MENU: on board
- Goal 1: Tag! You’re It: POS tagging by HMMs (Hidden Markov Models)
  o Key: remembrance of things past (6.034, A* search)
- Goal 2: Going nonlinear: why move from beads-on-a-string to context-free grammars? Learn why CFGs and what their strengths & weaknesses are
- I believe in an America where...
- Martha Stewart’s revenge: a new complexity measure – Recipe-complete

1. Goal: part of speech tagging. WHY? (for parsing, named-entity tagging)
Example Input: the can can fly home
Output: tagged, using Penn Treebank tags as: DT NN MD VB NN (‘determiner noun modal-verb, verb, noun’)

We can view this as a pair of sequences:
\[ x_1, x_2, \ldots, x_n \] the word sequence (input)
\[ y_1, y_2, \ldots, y_n \] the tag sequence (output)

Task: learn the function \( f \) that pairs \( x^{(i)} \) with it matching \( y^{(i)} \)
This is a supervised machine learning problem, maps input \( x \) to labels \( f(x) \)

There are two ways to look at this problem. The first, we used last time for our language ID problem is the conditional probability approach:

\[
\text{Define } p(y|x), \text{ then estimate the parameters of the model } p:
\]

\[
f(x) = \text{argmax } y \in Y \ p(y|x)
\]

This is what we did before. But recall, we also used Bayes’ theorem. This second approach, common in NLP, is a generative approach. Rather than directly estimating the condition probability \( p(y|x) \), we model the joint probability,

\[
p(y, x)
\]

with parameters estimated from the \( x, y \) pairs, as seems natural. Moreover, we can further decompose this as:

\[
p(y, x) = p(y) \ p(x \mid y)
\]

and estimate \( p(y) \) and \( p(x \mid y) \) separately. Recall that:

\[
p(y) = \text{prior probability distribution of the tags } y, \text{ while}
\]

\[
p(x \mid y) = \text{probability of generating the observed word sequence } x, \text{ given that the underlying tag sequence was } y. \ (\text{Like speech recognition.})
\]

So we can turn this equation around using Bayes’ Thm to get \( p(y|x) \), for every \( (x, y) \) pair:

\[
p(y \mid x) = p(x \mid y) \ p(y)/ \ p(x)
\]

So, to find the best value for \( p(y \mid x) \), we want to compute argmax \( y \):

\[
f(x) = \text{argmax } y \in Y \ p(y \mid x) = \text{argmax } y \in Y \ p(x \mid y) \ p(y)/ \ p(x)
\]

As with Google translate, \( p(x \mid y) \) is the ‘noisy channel’ that takes the source tags \( y \) with probability \( p(y) \), and then ‘scrambles’ them. It is a generative model (the tags generating the ‘surface’ sequence \( x \)).

To proceed, let \( V \) be the vocabulary for the words, a finite set. A word will be denoted \( w \) and a word sequence \( n \) long is denoted \( w^n \)
Let $T$ be the set of tags, also a finite set. A tag will be denoted $t$, and a tag sequence $n$ tags long is denoted $t^n$.

(This is to follow the JM book, p. 139-141, more or less.)

We want to choose the tag sequence $t^n$ that maximizes our Bayesian equation, the maximum likelihood times the prior:

$$\hat{t}^n = \arg \max_{t^n} p(w^n | t^n) \cdot p(t^n)$$

Recall, we are going to make two simplifying assumptions to do POS tagging in a new probabilistic system, a Hidden Markov Model:

1. The probability that a particular word appears depends only on its own part of speech tag, independent of the other words around it and independent of the other tags around it:

$$p(w^n | t^n) \approx \prod_{i=1}^{n} p(w_i | t_i)$$

For example, if we have the tag VBZ, then the word is like to be is.

2. The probability of a tag appearing is dependent only on the previous tag, rather than the entire tag sequence—the bigram assumption we’ve seen before.

$$p(t^n) \approx \prod_{i=1}^{n} p(t_i | t_{i-1})$$

We can now rewrite our likelihood x prior equation using these 2 simplifying assumptions to get the following to estimate the most probable tag sequence:

$$\hat{t}^n = \arg \max_{t^n} p(\hat{t}^n | w^n) \approx \arg \max_{t^n} \prod_{i=1}^{n} p(w_i | t_i) p(t_i | t_{i-1})$$

This has two kinds of probabilities: (1) the tag transition probabilities (which are based on bigrams); and (2) the word likelihoods.

**Q: what do these probabilities represent?**

**Ans:** $p(t_i | t_{i-1})$ denotes the probability of a tag given the previous tag, e.g., that determiners are very likely to precede adjectives and nouns (the horse, a big horse). So, $p(\text{NN} \mid \text{DT})$ and $p(\text{JJ} \mid \text{DT})$ ought to be relatively high. But in English, adjectives don’t usually precede determiners, so $p(\text{DT} \mid \text{JJ})$ should be low.

We can compute this by the usual MLE methods (using smoothing!)

Example: In the Brown corpus, DT occurs 116,464 times; DT followed by NN 56,509 times. The MLE estimate is $56,509 / 116,454 = 0.49$ (as expected)

The word likelihood probabilities $p(w_i \mid t_i)$ denote the probability that, given we see a certain tag, it will be associated with a particular word. For example, if we see VBZ, likely to be associated with verb is. (It is not the most likely tag, given a word! Bear this in mind!)
We can estimate these probabilities from counts in the usual way as well. In the Brown corpus, VBZ occurs 21,627 times, and VBZ is the tag for *is* 10,073 times. So, \( p(\text{is}|\text{VBZ}) = \frac{10,073}{21,627} = 0.47 \)

To repeat: this value is not the most likely tag for *is*; what this probability denotes is that if we see the tag VBZ, how likely is that associated word going to be *is*?

To actually do the computation, we will use a so-called **Hidden Markov Model** (HMM) tagger as a model for this, which is the most appropriate model for this situation, because it captures precisely these **TWO** knowledge sources. In fact, we’ll build a **bigram HMM**.

For Markov chains, the output symbols are the **same as states**. (If we see the word ‘it’, we’re in the state where we’ve seen the word ‘it’.)

An HMM is like a Markov model, but with an important addition: the transitions occur between states, here **tags**, are **not observed**, (hence, hidden), from direct observation. As we make a transition between states, based on the two preceding tags \( t_{i-2} \), and \( t_{i-1} \), the HMM can **emit** an output \( x_i \), the word that is observed. We have to infer what the most likely state sequence is ‘underneath’ – the tag sequence – given an observed word sequence.

Here’s a simple picture. To capture the info about what tags follow what other tags, we might suppose that or Nouns **always** occur after Determiners like *the*, as in *the book, the idea,* (but not: *the the*…). We use a familiar Markov chain to capture facts like these – at the level of POS sequences (remember *colorless green ideas*?)

![Diagram](DT--1.0--NN)

So that’s the first part of the information we represent.

The second part is to add an **output** from the NN state, that represents the likelihood that we will see a particular word, say *book*, **given** that the state is NN, ie, the likelihood of *book*, given NN. Of course, we’d have to have output likelihoods for every word in our vocabulary: *apple, the*, etc. – clearly for *the* the output probability would be very low, or 0. Note that what we observe is only the word – the state NN is ‘hidden’ from us. Also note that the likelihood emissions should sum to 1, to make it a valid probabilistic model, just as the probabilities of the transitions leaving any particular state, like DT, must sum to 1, for the same reason. (So if it there were a JJ **and** an NN states after DT, those two transitions probabilities would have to add to 1.0).
An HMM will always consist of these two components. The transitions – the $A$ matrix, and the emission likelihoods, the $B$ matrix.

More precisely, an HMM consists of the following parts:

- A finite set of states, $Q$ (these correspond to the tags)
- A transition probability matrix $A$. Each $a_{ij}$ denotes the probability of moving from tag state $i$ to tag state $j$ such that the sum over all the probabilities from state $i$ is 1, for all states $i$. That is, $\sum_{j=1}^{n} a_{ij} = 1, \forall i$ (this must add to 1 to be a valid probability over the transitions from any particular state)
- A sequence of $T$ observations, (“observed words” ), $o_1, o_2 \ldots o_T$ each one drawn from a (finite) vocabulary $V$ – these correspond to the word sequences, $w$.
- A sequence of observation likelihoods $B=b_{i}(o_t)$, also called emission probabilities, each denoting the probability of an observation $o_t$ begin generated from a state $i$.
- A special start state and end state, not associated with observations, along with transitions out of the start state, and into the end (final) state, [along with a special initial probability vector to enter any starting states, $\pi$, the initial state distribution that adds up to 1.0] (e.g., if just two possible states after the Start state, then the vector has to be, eg, [0.8, 0.2])

So we have our 2 matrices: $A$ corresponds to the prior probabilities, for the tag-to-tag transitions, and $B$ corresponds to the likelihoods, for the tag-output word ‘emissions’. Let’s draw a simple HMM to correspond to a (simplified!) task for HMM tagging. This will let us illustrate one of the two (or 3) key tasks involved with HMMs:

1. Given a particular HMM, and a sequence of observations (here, words), find the highest probability state sequence (tags) and corresponding emitted word sequences that accounts for the words actually observed (“decoding,” in particular, Viterbi algorithm)
2. Given an HMM, learn the transition tables $A$ and $B$ from paired input, tag sequence. (We defer this for now, returning to it in a later lecture.)

OK, for our simple tagging example, let us suppose that we have the following words and tags:

- 4 Part of speech tags: DT (determiner), NN (noun), MD (modal verb), VB (verb, no tense)
- 4 words: the (DT), can (DT, NN, VB), fly (NN, VB), home (NN, VB)

Assume from training data that we have learned the following bigram transitions, represented here as a network. Note that we have assumed the initial state vector to each of the 4 possible tag states is equal – a uniform probability of 0.25 each (we might want to make it DT)…

This diagram just gives the expected probability relationships from tag to tag, in sequence. E.g., that NN (nouns) are preceded by determiners (DT), etc.
Q: What should the sum of probabilities **exiting** any state be?
A: They must add to 1!

So all this part of the HMM does is to express the tag bigrams, the transition probabilities between 6 hidden states. We store these probabilities in a variable $a$: $a_{i,j}$ gives the transition probability from state $i$ to state $j$; e.g., DT $\rightarrow$ NN is 0.9, so if we call DT state #1, and NN state #2, we have $a_{1,2}=1$. $a_{0,j}$ gives the transitions from a special ‘start’ state 0, that sets things in motion, while $a_{6,j}$ gives the transitions to a special final state, where the end-of-sentence symbol has been read.

For the second part of an HMM, the $B$ table, we have to draw the **emission probabilities** (the likelihoods) from each hidden state to the each possible output word. Let’s tack this on below. (The numbers are made up as before, but are meant to correspond to what one might expect.) To avoid clutter, just the emission probabilities are drawn in here, and we have color-coded the emissions from each tag state. (Black from DT; Blue from NN; Green from MD; and Read from VB).

**Q: again, note** that the emission probabilities from each tag set **must sum to 1, to be a valid probability.** For instance, from the VB state, we have the emission pr’s: 0.001 (to the); 0.20 (to can); 0.7 (to fly); to 0.099 (to home). **Note** that the sum of probabilities going into a particular output word need not, and do not, generally sum to 1.
The basic computation of a single tag-output word pair then looks like this:

\[
\begin{align*}
\text{pr} &= \text{pr transition} \times \text{pr emission} \\
\hat{a}_{ij} \times b_s(o_t) &= a_{12} \times b_2(o_2) = 1.0 \times 0.249
\end{align*}
\]

If we put these two parts of the HMM together, we get one big, hard to read mess.

Here’s a picture of these two tables in – table form:
Our goal: given some finite word sequence that is \( T \) (time steps) long, \( w_1, w_2, \ldots, w_T \), find the sequence of tag states that has the highest probability of generating that sequence (the hidden state sequence).

We can lay out the picture as a *trellis*, a sequence of states that is \( T \) words long, and \( N+2 \) high, that looks like the following. We work our way through the words, left to right. Above each word is a *column* of the possible states (here, 4 such states, for the tags, plus special start and end states, so in fact 6 in all). Thus the trellis diagram is a matrix that is \( T \) long, where \( T=\# \) of words in the sentence, and \( N+2 \) high.

---

**Transition Table A**, for tag-to-tag bigram transition probabilities estimated from training data. (Well, fake data)

(Rows must sum to 1)

<table>
<thead>
<tr>
<th>TO TAG:</th>
<th>1 DT (determiner)</th>
<th>2 NN (noun)</th>
<th>3 MD (modal verb)</th>
<th>4 VB (verb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM TAG:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 DT</td>
<td>0.005</td>
<td>0.9</td>
<td>0.09</td>
<td>0.005</td>
</tr>
<tr>
<td>2 NN</td>
<td>0</td>
<td>0.11</td>
<td>0.19</td>
<td>0.7</td>
</tr>
<tr>
<td>3 MD</td>
<td>0.05</td>
<td>0.05</td>
<td>0.10</td>
<td>0.8</td>
</tr>
<tr>
<td>4 VB</td>
<td>0.45</td>
<td>0.54</td>
<td></td>
<td>0.005</td>
</tr>
</tbody>
</table>

**Transition Table B**, for ‘emission’ probabilities, probability of outputting word, given a tag, estimated from training data. (Well, fake data)

(Columns must sum to 1 if over all words in vocabulary \( V \))

<table>
<thead>
<tr>
<th>TAG:</th>
<th>1 DT (determiner)</th>
<th>2 NN (noun)</th>
<th>3 MD (modal verb)</th>
<th>4 VB (verb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prob output word</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the</td>
<td>0.99</td>
<td>0.001</td>
<td>0.0002</td>
<td>0.001</td>
</tr>
<tr>
<td>can</td>
<td>0.005</td>
<td>0.249</td>
<td>0.999</td>
<td>0.200</td>
</tr>
<tr>
<td>fly</td>
<td>0.004</td>
<td>0.350</td>
<td>0.0004</td>
<td>0.700</td>
</tr>
<tr>
<td>home</td>
<td>0.001</td>
<td>0.400</td>
<td>0.0004</td>
<td>0.099</td>
</tr>
</tbody>
</table>

Our goal then: *given* some finite word sequence that is \( T \) (time steps) long, \( w_1, w_2, \ldots, w_T \), find the sequence of tag states that has the highest probability of generating that sequence (the hidden state sequence).

---

\( N+2 \) rows, \( N=\# \) tag states (here, 6)
Let’s see how we would start the computation. We must go from the `start` state, at time step 0, to each state in the next column, column 1. We can calculate what the probability of each of these 4 paths is by using the \( a*b \) computation illustrated above. The \( a \) value is the same for each state transition from state 0 to the states in column 1: 0.25. (We could have been more sensible and used a different distribution than this uniform one; for example, *the* might be a more common sentence beginning.) In any case, when we do the computation we get the following. We store the accumulated probability so far in an array `viterbi[s, t]` where `s` is the number of the state we have moved to, and `t` is the time step. So in, in the first link from state 0 to state 1, we have `viterbi[0, 0] = 0.2475`.

What next though? If we try to do this computation in a ‘brute force’ way, it explodes combinatorially: if we wrote down the set of all possible tag states, if there are \( N \) states in all then at each step, we need to keep track of \( N^2 \) probabilities, so the calculation would grow as \( N \times N \times \ldots \times N \times T \) times, or \( N^T \). This is far too slow if \# tags = 40, and \( T=10 \). Interestingly, it wasn’t until 1986 that a solution was proposed for this in the area of NLP, even though the method, which is just A* search, had been invented (and re-invented) many times.

We want the method to be polynomial in the number of states and linear in the length of the word sequence. We in fact will use a method that is \( O(T N^2) \), so just the \# of states squared times the sequence length, and has an accuracy of just over 90%.

The so-called *Viterbi algorithm* can do this, to efficiently read off the most likely sequence of tags, given some observed sequence of words, by making use of dynamic programming, A* search in effect (but now we’re trying to calculate the maximum ‘length’ path, not the shortest). The key idea is that we can compute the best paths looking at the maximum probabilities to the states immediately preceding the one at step \( t \), that is, using the values stored at `viterbi[s, t-1]`. 
In all, we will use 3 variables to store and manipulate the probabilities to find the highest probability path from the start state to the final state.

- The variable $viterbi(s, t)$ will be used to store the maximum (highest) probability path to a state $s$ at time step $t$.
- The variable $a_{ij}$ stores the transition probability from state $i$ to state $j$.
- The variable $b_i(o_t)$ stores the emission probability from state $i$ given observed word $o$ at time $t$.
- Thus the total probability of extending the path from state $i$ 1 step to state $j$, given the observed word $w_i$ is this, since we go back to find the previous best (highest) probability value stored at the state prior to the current one we are working on, via the computation: $viterbi[s,t–1] * a_{ij} * b_i(o_t)$
- The key idea behind the Viterbi algorithm is that it is like A* search in 6.034: we use dynamic programming to store only the best possible paths so far in our search from start to the goal (end of the sentence).

The algorithm below does the job. Let’s see how. We have already seen how to compute the initial step to each of the states in column 1, 1 through N, which is the initialization step of this method. We also set a backpointer from each of these states back to state 0. The backpointer value will be used to keep track of the sequence of states to follow once we’re done: we’ll trace back from the best state at the end of the sentence, which will give the state (tag) sequence with the highest probability of generating the observed word sequence.

```plaintext
function VITERBI(observations of len $T$, state-graph of len $N$) returns best-path

create a path probability matrix $viterbi[N+2,T]$

for each state $s$ from 1 to $N$ do ; initialization step
    $viterbi[s,1] ← a_{0,s} * b_s(o_1)$
    $backpointer[s,1] ← 0$
for each time step $t$ from 2 to $T$ do ; recursion step
    for each state $s$ from 1 to $N$ do
        $viterbi[s,t] ← \max_{s'} viterbi[s',t-1] * a_{s',s} * b_s(o_t)$
        $backpointer[s,t] ← \arg\max_{s'} viterbi[s',t-1] * a_{s',s}$
    $viterbi[q_F,T] ← \max_{i=1}^N viterbi[s,T] * a_{s,q_F}$ ; termination step
    $backpointer[q_F,T] ← \arg\max_{i=1}^N viterbi[s,T] * a_{s,q_F}$ ; termination step
return the backtrack path by following backpointers to states back in time from $backpointer[q_F,T]$
```

The outermost loop in the algorithm, for each time step $t$ from 2 to $T$ do goes left to right, word by word, from the to home in our example (so $t=1, \ldots, 5$, and $T=5$).

When we reach a column at time step $t$, we can show how we can compute the required values for $viterbi[s, t]$ in terms of the maximum probabilities to each of the previous column’s states, namely, $viterbi[s, t–1]$. Assuming that $viterbi[s, t–1]$ holds the maximum path probability from the start state to state $s$, for $s=1, \ldots, 5$, we can compute the value for $viterbi[s, t]$ for each state (column value in column
by means of the innermost loop, finding the \textit{max} value from a state \( s \) that points back to each of the previous states \( s' \) in turn, as the product of 3 values: (1) the previous \textit{viterbi} value for the state \( s' \) (running from 1 to \( N \)); (2) the transition probability from state \( s' \) to \( s \), \( a_{s',s} \); and (3) the output emission probability (or likelihood) for that state \( s \), given the word at column \( t \), \( b_s(o_t) \). This maximum value is therefore the maximum probability extension from column \( t-1 \), state \( s \), to column \( t \), for that same state:

\[
\text{for each state } s \text{ from 1 to } N \text{ do} \\
\text{viterbi}[s, t] = \max_{s'=1, \ldots, N} \text{viterbi}[s', t-1] \ast a_{s',s} \ast b_s(o_t)
\]

For example, if we had just finished filling in the column of \textit{viterbi} values for column 4, and moved over to column 5, then we would first try to find the maximum extension to state 1, DT. We can imagine this as four arrows pointing back to each column 4 state from state 1, DT, in column 5. Computing the maximum of the three terms above, over each previous state in column 4, will give the value for \textit{viterbi}[1, 5]. Of course, we must also add the right \textit{backpointer}: we must record the previous state \( s' \) that gives us this maximum value. This is done by one additional line of code, the third line below:

\[
\text{for each state } s \text{ from 1 to } N \text{ do} \\
\text{viterbi}[s, t] = \max_{s'=1, \ldots, N} \text{viterbi}[s', t-1] \ast a_{s',s} \ast b_s(o_t) \\
\text{backpointer}[s, t] = \text{argmax}_{s'=1, \ldots, N} \text{viterbi}[s', t-1] \ast a_{s',s}
\]

Here we have used \textit{argmax} to find the \textit{value} of \( s' \) that yields the maximum value. Note that we do not have to multiply by the third term \( b_s(o_t) \) when doing this computation, because \( b_s(o_t) \) is constant for all states at a fixed column \( s \) (here, 1) and time \( t \) (here, 5).

Once finished with this state, we can then move to compute the \textit{viterbi} value for state 2, corresponding to tag, NN, and so on, completing filling in the \textit{viterbi} values and backpointers for the rest of column 5, as shown below.

To fill in the \textit{max prob} value for a cell, here, NN, take \textit{max} over the \textit{extensions} of all paths that lead to that cell.

To finish, we carry out a calculation that is similar to the initialization step: all states make transitions to the special final state, state 6, with the probabilities as given in the transition matrices above (we assume these to be uniform). If we actually run the calculation with the numbers for the transitions and
emission probabilities given earlier, we get the following for the sequence *the can can fly home*, showing the probabilities at each time step and the final overall probability for the best path:

\[
\text{('the', 'can', 'can', 'fly', 'home')}\]

\[
\begin{array}{c|ccccc}
0 & 1 & 2 & 3 & 4 \\
\hline
\text{MD} & 0.00005 & 0.02225 & 0.01052 & 0.00000 & 0.00000 \\
\text{DT} & 0.24750 & 0.00000 & 0.00000 & 0.00001 & 0.00000 \\
\text{VB} & 0.00025 & 0.00024 & 0.00776 & 0.000589 & 0.00010 \\
\text{NN} & 0.00025 & 0.05546 & 0.00151 & 0.000146 & 0.00127 \\
\end{array}
\]

\((0.0012734383573296004, [\text{'DT', 'NN', 'MD', 'VB', 'NN']})\)

To see what the order time complexity is for this algorithm, note that the inner most **for** loop takes time \(O(N)\), because we loop over each state \(s'=1\ldots N\). Moving out one loop, we see that the next **for** statement is executed for \(s=1\ldots , N \) times (moving up a column). Finally, the outermost loop runs over time steps \(t=1\ldots T\). So in all, the running time is \(O(N^2 T)\), quite a considerable savings over the brute force exponential time calculation. Here are some pictures to show how the running time can be calculated.

This shows that the two inner loops work in time \(O(N^2)\). Now we add the outer loop, the operation column by column, for \(t=1\ldots , T\), obtaining an overall running time \(O(N^2 T)\):
A bigram tagger like this will achieve an accuracy of a bit close 90%. (Remember, it is hard to do good: a unigram system that simply assigns part of speech tags by just using the MLE estimate of a word’s tag (e.g., that ‘fly’ is 0.89% of the time a Noun), will get performance in the 80% range.) To get better, one must use trigrams, for the part of speech tag sequences. In this case, instead of calculating viterbi[s, t], we would have a 3-dimensional matrix, viterbi[u v t]. Of course, training here requires more data, as there are more parameters. Past trigrams, however, there seem to be diminishing returns in terms of accuracy. The hardest part of the job, whether bigram or trigram, is getting good estimates for the emission probabilities from each state, $b_i(\cdot)$, which must be estimated for each possible word in the vocabulary.

We will revisit the tagging issue in the context of a more powerful method called log-linear regression or maximum-entropy models, a few lectures later on.

Goal 2: Going nonlinear. The why and what of context-free grammars.
Main points:
Why context-free grammars? (CFGs)
Defining context-free grammars
Defining the notion of ‘derivation’ in context-free grammars
Defining the notion of ‘parsing’ with respect to CFGs
What makes CFGs hard computationally? (Many rules – how do we figure out the rules for any human language? And ambiguity – an exponential # of difft derivations in some cases)
So far, our language models have been based on a linear ‘beads on a string’ models: word sequence obey just concatenation, so they encode *precedence* information (what follows what), but nothing more. But is sufficient to describe human language? It does not appear so for several obvious reasons:

1. **Descriptive power.** Human language clumps words into groups, called phrases. There isn’t anything in the linear language models to do this.
2. **Linear models must** connect up a ‘long distance dependency’ (as in the Proust example!) by ‘paying attention’ to all the words in between, but this is not correct.
3. We need a better way to describe human language syntax that ‘hooks into’ semantic interpretation.

Let’s explore these points, and then present the formalism of *context-free grammars* (CFGs) as one way to meet these challenges. This is one way of adding *hierarchical* description to sentences. Then we will take up the computational issues that arise when we do this.

First, consider even a simple fragment like *deep blue sky*. This clearly has at least two meanings, one where *deep blue* is grouped together, as a kind of color that modifies *sky*, and another where *blue sky* is grouped together, and *deep* modifies these as if they were a single unit. (Groups of words that ‘group together’ as if they were single units are known as *phrases*, as we shall describe below.) In this case, we might annotate the two possible descriptions as follows, using square brackets to demarcate grouping:

```
[deep blue] sky vs. deep [blue sky]
```

Why do we need two different descriptions? Because if we are assuming something like the computational theory of mind, or how to get computers to act like human minds, then the only way to distinguish the two different *meanings* of *deep blue sky* is to associate them with two different *representations*, say like the two above. Otherwise, there is no way for a person (or a computer) to tell them apart. (Note there could of course be very different ways to represent the two possibilities; we can remain agnostic about this. Similarly, one could imagine that there are two different computational *sequences* that correspond to the 2 different representations, which we’ll arise naturally when we look at context-free grammars.) There is a kind of hack-ish way to ‘solve’ the *deep blue sky* problem in this particular case, using just the ‘linear’ machinery of state sequences that we have already provided, but it really is a hack. (You should think about how to solve it; answer in the next lecture.)

Taking a step back from our example, it’s pretty clear that if we introduce the notion of *hierarchical structure* then we can get two very natural representations for the two different meanings associated with *deep blue sky* as follows:
In fact, many words ‘clump together’ like this. To take another simple case, in the sentence, *I know...*, all of the following word sequences can follow: *I know {Romney}, or {the politician}, or {the guy who might be from Michigan but lives in Massachusetts}*. That is, for the purposes of English syntax, these word sequences all behave alike with respect to the way they follow words – we can substitute one for another one freely with respect to syntactic context, though sometimes obtaining a different meaning, of course. This behavior is confirmed by other syntactic manipulations; for example, all of these word sequences can be put at the front of a sentence: *Romney, I know; the politician, I know, etc.* In fact, the comma indicates an intonation break when these sentences are pronounced, that sets of the group of words. All facts like these suggest that the ‘grouping’ of words into sequential groups is psychologically real. We can call these groups *phrases*.

Superficially, there are many types of phrases in English (or other languages): what we have just seen above with *Romney, the politician, etc.* are called *Noun Phrases* (or NP for short). There are also *Verb Phrases* (a verb plus something more, as in: *know the politician, know that Romney likes ice-cream*, and so on); there are also *Prepositional Phrases*, which in English begin with a Preposition like *in, under, with, etc.: with Romney, under the politician, etc.* You may see a pattern emerging here which we’ll return to. How many different kinds of phrases are there in a language? The linguists’ answer may surprise you – we shall come to that in the next lecture.

We can put together the notion of a phrase along with a simple idea of production rules from 6.034 to give the classic definition of a *context-free grammar* (CFG), one way to represent hierarchical structure found in human language.

**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:
  
  \[ X \rightarrow Y_1...Y_n, \text{ where } X \in N \text{ is a single nonterminal, and } Y_i \in NV \text{ for each } Y_i, n \geq 0 \text{ (so the } Y_i\text{ comprise any string of terminals and nonterminals. We will include a special ‘empty symbol’ epsilon to denote the empty string. The key point is that there is just one nonterminal on the left-hand side of any context-free rule.)} \]
- *S* is a special designated ‘Start’ symbol (we will usually just call it ‘Start’)

OK, let’s see how this works by defining a simple CFG for a toy fragment of English. We will use the Penn Tree bank part of speech tags for the most part, so you can get used to them.

\[ \begin{align*}
N &= \{S, NP, VP, PP, DT, V_t, V_i, NN, IN\} \quad \text{('IN' is the funny tag that denotes a Prepositional Phrase)} \\
S &= \text{START} \\
V &= \{\text{sleeps, saw, man, dog, car, street, he, the with, in, down}\} \\
R &= \begin{align*}
S \rightarrow \text{START} &\rightarrow S \\
S \rightarrow NP \ VP &\rightarrow \text{sleeps} \ IN \rightarrow \text{with} \\
S \rightarrow NP \ VP &\rightarrow \text{saw} \ IN \rightarrow \text{in} \\
S \rightarrow V_t &\rightarrow \text{man} \ IN \rightarrow \text{down} \\
S \rightarrow V_t &\rightarrow \text{dog} \\
S \rightarrow V_t &\rightarrow \text{car} \\
S \rightarrow DT \ NN &\rightarrow \text{street} \\
S \rightarrow NN &\rightarrow \text{he} \\
PP \rightarrow IN \ PP &\rightarrow \text{the}
\end{align*} \]
The two columns on the right of this rule set all have part of speech categories as their left-hand sides, and just a single terminal item (a word) on the right; these are often called *preterminal rules*. The rules in the left-most column just derive nonterminals (phrases). We see that we have only two categories of verbs, $V_i$ to denote *intransitive verbs* (that take no arguments, as in *the dog sleeps*), and $V_t$ to denote *transitive verbs* (that take one argument, as in *the dog saw the man*). We shall see that many more finer grained categories are required for a ‘real’ English rule set.

We now turn to the notion of how a CFG *generates* a sentence. (This is why such systems are called *generative grammars.*) The basic idea is that we begin with the $START$ symbol and successively replace any left-hand side of a rule in $R$ with its right-hand side, until we have all terminal elements, and so cannot apply any further rules. The string of terminal elements is said to be a *sentence derived by the grammar*. Thus, derivation is relative to a grammar. The string so derived is said to be in the language *generated by the grammar*.

More precisely, given a CFG $G$, we say that a *leftmost derivation* is a sequence of strings $s_1, s_2, \ldots, s_n$ where:

- $s_1$ is the $START$ symbol;
- $s_n \in V^*$ (the final string is all terminals (this may be the empty string if we’ve included it in our vocabulary set and rules);
- Each $s_i$ for $i=2,\ldots,n$ is derived from $s_{i-1}$ by selecting the leftmost nonterminal $X$ in $s_{i-1}$ and replacing it with some string $\beta$ where $X \rightarrow \beta$ is a rule in $R$.

When there is more than one rule expands a given nonterminal, we can choose one at random.

**Example.** Using our toy grammar we can generate the following sentence by a left-most derivation, where we have *underlined* the nonterminal symbol expanded at each step $s$. Each line is called a (amazingly) – *derivation line*. (Q: how many sentences does this grammar generate? Ans: an infinite number.)

| S1 | $START$ |
| S2 | $S$ |
| S3 | $NP \ VP$ |
| S4 | $DT \ NN \ VP$ |
| S5 | $the \ NN \ VP$ |
| S6 | $the \ dog \ VP$ |
| S7 | $the \ dog \ V_t \ NP$ |
| S8 | $the \ dog \ saw \ NP$ |
| S9 | $the \ dog \ saw \ DT \ NN$ |
| S10 | $the \ dog \ saw \ the \ NN$ |
| S11 | $the \ dog \ saw \ the \ man$ |

Note that there could be *other* orders in which to derive the same sentence wrt a grammar. A left-most derivation is called *canonical* because it specifies a fixed non-terminal expansion sequence. The other important canonical derivation order is called *right-most*, which expands the right-most nonterminal at each step; this is useful in parsing, as we shall see.

A derivation sequence establishes a hierarchical tree representation that looks like this:
This graph structure represents all the hierarchical and precedence relationships among the words, all the groups of words that form phrase, along with (a part of) the way in which the sentence was derived from the grammar. (We will pursue what this kind of ‘graph structure’ does and does not represent about sentences, and what is actually needed for natural language.)

The set of all sentences that can be generated by a grammar is called the language generated by the grammar, denoted $L_G$. It is therefore the reflexive, transitive closure of the ‘replacement’ arrow operation, sometimes denoted by the double arrow $\Rightarrow$ with an asterisk placed above it to indicate “$0$ or more applications of $\Rightarrow$”. So, another way to define the language $L_G$ we can say:

$$L_G = \{ w \mid w \in V^* \& \text{Start} \Rightarrow^* w \}$$

To parse a sentence $w$ with respect to a grammar $G$, we have to find some sequence of derivation steps (rule applications) that can derive $w$ using $G$. We shall see that we have to be clever here, because there can be many possible derivations for some sentences and grammars, and the problem is actually easy shown to be NP-hard in many cases. In fact, there at least three big problems with using CFGs: (1) Ambiguity (multiple different derivations for the same sentence); a large number of rules; and representational problems with CFGs (they are both too strong and too weak).

To touch briefly on the ambiguity question, consider first these simple sentences, which usually boggle human language processors, but not computers:

The boy got fat melted
The boat floated sank

People usually quickly clump ‘boy’ and ‘got fat’ together, and assume the first sentence means something like, ‘the boy got fat’ and then the boy ‘melted’ – an awful Ridley Scott scenario. Similarly, people clump ‘boat’ and ‘floated’ together, so ‘sank’ comes as a surprise. Of course, these sentences are not ambiguous at all. The first one means that the boy by some means melted fat; and the second, that the boat that was floated (by someone or something), then sank. These kinds of sentences fool the human sentence processor, and are called ‘garden path sentences’ because they lead people down a blind alley in parsing. We shall see several other similar examples of this sort; human sentence processors seem to act reflexively, even if the result is nonsense.

The problem of alternative derivations for the same sentence can be just as hard. In an example like, the dog saw the man on the street, the Prepositional Phrase on the street can modify either the man (telling us where the man is), or saw, telling us where the ‘seeing action’ by the dog took place. These kinds of differences are called syntactic ambiguity, because they derive from different syntactic analyses, and result in distinct differences in meanings (which is partly what we wanted with CFGs to start with).
Such cases can quickly explode combinatorially. This is the ‘Recipe Problem’ or what might be called ‘Martha Stewart’s revenge’. The following fragment shows that if one uses conjunctions like and or or, then the number of possible parses can increase enormously. This is because when one says, “the big cars and trucks” this can be analyzed as either “the big cars” and “the trucks”, so that big applies only to cars; or “the big” “cars and trucks” (so that big applies to both cars and trucks). If we add more conjunctions, this explodes. So for instance, this fragment has over 100,000 possible interpretations given the usual way to think about conjunctions. A lot of possibilities for such a simple recipe:

*Combine grapefruit with bananas, strawberries and bananas, bananas and melon balls, raspberries strawberries and melon balls, raspberries strawberries and melon balls, seedless white grapes and melon balls, or pineapple cubes with orange slices...*

How can we deal with this? The answer, next time.