
6 os.environ['PATH'] = os.environ['PATH']+";C:\\Program Files\\gs\\gs9.09\\bin\\"
7
8 display(chunk_tree)
```

nlp_strategy_15.py hosted with ❤ by GitHub

[view raw](#)



Shallow parsed news headline

The preceding output gives a good sense of structure after shallow parsing the news headline.

Constituency Parsing

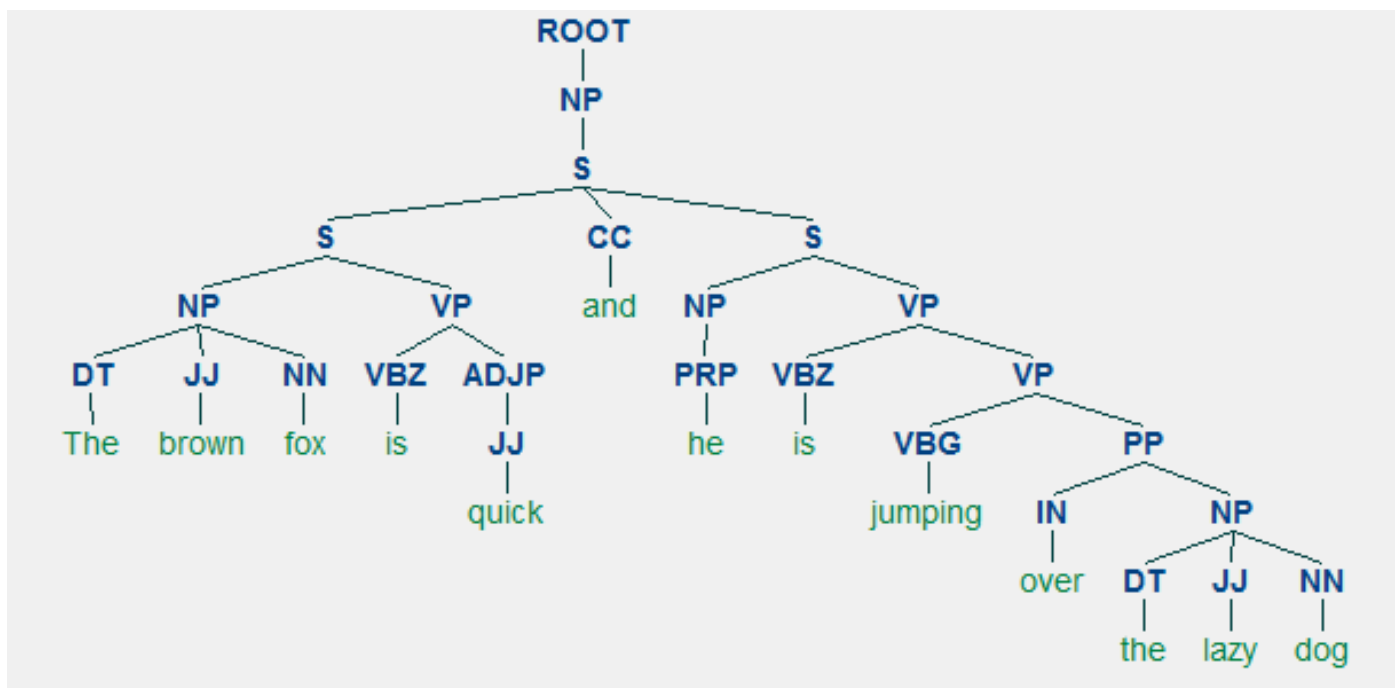
Constituent-based grammars are used to analyze and determine the constituents of a sentence. These grammars can be used to model or represent the internal structure of sentences in terms of a hierarchically ordered structure of their constituents. Each and every word usually belongs to a specific lexical category in the case and forms the head word of different phrases. These phrases are formed based on rules called *phrase structure rules*.

Phrase structure rules form the core of constituency grammars, because they talk about syntax and rules that govern the hierarchy and ordering of the various constituents in the sentences. These rules cater to two things primarily.

- They determine what words are used to construct the phrases or constituents.
- They determine how we need to order these constituents together.

The generic representation of a phrase structure rule is $S \rightarrow AB$, which depicts that the structure S consists of constituents A and B , and the ordering is A followed by B . While there are several rules (refer to Chapter 1, Page 19: *Text Analytics with Python, if you want to dive deeper*), the most important rule describes how to divide a sentence or a clause. The phrase structure rule denotes a binary division for a sentence or a clause as $S \rightarrow NP VP$ where S is the sentence or clause, and it is divided into the subject, denoted by the noun phrase (NP) and the predicate, denoted by the verb phrase (VP).

A constituency parser can be built based on such grammars/rules, which are usually collectively available as context-free grammar (CFG) or phrase-structured grammar. The parser will process input sentences according to these rules, and help in building a parse tree.



An example of constituency parsing showing a nested hierarchical structure

We will be using **nltk** and the **StanfordParser** here to generate parse trees.

Prerequisites: Download the official Stanford Parser from [here](#), which seems to work quite well. You can try out a later version by going to [this website](#) and checking the **Release History** section. After downloading, unzip it to a known location in your filesystem. Once done, you are now ready to use the parser from **nltk**, which we will be exploring soon.

The Stanford parser generally uses a **PCFG (probabilistic context-free grammar) parser**. A PCFG is a context-free grammar that associates a probability with each of its production rules. The probability of a parse tree generated from a PCFG is simply the production of the individual probabilities of the productions used to generate it.

```
1 # set java path
2 import os
3 java_path = r'C:\Program Files\Java\jdk1.8.0_102\bin\java.exe'
4 os.environ['JAVAHOME'] = java_path
5
6 from nltk.parse.stanford import StanfordParser
7
8 scp = StanfordParser(path_to_jar='E:/stanford/stanford-parser-full-2015-04-20/stanford-parser.jar',
9                       path_to_models_jar='E:/stanford/stanford-parser-full-2015-04-20/stanford-parser-models.jar')
10
11 result = list(scp.raw_parse(sentence))
12 print(result[0])
```

nlp_strategy_16.py hosted with ❤ by GitHub

[view raw](#)

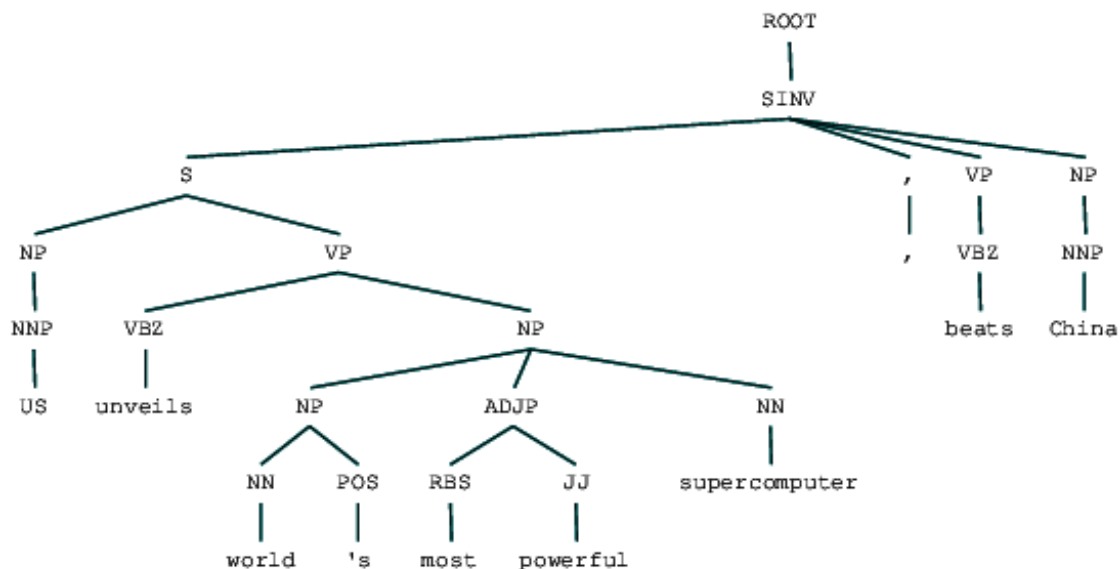
```
(ROOT
  (SINV
    (S
      (NP (NNP US))
      (VP
        (VBZ unveils)
```



```
(NP
  (NP (NN world) (POS 's))
  (ADJP (RBS most) (JJ powerful))
  (NN supercomputer)))
(, ,)
(VP (VBZ beats))
(NP (NNP China)))
```

We can see the constituency parse tree for our news headline. Let's visualize it to understand the structure better.

```
from IPython.display import display
display(result[0])
```



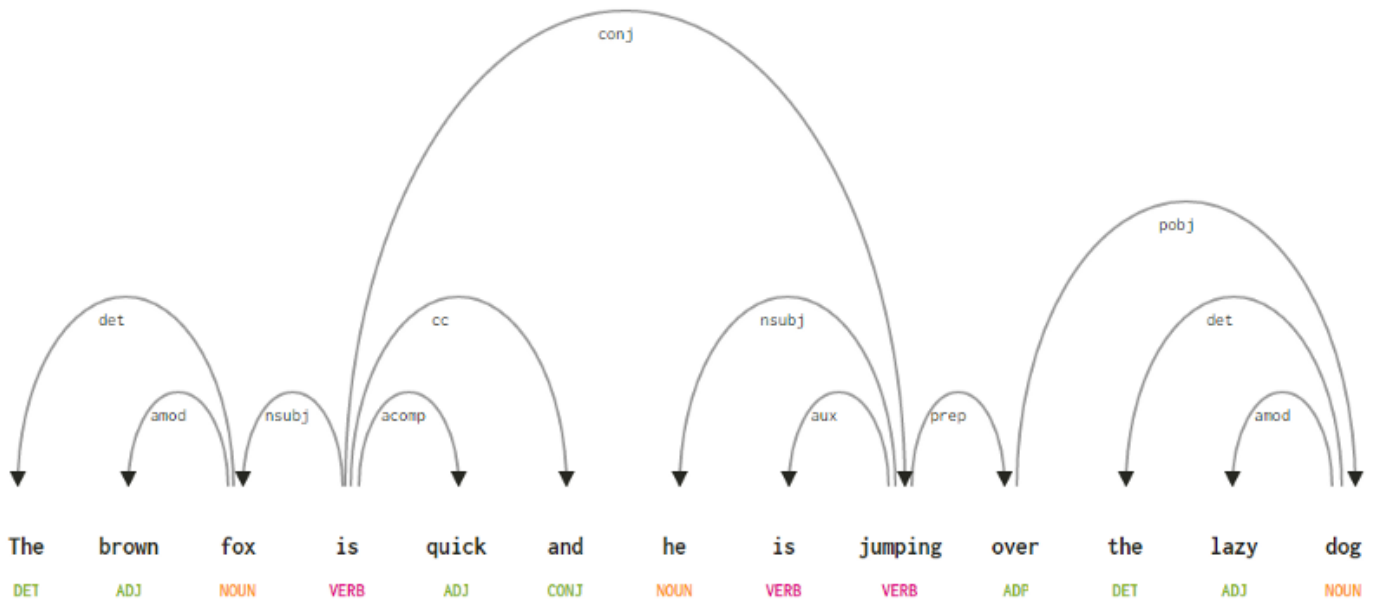
Constituency parsed news headline

We can see the nested hierarchical structure of the constituents in the preceding output as compared to the flat structure in shallow parsing. In case you are wondering what *SINV* means, it represents an *Inverted declarative sentence*, i.e. one in which the subject follows the tensed verb or modal. Refer to the [Penn Treebank reference](#) as needed to lookup other tags.

Dependency Parsing

In dependency parsing, we try to use dependency-based grammars to analyze and infer *both structure and semantic dependencies* and relationships between tokens in a sentence. The basic principle behind a dependency grammar is that in any sentence in the language, all words except one, have some relationship or dependency on other words in the sentence. The word that has no dependency is called the root of the sentence. The verb is taken as the root of the sentence in most cases. All the other words are directly or indirectly linked to the root verb using links, which are the dependencies.

Considering our sentence *“The brown fox is quick and he is jumping over the lazy dog”*, if we wanted to draw the dependency syntax tree for this, we would have the structure



A dependency parse tree for a sentence

These dependency relationships each have their own meaning and are a part of a list of universal dependency types. This is discussed in an original paper, [Universal Stanford Dependencies: A Cross-Linguistic Typology by de Marneffe et al, 2014](#)). You can check out the exhaustive list of dependency types and their meanings [here](#).

If we observe some of these dependencies, it is not too hard to understand them.

- The dependency tag **det** is pretty intuitive—it denotes the determiner relationship between a nominal head and the determiner. Usually, the word with POS tag **DET** will also have the **det** dependency tag relation. Examples include **fox** → **the** and **dog** → **the**.
- The dependency tag **amod** stands for adjectival modifier and stands for any adjective that modifies the meaning of a noun. Examples include **fox** → **brown** and **dog** → **lazy**.
- The dependency tag **nsubj** stands for an entity that acts as a subject or agent in a clause. Examples include **is** → **fox** and **jumping** → **he**.
- The dependencies **cc** and **conj** have more to do with linkages related to words connected by coordinating conjunctions. Examples include **is** → **and** and **is** → **jumping**.
- The dependency tag **aux** indicates the auxiliary or secondary verb in the clause. Example: **jumping** → **is**.
- The dependency tag **acom** stands for adjective complement and acts as the complement or object to a verb in the sentence. Example: **is** → **quick**
- The dependency tag **prep** denotes a prepositional modifier, which usually modifies the meaning of a noun, verb, adjective, or preposition. Usually, this representation is used for prepositions having a noun or noun phrase complement. Example: **jumping** → **over**.
- The dependency tag **pobj** is used to denote the object of a preposition. This is usually the head of a noun phrase following a preposition in the sentence. Example: **over** → **dog**.

Spacy had two types of English dependency parsers based on what language models you use, you can find more details [here](#). Based on language models, you can use the [Universal Dependencies Scheme](#) or the [CLEAR Style Dependency Scheme](#) also available in [NLP4J](#) now. We will now leverage **spacy** and print out the dependencies for each token in our news headline.

```
1 dependency_pattern = '{left}<---{word}[{w_type}]--->{right}\n-----'
2 for token in sentence_nlp:
3     print(dependency_pattern.format(word=token.orth_,
4                                     w_type=token.dep_,
5                                     left=[t.orth_
6                                         for t
```

```

7             in token.lefts],
8         right=[t.orth_
9             for t
10             in token.rights]))

```

nlp_strategy_17.py hosted with ❤ by GitHub

[view raw](#)

```

[ ]<---US[compound]--->[ ]
-----
[ 'US' ]<---unveils[nsubj]--->[ 'supercomputer', ', ' ]
-----
[ ]<---world[poss]--->[ " 's " ]
-----
[ ]<--- 's[case]--->[ ]
-----
[ ]<---most[amod]--->[ ]
-----
[ ]<---powerful[compound]--->[ ]
-----
[ 'world', 'most', 'powerful' ]<---supercomputer[appos]--->[ ]
-----
[ ]<---,[punct]--->[ ]
-----
[ 'unveils' ]<---beats[ROOT]--->[ 'China' ]
-----
[ ]<---China[doobj]--->[ ]
-----

```

It is evident that the verb beats is the ROOT since it doesn't have any other dependencies as compared to the other tokens. For knowing more about each annotation you can always refer to the [CLEAR dependency scheme](#). We can also visualize the above dependencies in a better way.

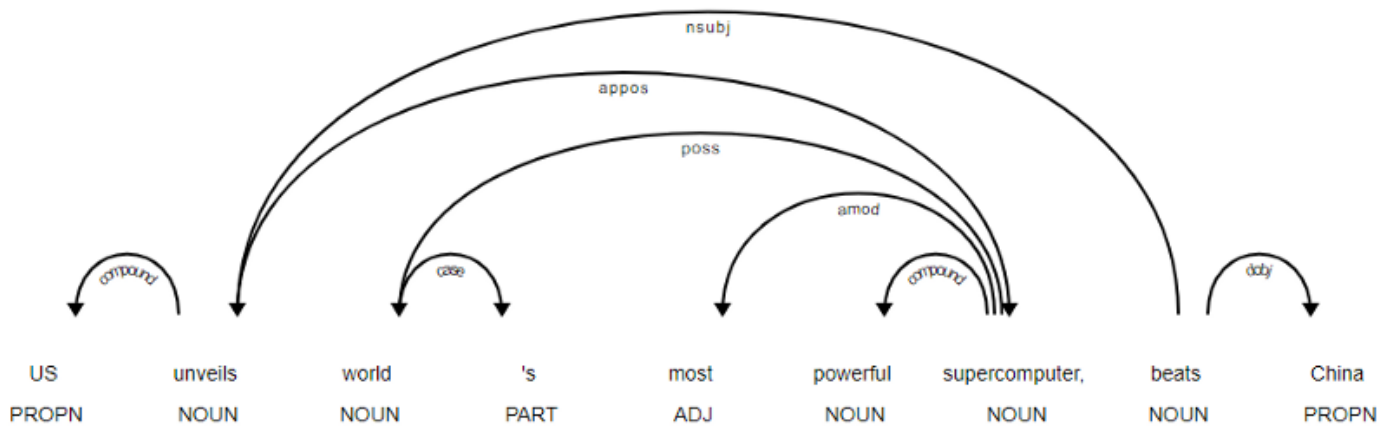
```

1  from spacy import displacy
2
3  displacy.render(sentence_nlp, jupyter=True,
4                  options={'distance': 110,
5                          'arrow_stroke': 2,
6                          'arrow_width': 8})

```

nlp_strategy_18.py hosted with ❤ by GitHub

[view raw](#)



News Headline dependency tree from SpaCy

You can also leverage **nltk** and the **StanfordDependencyParser** to visualize and build out the dependency tree. We showcase the dependency tree both in its raw and annotated form as follows.

```

1  from nltk.parse.stanford import StanfordDependencyParser
2  sdp = StanfordDependencyParser(path_to_jar='E:/stanford/stanford-parser-full-2015-04-20/stanford-par
3                                  path_to_models_jar='E:/stanford/stanford-parser-full-2015-04-20/stanf
4
5  result = list(sdp.raw_parse(sentence))
6
7  # print the dependency tree
8  dep_tree = [parse.tree() for parse in result][0]
9  print(dep_tree)
10
11 # visualize raw dependency tree
12 from IPython.display import display
13 display(dep_tree)
14
15 # visualize annotated dependency tree (needs graphviz)
16 from graphviz import Source
17 dep_tree_dot_repr = [parse for parse in result][0].to_dot()
18 source = Source(dep_tree_dot_repr, filename="dep_tree", format="png")
19 source

```

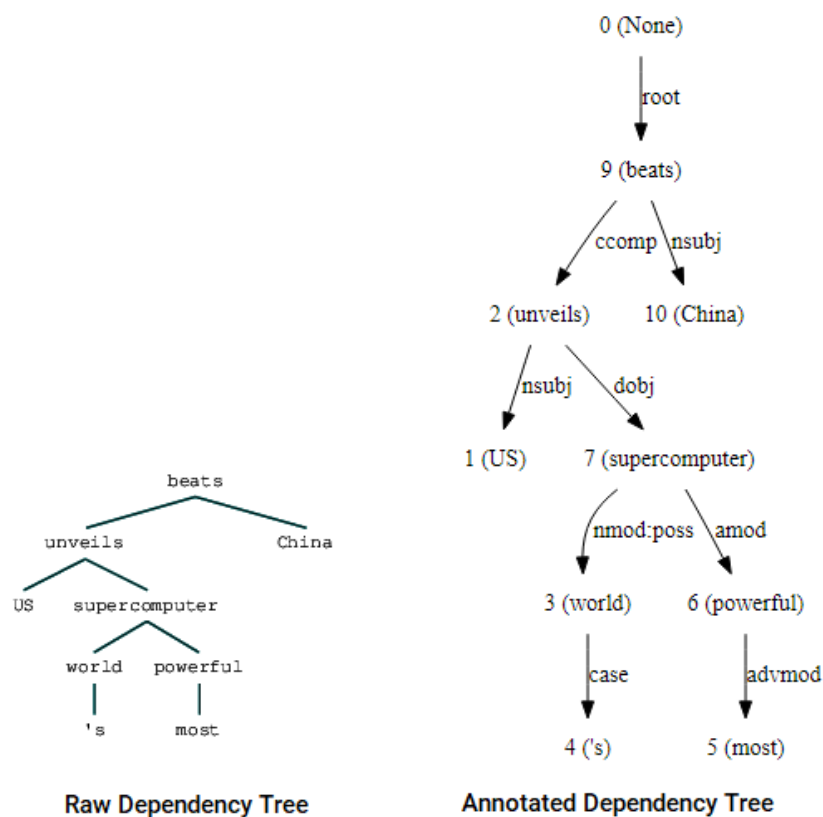
nlp_strategy_19.py hosted with ❤ by GitHub

[view raw](#)

```

(beats (unveils US (supercomputer (world 's) (powerful most)))
China)

```



Dependency Tree visualizations using nltk's Stanford dependency parser

You can notice the similarities with the tree we had obtained earlier. The annotations help with understanding the type of dependency among the different tokens.

Bio: [Dipanjan Sarkar](#) is a Data Scientist @Intel, an author, a mentor @Springboard, a writer, and a sports and sitcom addict.

[Original](#). Reposted with permission.

Related:

- [Robust Word2Vec Models with Gensim & Applying Word2Vec Features for Machine Learning Tasks](#)
- [Human Interpretable Machine Learning \(Part 1\)—The Need and Importance of Model Interpretation](#)
- [Implementing Deep Learning Methods and Feature Engineering for Text Data: The Skip-gram Model](#)