Dependency Syntax

Jonathan May

November 8, 2024

1 Why dependencies?

It turns out *lexicalization* is pretty important to syntactic parsing. This is at first a bit surprising, since syntax is concerned with the order of words and not their content. The famous sentence 'Colorless green ideas sleep furiously' (Chomsky 1957) is an example of a syntactically valid but semantically vacant sentence. It clearly has a parse:

```
(S
  (NP
    (JJ colorless)
    (JJ green)
    (NN ideas))
  (VP
    (VBP sleep)
    (RB furiously)))
```

But as we previously saw, rules like `S -> NP VP` without lexicalization can lead to lots of apparent ambiguity.

Another problem is that not all syntactic behavior occurs in contiguous phrases. This is particularly true in languages with *free word order* but even occurs in English; e.g. 'The hearing is scheduled on the issue today.' There is a relationship between 'on the issue' and 'the hearing' but that relationship isn't really expressible only through contiguous phrases.

Finally, as sentence length grows, a significant part of the parse is far from the word level, and this, it turns out, is less helpful in downstream tasks. We mostly want to know the relationship between words in a sentence. Specifically, the connection between phrases and the *heads* of subphrases contained within them is important.

What is a head? This is actually a tricky linguistic question. Informally it's:

- The most important word in a phrase
- The main content word in a phrase
- The word that determines the phrase's label
- The word that other phrases have to agree with (e.g. in case or gender)

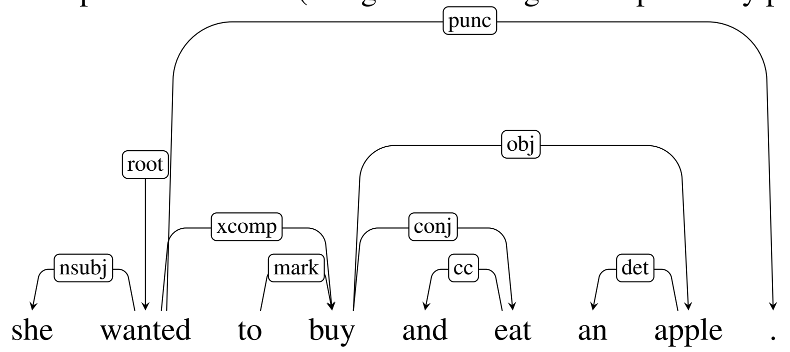
This can sometimes lead to inconsistencies. In the prepositional phrase ‘with the mayor’ is ‘with’ the head (determines the label) or is ‘mayor’ (the main content word)? It depends on which linguist you ask; the rules behind the Penn treebank say the preposition, while the rules behind universal dependencies say the noun. As long as you use consistent head finding you will be okay.

2 What are dependencies?

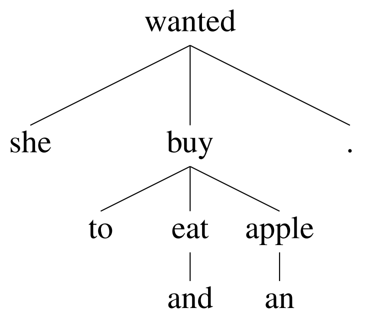
Here is an example for the sentence ‘She wanted to buy and eat an apple’ (from the UD guidelines):

1	she	2	nsubj
2	wanted	0	root
3	to	4	mark
4	buy	2	xcomp
5	and	6	cc
6	eat	4	conj
7	an	8	det
8	apple	4	obj
9	.	2	punct

Pictorially we can represent it like so (using the amazing tikz-dependency package):



Or this:

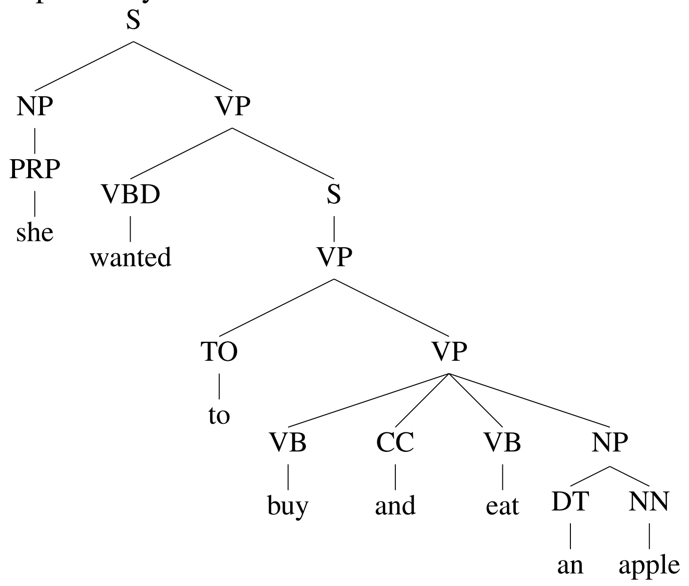


- Every word in a sentence except one has only one parent (or governor) word, which is the head of the smallest syntactic unit it is not the head of. A word may have zero or more words that consider it their head.
- The word which has no head is the *root* of the sentence.
- Each parent-child relationship may be annotated with a label connoting the role of the phrase the child is the head of. There are 37 such label types in universal dependencies.

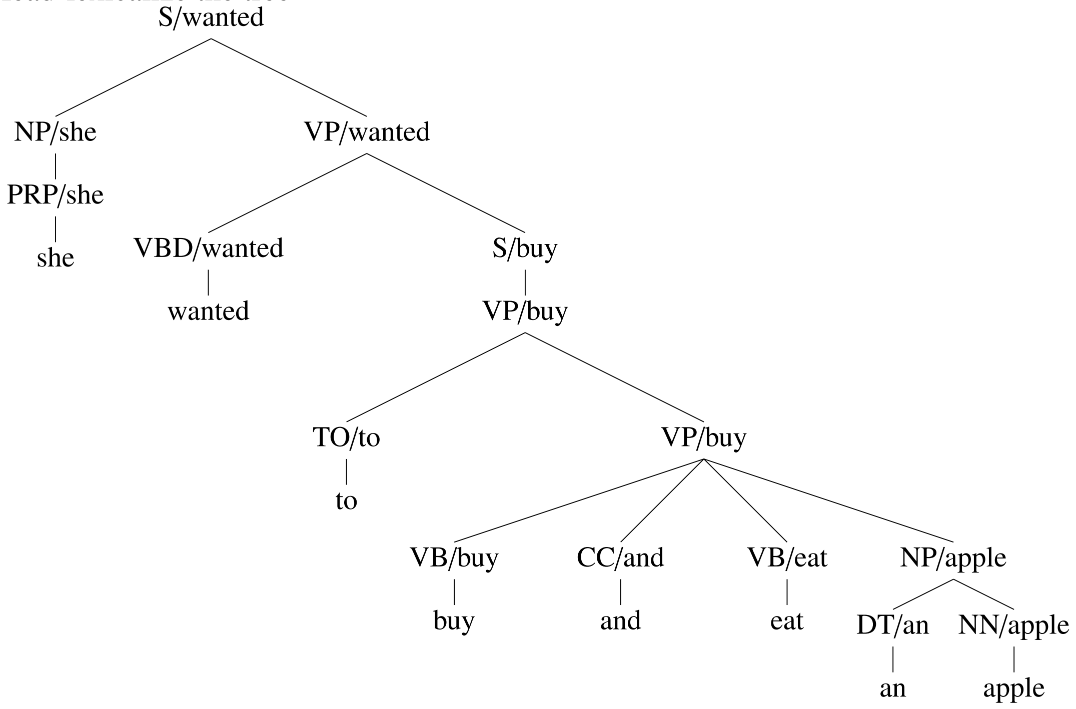
The consequence of these requirements (particularly the first two) is that this will form a tree. There is an extension to this formalism called *enhanced dependencies* that annotates more relationships and forms *graphs* (i.e. words can have more than one head). We won't discuss them here.

3 Conversion from Constituencies, planarity, and projectivity

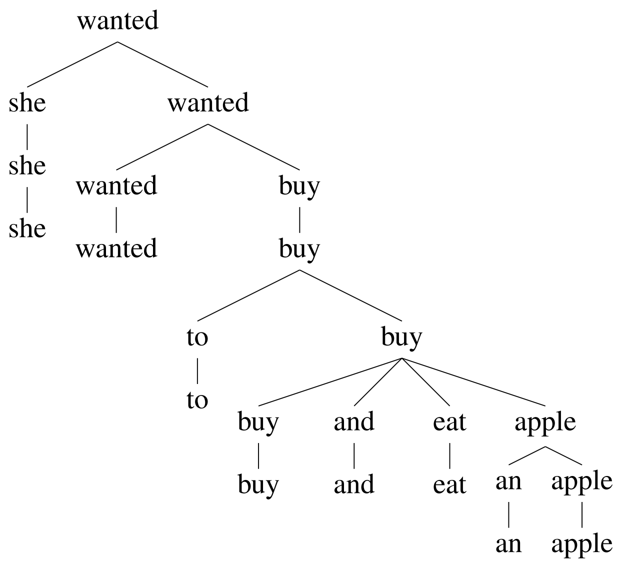
What is the relationship between constituencies and dependencies? You can convert a constituent tree into a (unlabeled) dependency tree as follows:



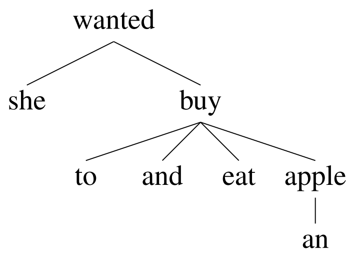
1. Head-lexicalize the tree



2. remove phrase labels

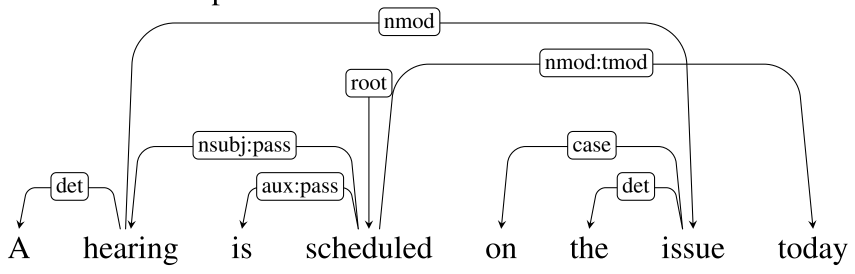


3. merge children into parents with same label



Notice it's not the same dependency as above! This is because the original example was annotated as a dependency, and this is converted from another structure, with different annotation standards and possibly different head rules.

Converting from dependencies to constituencies is in general not possible. Apart from the labeling problem (which also exists in the other direction) there is too much ambiguity introduced in the simpler dependency structure. More importantly, dependencies cannot be converted at all if they are not *planar*[1], i.e. if the arcs cross when the words are arranged in order. Such a tree is *non-projective.* Here is an example:



Thankfully non-projective dependencies are pretty rare in English.

[1]misleading to those very familiar with graph theory; see Kuhlman 1998

4 Parsing Methods

It turns out that parsing methods for dependencies are often a lot faster than those for constituencies. The one we will discuss, *shift-reduce*, is linear-time and greedy (though it can be beamed) and can take a wide variety of features. It doesn't handle non-projectivity, however. The second one, *Chiu-Liu-Edmonds*, is quadratic and optimal but somewhat limited in its feature set. We won't discuss it in detail but I provide several pointers.

4.1 Shift-Reduce

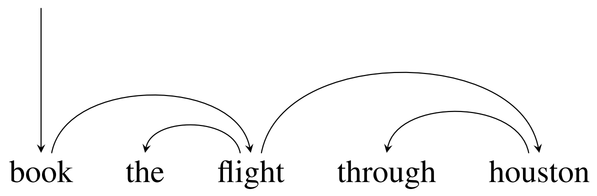
Big idea of shift-reduce parsing: you keep a *stack* of words/partial structures you are processing and a *buffer* of words you haven't started processing yet. At each time step you do some work (an *operation*) at the top of the stack. In 'arc-standard' parsing there are the following operations:

- SHIFT = move a word from the buffer to the top of the stack; $\alpha w|x\beta \Rightarrow \alpha wx|\beta$
- LEFT-ARC-*label* = top of stack is parent of second in stack; add *label*; top of stack stays in (pop 2nd); $\alpha wx|\beta \Rightarrow \alpha x|\beta; w \leftarrow x$
- RIGHT-ARC-*label* = second in stack is parent of top in stack; add *label*; second in stack stays in (pop top); $\alpha wx|\beta \Rightarrow \alpha w|\beta; w \rightarrow x$

So this becomes a classification problem with $1+2\times$labels choices. We'll get into what makes good features for this but it's first helpful to walk through a parse. Additionally, in order to train a classifier you need a lot of examples of a configuration and a choice of label. There is a general procedure for converting from a dependency tree into the sequence of parse steps that will form it. Using the tree as a guide:

- If `stack[0]` is the parent of `stack[1]` with label *l*, LEFT-ARC-*l*.
- If `stack[1]` is the parent of `stack[0]` with label *l* and no dependents of `stack[0]` are still in the buffer, RIGHT-ARC-*l*.
- Otherwise, SHIFT

Here is an example parse tree (unlabeled) and a walkthrough; we'll go over it in class:



ROOT	book the flight through houston	SHIFT	
ROOT book	the flight through houston	SHIFT	
ROOT book the	flight through houston	SHIFT	
ROOT book the flight	through houston	LEFT	the ← flight
ROOT book flight	through houston	SHIFT	
ROOT book flight through	houston	SHIFT	
ROOT book flight through houston		LEFT	through ← houston
ROOT book flight houston		RIGHT	flight → houston
ROOT book flight		RIGHT	book → flight
ROOT book		RIGHT	ROOT → book
ROOT		Done	

A potential problem is that shift-reduce is *greedy* and an early bad decision can lead to later problems. We can beam, i.e. consider k possibilities simultaneously. We then consider the k best successors of these, trim back to only k, and continue. This is still linear in sentence length, i.e. nk^2.

Another problem is that arc-standard is strictly 'bottom-up' in that it is cautious, particularly about RIGHT, e.g. waiting a long time after seeing the initial 'book flight' to make that arc. The useful features could be inaccessible to a classifier. A variant, called 'arc-eager' seeks to improve things. It has slightly different definitions and one more operation:

- SHIFT = (as before) move a word from the buffer to the top of the stack; $\alpha w|x\beta \Rightarrow \alpha w x|\beta$
- LEFT-ARC-*label* = top of *buffer* is parent of top in stack; add *label*; top of stack is popped; $\alpha w|x\beta \Rightarrow \alpha|x\beta; w \leftarrow x$
- RIGHT-ARC-*label* = top of stack is parent of top in buffer; add *label*; shift buffer to stack; $\alpha w|x\beta \Rightarrow \alpha w x|\beta; w \rightarrow x$
- REDUCE = pop the top of the stack; $\alpha w|x\beta \Rightarrow \alpha|x\beta$

The heuristics for converting from a tree to an instruction are;

- if the top of the buffer is parent to the top of the stack, do LEFT;
- if the top of the stack is parent to the top of the buffer, do RIGHT;
- if the top of the stack has an assigned parent and no unassigned children, REDUCE
- if it can be done, SHIFT

Here's how the parse goes under arc-eager:

ROOT	book the flight through houston	RIGHT	ROOT → book
ROOT book	the flight through houston	SHIFT	
ROOT book the	flight through houston	LEFT	the ← flight
ROOT book	flight through houston	RIGHT	book → flight
ROOT book flight	through houston	SHIFT	
ROOT book flight through	houston	LEFT	through ← houston
ROOT book flight	houston	RIGHT	flight → houston
ROOT book flight houston		REDUCE	
ROOT book flight		REDUCE	
ROOT book		REDUCE	
ROOT		Done	

4.1.1 Neural Dependency Parser

An excellent application of neural networks to dependency parsing is the work of Danqi Chen (student of Manning, now professor at Princeton) from 2014 (ancient history!). It's pretty straightforward and still uses hand-engineered features; the trick is that it uses a lot of them, and the neural network takes care of the smoothing. The features are:

- the first three words on the stack and the buffer (and their POS tags) (12 features)
- the words, POS tags, and arc labels of the first and second leftmost and rightmost children of the first two words on the stack, (24 features)
- the words, POS tags, and arc labels of leftmost child of the leftmost child and rightmost child of rightmost child of the first two words of the stack (12 features)

Collobert et al. (2011) pretrained word embeddings were used; the rest were learned. A n^3 activation function is used (quite unusual nowadays but Chen was writing her code all by hand). The parser scored 90.7 LAS and 92.0 UAS on the converted treebank, a SOTA for the time. It was also much faster than other parsers; this was in large part due to a lot of precomputation of values. The latest (2019) using BERT/XLNet is around 95.7 LAS and 97.2 UAS.

4.2 Chiu-Liu-Edmonds

Shift-reduce parsers have a major flaw; they can't handle non-projective trees. For English this isn't a problem; over 99.4% of the English (and 100% of the Chinese) Treebank is projective. But for Czech this would be a problem. Another algorithm, called Chiu-Liu-Edmonds, after its simultaneous creators, can be used instead. It's a fairly elegant algorithm but unless there's great demand I'll leave it to you to research (e.g. `https://www.cs.cmu.edu/~sswayamd/talks/cle.pdf`, `https://user.phil.hhu.de/~waszczuk/teaching/depparse-su18/exercises/session_5/example.pdf`)