Computational Syntax: Lecture Notes

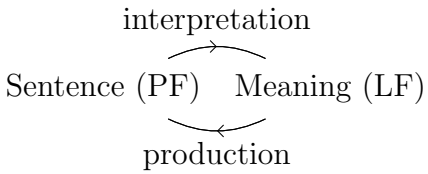
2019-08-26

1 About the Course

The syllabus and schedule are available at https://lin628.thomasgraf.net and readings will be available at https://lin628.thomasgraf.net/readings. The latter is a private GitHub repository. In order to get access you must email your GitHub username to Thomas. All email pertaining to this course (including that one) should be sent to lin628@thomasgraf.net.

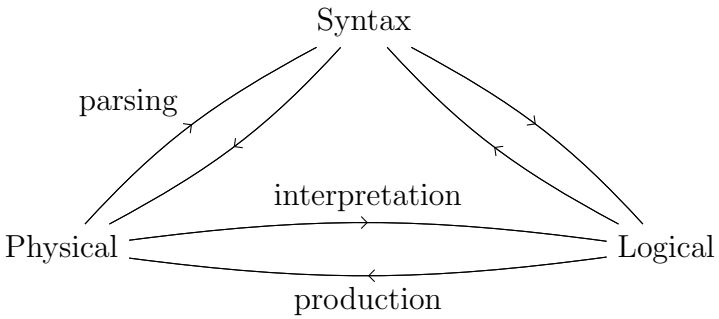
2 What Syntax Is

The average person is not concerned with word order, agreement, or other formal notions. Instead, the focus lies on extracting meaning from a sentence (*interpretation*) or creating a sentence that expresses an idea (*production*).

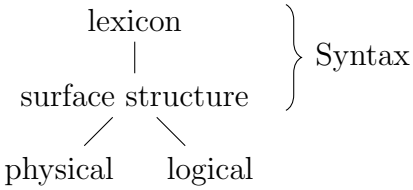


The sentences we see and hear are known as the *physical form* (PF), and their meaning the *logical form* (LF). We can define this as two mappings: *interpretation* : $PF \rightarrow LF$ and *production* : $LF \rightarrow PF$. These kinds of mappings exist all over the place, in and out of linguistics, but in our case it isn't really possible to define the exact set of possible sentences in a language, nor the set of possible meanings associated with those sentences. This becomes further complicated if we try to directly define what those mappings are, themselves. So instead of trying to figure out what that direct mapping is, it turns out that mapping back and forth between two domains is easier when the mapping is done through some third "mediator" domain, called an *interpolant*. For us, syntax is that interpolant, and its usefulness arises from the mathematical fact of *bimorphism*.

We can say that we have a couple more mappings to go both ways through this "mediator," namely *parsing* : *syntactic object* $\leftrightarrow$ *PF* and *???* : *syntactic object* $\leftrightarrow$ *LF*. (There doesn't seem to be a term for the mapping between syntax and meaning, since the syntax-semantics interface is fuzzy at best.)



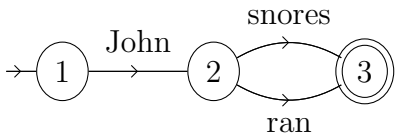
Sometimes syntax is split into two components, lexicon and surface structure, which then makes this diagram took like an inverted T or Y.



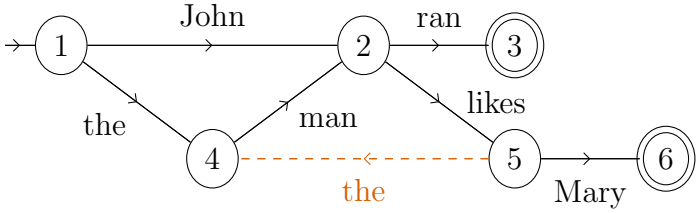
No matter how you look at it, the most important concept is that syntax sits between sentences and meaning and facilitates mapping between the two.

3 Why Trees?

Sentences are generally analyzed using trees, but does it have to be that way? One might try to use a finite-state automaton to represent a sentence:



This describes the two sentences "John snores." and "John ran." Supporting more complex sentences is entirely possible:



Note that the dashed reddish edge cannot be included to incorporate "the man" as a possible object for "likes" for a few reasons:

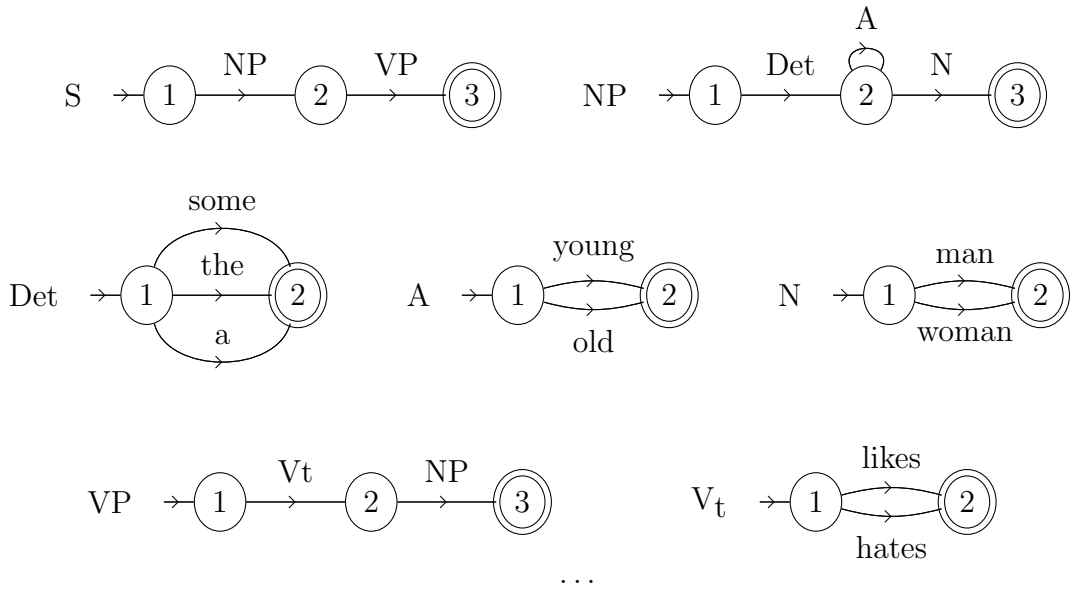
- State (2) isn't accepting, so "John likes the man." is not admitted. But if it were, "John." would be. Perhaps the latter is fine.

- On the other hand, ② can be followed by a verb, admitting "John likes the man ran." This is not fine.
- And, though it doesn't show up in this case, subject and object positions might follow different rules, such as the distinction between "they" and "them".

For these reasons, the structures in this automaton cannot really be shared, which means that in general there will be a lot of redundant machinery in the automaton. In fact, there are two main issues with this approach (and a couple other related ones):

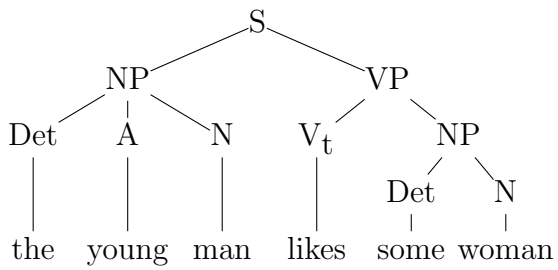
1. This redundancy within the automaton and the corresponding large description size.
2. The lack of generalization: for example there is no indication that subject noun-phrases resemble object noun-phrases at all.
3. Allowing things like arbitrarily nested relative clauses prevents a finite-state representation (though in practice this is irrelevant).
4. There is no clear way to map from this automaton to a meaning, one of the primary goals of syntax. The meaning of an ambiguous phrase like "John and Bill or Mary" remains ambiguous.

So let us solve the redundancy problem by constructing modular automata.



...

This is a *recursive transition network*. Replacing the edges labelled by nonterminals with the associated automata will (except when an X contains an X) produce some finite-state automaton, but this factoring has resulted in a non-redundant description with reusable components. Ultimately, it turns out that factoring in this way gives us a tree structure. The computational trace of these modular machines working through a sentence reveals that tree structures are naturally inherent in syntax.



That means that trees naturally arise in trying to describe these sets without redundant information in a way that allows generalizations.

It is of note that once we have factored the automata out like this, it is only the case that this will compile out to an automaton with finitely many states if there is a maximum recursion depth imposed on the levels of embedding in the automaton. Failure to do this will result in a structure that is more complex than a finite-state automaton, but the description of the grammar remains fundamentally the same. Regardless of whether or not there is a numerical limit on how many times we can embed a phrasal automaton, we are still describing the language in terms of factored machines. From this perspective, the competence-performance split (what theory allows vs. what people can actually do) becomes irrelevant. Those who believe describing the language user's performance takes priority will insist that we do have these hard limits on recursive structures, as opposed to upholders of the competence model, who may say that such numbers may be there but we may as well ignore them since we only care about the description of a general representation. Either of these may well be the case, but it doesn't matter in the end. We do not factorize these machines because we think that humans are capable of unbounded center embedding; rather, we do it because we do not otherwise get compact representations that uncover insightful generalizations.

Another way to represent the same information as these automata modules is a *context-free grammar* (CFG), which linguists are already familiar with in the form of *phrase structure grammars*.

$$\begin{aligned} \text{S} &\rightarrow \text{NP VP} \\ \text{NP} &\rightarrow \text{Det (A) N} \\ \text{Det} &\rightarrow \text{some} \mid \text{the} \mid \text{a} \\ &\ldots \end{aligned}$$

4 Improving Modular Automata

One can see how the grammar at the end of the previous section directly relates to the automata from which it was derived. But note that there are many things that this grammar doesn't capture: subject-verb agreement, case, the distinction between "a" and "an", reflexive licensing, NPI licensing, and others. For case in particular, we should require an object-NP inside a VP, but then there is again no guarantee that object-NP acts like a subject-NP in any way.

We can get around this issue by using *attribute value matrices* (AVMs) to collapse similar categories. Each chunk of information we want to track is an attribute, and we can use that

to carry information and enforce agreement. Consider the following rewrite rule:

$$[\mathbf{S}] \rightarrow \begin{bmatrix} \mathbf{NP} \\ \textsc{Num}: & \boxed{\alpha} \\ \textsc{Person}: & \boxed{\beta} \\ \textsc{NegPol}: & \boxed{\gamma} \end{bmatrix} \begin{bmatrix} \mathbf{VP} \\ \textsc{Num}: & \boxed{\alpha} \\ \textsc{Person}: & \boxed{\beta} \\ \textsc{NegPol}: & \boxed{\gamma} \end{bmatrix}$$

This describes only the cases where the (boxed) variables agree. Now consider this:

$$\begin{bmatrix} \mathbf{VP} \\ \textsc{Num}: & \boxed{\alpha} \\ \textsc{Person}: & \boxed{\beta} \\ \textsc{NegPol}: & \boxed{\gamma} \end{bmatrix} \rightarrow \begin{bmatrix} \mathbf{V} \\ \textsc{Val}: & 1 \\ \textsc{Num}: & \boxed{\alpha} \\ \textsc{Person}: & \boxed{\beta} \\ \textsc{NegPol}: & \boxed{\gamma} \end{bmatrix} \begin{bmatrix} \mathbf{NP} \\ \textsc{Case}: & \text{object} \\ \textsc{NegPol}: & \boxed{\gamma} \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{VP} \\ \textsc{Num}: & \boxed{\alpha} \\ \textsc{Person}: & \boxed{\beta} \\ \textsc{NegPol}: & \boxed{\gamma} \end{bmatrix} \rightarrow \begin{bmatrix} \mathbf{V} \\ \textsc{Val}: & 0 \\ \textsc{Num}: & \boxed{\alpha} \\ \textsc{Person}: & \boxed{\beta} \\ \textsc{NegPol}: & \boxed{\gamma} \end{bmatrix}$$

Here we specify that what follows a transitive verb is an object-type noun phrase. Other features on that noun phrase are irrelevant and left simply unspecified. This AVM is *under-specified*, but still valid.

All of these cases can still be compiled into a CFG with finitely many rewrite rules. You can still define arbitrary variations between what should be similar constructions like subject and object noun phrases, but considerations of grammar size will deter this. A larger grammar has more choices and makes it harder to determine the correct parse of a given sentence.

GPSG uses these AVMs to encode information about long-distance dependencies like movement with a Slash feature that denotes a missing dependency. FIN