1. Defining context-free grammars: this is just for reference

**Definition.** A context-free grammar $G$ is a four-tuple, $G = (N, V, R, S)$ where:

- $N$ is a finite set of nonterminals (that is, all the phrase names, like $NP$)
- $V$ is a finite set of terminals (all the Vocabulary items, words, e.g., $Romney$, $dog$)
- $R$ is a finite set of rules in the form: $X \rightarrow Y_1...Y_n$, where $X \in N$ is a single nonterminal, and $Y_i \in N \cup V$ for each $Y_i$, $n \geq 0$ (so the $Y_i$ comprise any string of terminals and nonterminals. We will include a special ‘empty symbol’ epsilon to denote the empty string. The key point is that there is just one nonterminal on the left-hand side of any context-free rule.)
- $S$ is a special designated ‘Start’ symbol (we will usually just call it ‘Start’)

**Motivation.** First, a review as to why we are using grammars, what parsing is for, and what you should learn from this. Most broadly, you will learn how to build state-of-the-art statistically based parsers for language, but also learn to appreciate their strengths and weaknesses. Most of these systems use as a front end some kind of statistical part of speech tagger – which you have learned about in the last lecture. But what about the next step past part of speech tagging? Why do we want to be able to parse sentences at all?

Parsing uncovers the hidden structure of linguistic input. In many applications involving natural language, the underlying predicate-argument structure of sentences can be useful: this describes ‘who did what to whom’ and is sometimes called “thematic structure.” For example, in the sentence, “John ate the spider”, we can identify the main predicate or ‘action’ of the sentence as eating, and further that John is the person who did the eating, and it was the spider that got eaten. Here, John and the spider serve as arguments to a predicate like those of a programming language function call: $\text{eat}(John, \text{spider})$.

We shall examine the ‘semantics’ of such representations later in the course, but here we will focus on the first step: figuring out the right syntactic analysis given the words of the sentence. (The translation to what a sentence means can be very much like the analogous operations in a programming language: first one checks that a program’s syntax can be parsed into the right form; and then one attaches bits of procedural ‘operations’ – the ‘meanings’ of program pieces – to the syntactic parts of the program.) Importantly, if the syntax of the sentence is different, we can get a very different meaning: the spider ate John has a very different impact on John!

Thus it is very important to be able to parse sentences correctly. The syntactic analysis of language provides a means to explicitly discover the various predicate-argument dependencies that may exist in a sentence. In natural language processing, the syntactic analysis of natural language input can vary from being very low-level, such as simply tagging each word in the sentence with a part of speech, or very high level, such as recovering a structural analysis that identifies the dependency between each predicate in the sentence and its explicit and implicit arguments. The major bottleneck in parsing natural language is the fact that ambiguity is so pervasive. In syntactic parsing, ambiguity is a particularly difficult problem since the most plausible analysis has to be chosen from an exponentially large number of alternative analyses. From tagging to full parsing, algorithms have to be carefully chosen that can handle such ambiguity.
Another motivation for parsing comes from the natural language task of summarization, in which several documents about the same topic should be condensed down to a small digest of information typically limited in size to a 100 or 250 words. Such a summary may be in response to a question that is answered (perhaps in different ways) in the set of documents. In this case, a useful sub-task is to compress an individual sentence so that only the relevant portions of a sentence are included in the summary. This allows the summary to be concise, informative and fluent. For example, we may want to compress sentence 1 to a shorter sentence 2.

1. Beyond the basic level, the operations of the three products vary widely.

2. The operations of the products vary.

One way to approach this task is to first parse the sentence in order to find the various constituents: where we recursively partition the words in the sentence into individual phrases such as a verb phrase, VP or a noun phrase, NP. The output of the parser for the input sentence 1 is shown in Figure 1. The parse tree produced by the parser can now be edited using a compression model that is aware of constituents and few choice constituent deletions can produce a fluent compressed version of the original sentence.

The way this works is by deleting whole subtrees in the full sentence that do not seem to play a crucial role in the predicate-argument structure. But this presumes that we can build a proper hierarchical structure like this in the first place. That is what parsing is all about, in particular, parsing with context-free grammars.

Another example is the paraphrasing of text. In the sentence fragment 3 below, the capitalized phrase ‘EUROPEAN COUNTRIES’ can be replaced with other phrases without changing the essential meaning of the sentence. A few examples of replacement phrases are shown in italics in sentence fragments 4 to 8. This kind of replacement cannot simply rely on substitution of arbitrary words in the sentence since such an approach can lead to incoherent and badly formed paraphrases. Paraphrasing models build on top on parsers to identify target constituents to replace, and also to find appropriate replacement phrases that can substitute for the original phrase. Paraphrases of this type have been shown to be useful in applications such as statistical machine translation.

3. open borders imply increasing racial fragmentation in EUROPEAN COUNTRIES.

4. open borders imply increasing racial fragmentation in the countries of Europe.

5. open borders imply increasing racial fragmentation in European states.

6. open borders imply increasing racial fragmentation in Europe.
7. Open borders imply increasing racial fragmentation in European nations.

8. Open borders imply increasing racial fragmentation in the European countries.

In contemporary natural language processing, syntactic parsers are routinely used in many applications including but not limited to: statistical machine translation, information extraction from text collections, language summarization, producing entity grids (company names, places, etc.) for language generation, error correction in text, knowledge acquisition from language (e.g. discovering semantic classes or “x IS-A kind of y” relationships), and so forth. Parsers have been written for a large number of languages around the world, and are an essential component in many kinds of multilingual processing tasks.

Parsing recovers information that is not explicit in the input sentence. This implies that a parser requires some knowledge in addition to the input sentence about the kind of syntactic analysis that should be produced as output. One method to provide such knowledge to the parser is to write down a grammar of the language—a set of rules of syntactic analysis. For instance, one might write down the rules of syntax as a context-free grammar (CFG)

The following CFG (written in a simple Backus-Naur form) represents a simple grammar of transitive verbs in English, verbs (V) that have a subject and object noun phrase (NP), plus modifiers of verb phrases (VP) in the form of prepositional phrases (PP).

\[
\begin{align*}
\text{Start} & \rightarrow S \\
S & \rightarrow \text{NP} \ \text{VP} \\
\text{NP} & \rightarrow \text{‘John’} \ | \ ‘\text{pockets’} \ | \ DT \ N \ N \ | \ NP \ PP \\
\text{VP} & \rightarrow \text{VB} \ \text{NP} \ | \ \text{VP} \ \text{PP} \\
\text{VB} & \rightarrow \text{‘bought’} \\
\text{DT} & \rightarrow \text{‘a’} \\
\text{NN} & \rightarrow \text{‘shirt’} \\
\text{PP} & \rightarrow \text{IN} \ \text{NP} \\
\text{IN} & \rightarrow \text{‘with’}
\end{align*}
\]

\[
\begin{align*}
(S & \ (NP \ John) \\
(VP & \ (VB \ bought) \\
(NP & \ (NP \ (DT \ a) \\
(N & \ (NN \ shirt)) \\
(PP & \ (IN \ with) \\
(NP & \ (NP \ pockets))))))
\end{align*}
\]

Natural language grammars typically have the words as terminal symbols in the CFG and they are generated by rules of type \(X \rightarrow w\) where \(X\) is the part of speech for the word \(w\). For example in the above CFG, the rule \(\text{VB} \rightarrow \ ‘\text{saw’}\) has the part of speech symbol \(\text{VB}\) generating the verb ‘saw’. Such non-terminals are called \textit{part-of-speech tags} or \textit{pre-terminals}. The above CFG can produce a syntax analysis of a sentence like ‘John bought a shirt with pockets’ with \textit{Start} as the start symbol of the grammar. Parsing the sentence with the CFG rules gives us two possible derivations for this sentence. In one parse, pockets are a kind of currency that can be used to buy a shirt, and the other parse, which is the more plausible one, John is purchasing a kind of shirt that has pockets.

\[
\begin{align*}
(\text{START} & \ (NP \ John) \\
(VP & \ (VB \ bought) \\
(NP & \ (NP \ (DT \ a) \\
(NN & \ (NN \ shirt)) \\
(PP & \ (IN \ with) \\
(NP & \ (NP \ pockets))))))
\end{align*}
\]
(START (S (NP John)
    (VP (VP (V bought)
        (NP (DT a)
            (NN shirt)))
        (PP (IN with)
            (NP pockets)))
    ((NP (DT a)
        (NN shirt)))
    (PP (IN with)
        (NP pockets)))

However, writing down a CFG for the syntactic analysis of natural language is problematic. Unlike a programming language, natural language is far too complex to simply list all the syntactic rules in terms of a CFG. A simple list of rules does not consider interactions between different components in the grammar. We could extend this grammar to include other types of verbs, and other syntactic constructions, but listing all possible syntactic constructions in a language is a difficult task; some examples are given below. In addition it is difficult to exhaustively list lexical properties of words, for instance, listing all the grammar rules in which a particular word can be a participant. This is a typical knowledge acquisition problem that we will explore more deeply as lexical semantics.

Apart from this knowledge acquisition problem, there is another less apparent problem: it turns out that the rules interact with each other in combinatorially explosive ways. Consider a simple CFG that provides a syntactic analysis of noun phrases as a binary branching tree:

NP → NP NP
NP → ‘natural’ | ‘language’ | ‘processing’ | ‘book’

Recursive rules produce ambiguity: with NP as the start symbol, for the input ‘natural’ there is one parse tree (N natural); for the input ‘natural language’ we use the recursive rule once and obtain one parse tree (NP (NP natural) (NP language)); for the input ‘natural language processing’ we use the recursive rule twice in each parse and there are two ambiguous parses:

\[
\begin{align*}
NP & \rightarrow NP \ NP \\
NP & \rightarrow natural \ | \ language \ | \ processing
\end{align*}
\]

Note that the ambiguity in the syntactic analysis reflects a real ambiguity: is it a processing of natural language, or is it a natural way to do language processing? So this issue cannot be resolved by changing the formalism in which the rules are written—e.g.by using finite-state automata that can be deterministic but cannot simultaneously model both meanings in a single grammar. Any system of writing down syntactic rules should represent this ambiguity.

However, by using the recursive rule 3 times we get 5 parses for ‘natural language processing book’. I have left one out; you might want to try to figure out which one is missing!
For longer and longer input noun phrases, using the recursive rule 4 times we get 14 parses; using it 5 times we get 42 parses; using it 6 times we get 132 parses. In fact, for CFGs it can be proved that the number of parses obtained by using the recursive rule \(n\) times is the Catalan number of \(n\):

\[
Cat(n) = \frac{1}{n+1} \binom{2n}{n}
\]

This is a fun exercise in recurrence equations if you want to try to prove this – Knuth has a nice section in his Art of Computer Programming textbooks on this.

The problem is that this occurs not only for coordinate structures such as the noun phrase grammar, but also when you have recursive rules to deal with modifiers such as the recursive rule for prepositional phrase modification \(VP \rightarrow VP \, PP\) in the first CFG we gave above. In fact, the ambiguity of PP modification is not independent of the ambiguity of coordination: in a sentence with both types of ambiguity, the total number of parses is the cross product of the parses from each sub-grammar. This poses a serious computational problem for parsers. For an input with \(n\) words the number of possible parses is exponential in \(n\).

Now, for most natural language tasks we do not wish to explore this entire space of ambiguity, even if, as we show later, it is possible to produce a compact representation of the entire exponential number of parses in polynomial time (for CFGs, the worst case time complexity is \(O(n^3)\) and store it in polynomial space (for CFGs, the space needed is \(n^2\)). For example, for the input ‘natural language processing book’ only one out of the five parses obtained using the CFG above is intuitively correct (corresponding to a book about the processing of natural language):

\[
(NP \, (NP \, (NP \, (NP \, natural) \, (NP \, language)) \, (NP \, processing)) \, (NP \, book))
\]

This is a second knowledge acquisition problem—not only do we need to know the syntactic rules for a particular language, but we also need to know which analysis is the most plausible for a given input.
sentence. The construction of a Treebank, like the Penn Treebank we have seen in class, is a data driven approach to syntax analysis allows us to address both of these knowledge acquisition bottlenecks in one stroke. We shall return to the use of the Penn Tree Bank in a bit, but let us pursue other questions about CFGs first.

From the previous lecture (and notes) we saw how to derive a left-most derivation from a CFG. There is another ‘canonical’ derivation order, right-most, which is valuable for parsing. For example, using the toy grammar from the last set of notes, we can derive the same sentence, the dog saw the man, in a different way, expanding the rightmost nonterminal at each step (highlighted in red here):

<table>
<thead>
<tr>
<th>Derivation</th>
<th>Rules used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>S → NP VP</td>
</tr>
<tr>
<td>NP VP</td>
<td>VP → Vt NP</td>
</tr>
<tr>
<td>NP Vt NP</td>
<td>NP → DT NN</td>
</tr>
<tr>
<td>NP Vt DT NN</td>
<td>NN → man</td>
</tr>
<tr>
<td>NP Vt DT man</td>
<td>DT → the</td>
</tr>
<tr>
<td>NP saw the man</td>
<td>Vt → saw</td>
</tr>
<tr>
<td>DT NN saw the man</td>
<td>NN → dog</td>
</tr>
<tr>
<td>DT dog saw the man</td>
<td>DT → the</td>
</tr>
<tr>
<td>the dog saw the man</td>
<td>STOP no more rules apply</td>
</tr>
</tbody>
</table>

If we go backwards through the derivation, starting from the words, one can see that the rightmost derivation order is a natural one for parsing: it collects together the words and then the red nonterminals in the proper order as we would go through the sentence left to right. For example, first the is found to have the part of speech DT (in fact, this would usually be done by a preprocessing step to get the POS tag), and then dog is found to be an NN. The third step from the bottom of the derivation shows that we can put DT and NN together to form an NP, since there is a rule that combines DT and NN to form an NP. This ‘bottom up’ order naturally corresponds to a right-most derivation, so you should keep it in mind when we discuss parsing algorithms.

As noted, it is very hard just to think about the possible verb phrase expansions to write a CFG for English (or any other language). We can see that the simple difference between intransitive verbs (verbs that take 0 arguments, like sleep), and transitive verbs (verbs that take just 1 argument, like hit) barely scratch the surface of the individual differences between verbs, what are called verb subcategories. (We ignore the Subject of the sentence, for reasons that we’ll come to later.) For example, there are at least these verb subcategories in English:

1. 0 arguments (sleep, walk, ...)

2. 1 argument (*eat, hit, devour ...*)  (But note that for *eat*, the argument is optional, but for *devour* it is not: John ate his dinner/John devoured his dinner vs. John ate & ??John devoured. Further, notice the verb *dine*: it takes 0 arguments, even though it is close in meaning to *eat*: John dined; ? John dined the meal.)

3. 2 arguments (*give, tell*) eg, I gave John a book/I gave a book to John. But some of these can’t occur in both of these forms: I donated a book to the library/??I donated the library a book

4. 3 arguments. Very few of these in English – perhaps just 5 or 6. Eg, I bet *John five dollars that he would lose*

5. More arguments? No! It appears that human languages don’t go beyond this. There’s actually a linguistic explanation for this, again one we’ll cover a few lectures later. But this shows that CFGs are **too powerful** in the sense that they let you describe rules that **never appear** in human language. We will see other ways CFGs are too powerful below.

There are many other verb constraints like these, each would need some separate CFG rule:

a. Pat relies on Kim
b. *Pat relies
c. The child put the toy on the table
d. *The child put the toy
e. The teacher became angry with the students
f. *The teacher became
g. The jury believed the witness lied

Further, certain verbs take only certain kinds of arguments, sometimes full sentences only of a certain type:

John wondered *why John left* but: ?John thought *why John left*

But there are still more distinctions in language, beyond verbs, to all other parts of speech. Consider these examples of Noun Phrases, where I have put asterisks before examples to indicate that they sound ‘odd’. (We don’t care why; it shows simply that English speakers *do* have these distinctions built into their heads:

a. The scared guy jumped from his seat
b. *The guy scared jumped from his seat
c. The guy scared of porcupines jumped from his seat
d. *The scared of porcupines guy jumped from his seat
e. The children are happy
f. The children are happy with the ice cream
g. The children are happy that they have ice cream
h. *The children are happy of ice cream
i. *The children are fond
j. *The children are fond with the ice cream
k. *The children are fond that they have ice cream
l. The children are fond of ice cream

This happens for adjectives too:

a. A magazine appeared on the newsstands
b. A magazine about crime appeared on the newsstands
c. Newsweek appeared on the newsstands
d. *Newsweek about crime appeared on the newsstands
e. The report surprised many people
f. The report that crime was declining surprised many people
g. The book surprised many people
h. *The book that crime was declining surprised many people

And also for Nouns:

a. A magazine appeared on the newsstands
b. A magazine about crime appeared on the newsstands
c. Newsweek appeared on the newsstands
d. *Newsweek about crime appeared on the newsstands
e. The report surprised many people
f. The report that crime was declining surprised many people
g. The book surprised many people
h. *The book that crime was declining surprised many people

And for Prepositions:

a. The storm arrived after the picnic
b. The storm arrived after we ate lunch
c. The storm arrived during the picnic
d. *The storm arrived during we ate lunch
e. *The storm arrived while the picnic
f. The storm arrived while we ate lunch

All of this knowledge has to somehow be put into the CFG rules! We will compare the hand-rule construction method (or rather, you will, in the CGW exercise), and then we will turn to how we can do this using Treebanks. While Treebank rule extraction has proved extremely valuable, it has its own weaknesses, as do CFGs.

We mentioned earlier that CFGs are too strong, in the sense that they let us write down impossible human language rules. In the next installment, we will describe several other examples of this sort. But CFGs will also turn out to be too weak, in the sense that it is very hard, and sometimes impossible, to describe very natural human language examples using just CFG rules.

After that, we will define and describe probabilistic context free grammars (PCFGs), as one attempt to get at the ambiguity problem in human language syntax. We will also give one standard parsing algorithm, based on dynamic programming, that can parse sentences using CFGs in polynomial (cubic) time in the length of input sentences, as mentioned earlier. This kind of algorithm forms the basis of many current Treebank-based parsers, but it is also too strong and too weak. It is too strong in that it is too good: it parses sentences like the boy got fat melted without any trouble at all, unlike a person. And it finds all possible parses for, e.g., natural language processing book. We will use the probabilistic nature of PCFGs to try to pull out the ‘best’ or n-best parse, and this helps, but not always.