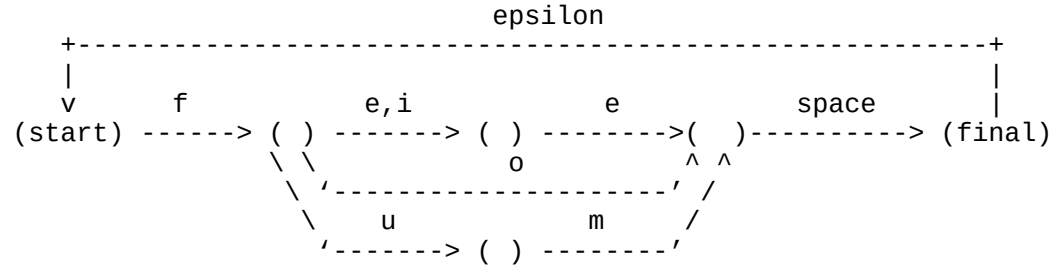
Official answers to the spring 2016 NLP final.

1. Here's one version:



2. unsmoothed (MLE): 1
   add-one smoothing: (3+1)/(3+50)
   add-one smoothing with backoff: (3+(50/1000))/(3+50)
   Witten-Bell smoothing with backoff: 3/4
     (We add the number of types seen in this context to the
     denominator. We don't back off since choo has count > 0
     in this context.)
   one-count smoothing with backoff: 1
     (Since there are no singleton types in this context!)

3. (a) 0.2.

       Remember that with the maximum-likelihood parameters, the
       expected features of a log-linear model match the observed
       features. (Because that's what makes the gradient vector 0 --
       recall http://cs.jhu.edu/~jason/tutorials/loglin/#6 and the
       log-linear handout.)

       The given formulas can be interpreted as saying that there are
       3 binary features. One fires only on "bwa," one fires only on
       "bwee," and one fires only on "kiki."

       The "bwee" feature is observed to fire 0.2 of the time,
       so in the maximum likelihood solution, the model matches
       this and predicts that it will fire 0.2 of the time.
       In this case, that simply means that p(bwee) = 0.2.

   (b) log 0.3, log 0.2, log 0.5 gives probabilities 0.3, 0.2, 0.5
       as required. In this case Z=1.

   (c) For any constant b, b+log 0.3, b+log 0.2, b+log 0.5 still
       gives probabilities 0.3, 0.2, 0.5. In this case, Z=exp b.
       So just choose any b != 0 to get a different solution.

   (d) The L1 regularizer prefers smaller values of
           |theta_bwa| + |theta_bwee| + |theta_kiki|

       Which value is best depends on what value of b you chose.
       In this case, the L1 regularizer is minimized if you
       choose b = -log 0.3, so that the median weight is 0
       and the other weights are close to 0.

       (Of course, the regularizer would be even happier if you also
       reduced the distance between the weights, bringing them all
       closer to 0! That gives you the regularized solution, which
       is NOT a maximum-likelihood solution.)

   (e) 0.25. We observe the "bw" feature in 0.5 of the training
       data, so under the maximum-likelihood parameters, the
       model expects this feature to fire 0.5 of the time. That
       means p(bwa)+p(bwee) = 0.5. But according to the formulas,

p(bwa)=p(bwee): the features don't distinguish between them.
Therefore p(bwee) = 0.25.

(f) Yes. The new model will correctly predict the observed counts of bwa, bwee, and kiki in an average sentence. Thus, it will correctly predict the SUM of these counts, which is 4.0.

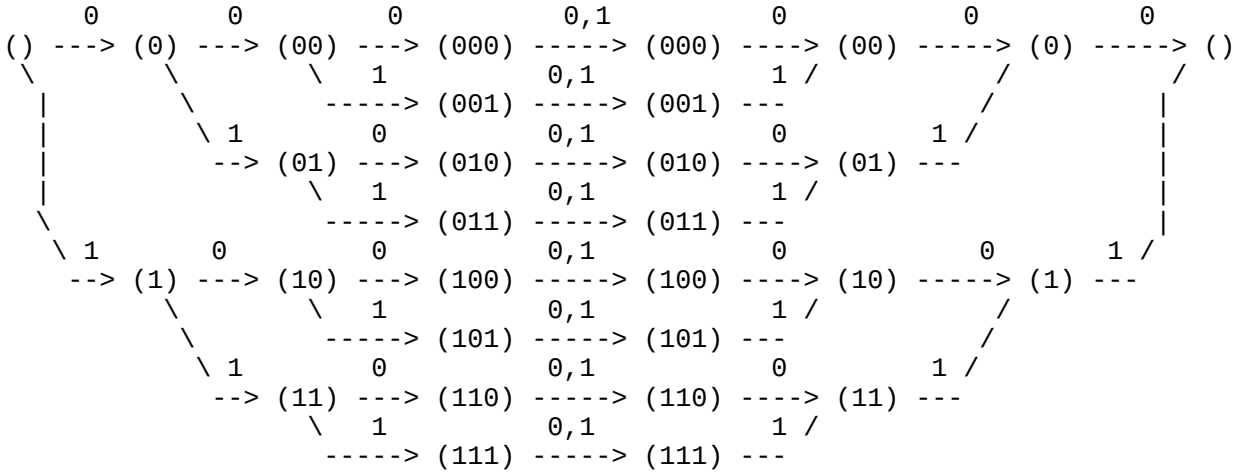
To be more concrete, if there are N training sentences, then the information in the problem implies that there are 4N training words, comprising 4N*0.3 bwa tokens, 4N*0.2 bwee tokens, and 4N*0.5 kiki tokens. Thus the average observed count per sentence is 4*0.3 bwa, 4*0.2 bwee, 4*0.5 kiki. The trained model's predictions will match these counts, and therefore it predicts 4*0.3 + 4*0.2 + 4*0.5 = 4 tokens per sentence.

4. (a) XFST syntax: [x .o. Phone*].l
Thrax syntax: Project[Compose[x,Phone*],'output']

(b) Nothing! It's the empty language.

(c) [(Nice* - ?*Nasty?*) .o. Phone*].l & ???????
or
[(Nice* - $Nasty) .o. Phone*].l & ???????
where $ is the XFST "containment" operator.

5. (a) Each of the first 3 states along a path must remember all characters that have been read so far, so it can emit them in reverse. Each of the remaining states must remember all characters that are left to emit.



One of you wrote, "There must be a better way!" Indeed - a CFG! The FSA above is basically keeping a stack in its state. CFGs have stacks built-in so that they can deal with nesting.

You can, however, make it smaller by using an RTN (recursive transition network), where an edge can be labeled with the name of another automaton. That makes it easier to repeat structure. It basically looks like this regexp:

```
Palindrome7 = 0 Palindrome5 0 | 1 Palindrome5 1
Palindrome5 = 0 Palindrome3 0 | 1 Palindrome3 1
Palindrome3 = 0 Palindrome1 0 | 1 Palindrome1 1
Palindrome1 = 0 | 1
```

(b)
```
P -> epsilon      # smallest length-0 palindrome
P -> 0            # smallest length-1 palindromes
```

```
P -> 1
P -> 0 P 0        # longer palindromes
P -> 1 P 1
```

where the root symbol is P, or where we have another rule ROOT -> P. Note the use of center-embedding - something that seems to be restricted in natural languages.

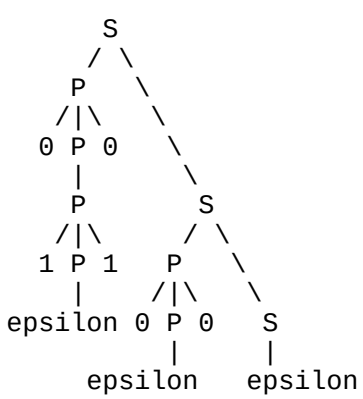
Many of you omitted the rule P -> epsilon, and thus you were only able to get odd-length palindromes. An alternative would be the pair of rules

```
P -> 0 0
P -> 1 1
```

but that misses the length-0 palindrome (the empty string), so P -> epsilon is better.

(c) This is possible because a parser segments a sentence (recursively). A sequence can be represented by a right-branching parse. In this case, our parse is a sequence of subtrees, each of which represents a palindrome.

For example, we want the best parse of 011000 to be



The division into palindromes can be found by looking at the top-level P constituents (the ones that are children of S). The number of palindromes k is the the number of S -> P S rules.

How do we set the probabilities to make the shortest sequence be the best parse? As long as the recursive rule S -> P S has sufficiently low probability, we will prefer shorter sequences of palindromes. To avoid giving long sequences any competing advantage, here's a nice solution:

```
1/2  S -> epsilon    # S generates a right-branching seq of P nodes
1/2  S -> P S

1/4  P -> epsilon    # same P rules as before, but with probs
1/8  P -> 0
1/8  P -> 1
1/4  P -> 0 P 0
1/4  P -> 1 P 1
```

With these probabilities, any given palindrome of length i is generated by P with probability 2^-(i+2). You can easily verify this by induction: it's true for i=0 and i=1 (base cases), and for i >= 2, you must select the appropriate rule (P -> 0 P 0 or P -> 1 P 1) and then recurse with i' = i-2.

Therefore, if you divide a string of length n into k palindromes, the overall probability of the parse will be

2^-(k+1) for the S rules, times 2^-(n+2k) for all of the P rules together, or 2^-(n+3k+1) altogether. This is maximized by making k as small as possible, i.e., finding the shortest sequence.

(Question: What probabilities should you use if the alphabet has 3 symbols instead of 2?)

6. In class, we gave a quick analysis of Earley's that made it look like it would be O(n^3 G^2) in runtime -- whereas CKY is only O(n^3 G).

So is Earley's really asymptotically slower than CKY? No. This question shows that with a little care, we can get it down to O(n^3 G) as well.

(a) Attach, in both cases.
(We'll see the runtime of attach below.)
(b) Original: 2,0,1 (n columns * n start positions * G dotted rules)

(Since the dot is always at the end for a complete entry, we could say R dotted rules instead of G dotted rules, where R is the total number of rules in the grammar. However, in the worst case, the grammar uses only binary rules, so R = G/3.)

Modified: 2,1,0 (n columns * n start positions * V dotted rules)
(The only complete dotted rules have the form A -> ... . We forget the entire right-hand side. That's the speedup.)

(c) Linear search: 1,0,1 (n start positions * G dotted rules)

Using auxiliary data structures: 1,0,1
(Now it is only necessary to consider rules with a particular nonterminal after the dot. A hash table for each column can map from each nonterminal to the customers for that nonterminal. In the best case, this is a speedup. However, in the worst case it is still possible that all O(nG) entries in the column have that nonterminal after the dot.)

(d) Original algorithm: 3,0,2 (by multiplying O(n^2 G) complete entries from (b) with O(n G) time per entry from (c))

Modified: 3,0,1 (Now the auxiliary data structure does save us something asymptotically in the total work! Let's count the work a different way. We were adding up the time for each COMPLETE entry to find all its incomplete customers to the LEFT. Instead, let's do the accounting from the other direction: add up the time for each INCOMPLETE customer to be found by all complete entries to its RIGHT that will attach to it. There are O(n^2 G) incomplete entries and each is found in O(1) time by each of O(n) different complete constituents to the right -- thus the total runtime is O(n^3 G). For example, an incomplete entry in column 4 with NP after the dot will only be found by complete entries in later columns

that have the form (4, NP -> ... .), and
there is at most one of these per later
column, thanks to the modified Earley's
algorithm. So, we've saved a factor of G!

(e) 0,1 (each a rule of length k will turn into k-2 rules of length 3,
with total length 3k-6. So the total size of the
grammar goes up by a factor of < 3.)

(f) 3,0,1 (with careless loop ordering, it would be 3,3,0, but
3,0,1 is better since G is often less than V^3 and
never more; we saw this optimization in class)

(g) w = w_1 + w_2 + ...

More generally, w is the semiring sum (using the oplus
operator) of all w_1, w_2, ... For example, it is the sum if
we are doing the inside algorithm, or the max if we're getting
the max-probability parse, or the min if we're getting the
min-cost parse.

7. (a) Recall that B = beginning, I = inside, O = outside.
Each phrase (dish) has the form BI*, and these
phrases are separated by non-phrases of the form O+.

System brackets:

The [homemade buffalo sauce], served with [flair], goes
surprisingly well on [fried rice].

We'll compare these to the gold brackets:

The homemade [buffalo sauce], served with flair, goes
surprisingly well on [fried rice].

Of the 3 system phrases, 1 was gold, so precision = 1/3.
Of the 2 gold phrases, 1 was found by the system, so recall = 1/2.

Note that [homemade buffalo sauce] != [buffalo sauce], and this
mismatch is both a precision error and a recall error. (We
found something that wasn't a dish, and we missed something
that was a dish.)

F1 is the harmonic mean of precision and recall, that is, the
reciprocal of the average of the reciprocals:
1 / average(3,2) = 1/2.5 = 0.4.

As a santity check, note that F1 should always be somewhere
between precision and recall ...

(b) One example is O O I I O.
Another is I O O O O.

In general, "I" must be immediately preceded by "B" or another
"I". (Equivalently, it may not be preceded by "O" or BOS.)
That's because the first word of a phrase is B, not I.

(c) Construct [Legal .o. M .o. x].u, which filters the upper string
to be a legal tag sequence. Then take the best path as before.

Here we define Legal by
[BI* | O]* (a sequence of dishes BI+ and outside words O)
or
I => [B | I] _ (every I must be preceded by B or I)
Some of you suggested

[O* [BI*]]*
but this doesn't allow the sequence to end in O; you meant
[O* [BI*]]* O*

(d) alpha(T_i = B) * beta(T_i = B) / Z

where Z is the total probability of all paths through the trellis.

For the trellis, you can find Z as
Z = \sum_t alpha(T_i = t) * beta(T_i = t)
This works because there is exactly one correct tag t at each time i. It can be regarded as the normalizing constant for the posterior distribution over T_i.

My favorite way to compute Z is as beta(initial state), which is correct for any weight FSA. Since an FSA conventionally has exactly one initial state but many final states, beta(initial state) is more convenient than the "mirror-image" solution of summing up alpha(state) over all final states.

(e) Like the Viterbi algorithm with log-probabilities, this will be an algorithm in the (max,+) semiring. That is, we are maximizing over possible paths to find the best sequence, and the score of a path is given by a sum (stated in the problem).

We'll use score[i,tag] to represent the maximum score
of any length-i prefix that ends with tag.
Assume that all score[...] values are initialized to -infinity.

We'll take "O" to be the tag of the BOS word at position 0 and also the EOS word at position n+1. Thus, [0,"O"] is the start state and [n+1,"O"] is the end state.

Define benefit[i,tag] to be the benefit of predicting tag i at position i. For i from 1 to n, benefit[i,tag] = p(T_i = tag | x). Note that it is only scoring the tag unigram, because we'll be evaluated by the number of tag unigrams we get correctly. (Of course, the actual value of p(T_i = tag | x) was found using a bigram tag model.) Also, benefit[n+1,tag] = 0, since we get no points for predicting a tag for EOS.

We've already computed for i from 1 to n,
and we define p(T_{n+1} = "O" | x) to be 0 because we get no points for predicting "O" as the tag for EOS.

```
for i := 1 to n+1
   for tag in {"B","I","O"}
      for prevtag in {"B","I","O"}
         if not (tag=="I" and prevtag=="O") // avoid illegal OI bigram
            new := score[i-1, prevtag] + benefit[i,tag]
            if (new > score[i, tag])
               score[i,tag] := new
               backpointer[i,tag] := prevtag

// print tag sequence in reverse order by following backpointers
tag = "O"    // tag of EOS at position n+1  (end state)
for i := n downto 1
   tag = backpointer[i+1,tag]
   print tag
```

Note that we are unusually summing probabilities in the line "new := ..." That's because the expected total number of correct tags is a sum of probabilities over the tag positions. Usually we multiply probabilities, and some of you did that instead.

Some of you didn't do dynamic programming. Instead you tried a simpler greedy algorithm where you didn't loop over possible values of prevtag, but assumed it had been chosen at the previous iteration of the outer loop. Unfortunately, that means that the previous iteration made its choice without considering the rest of the sentence. This will not necessarily give the optimal answer.

One of you suggested changing the transition probabilities in the original model M to prohibit illegal tag sequences. That might ALSO be a good idea, but the posterior decoder could still produce an illegal tag sequence even so, unless we constrain it as above!

(f) i. W_{i-1} == "homemade" && T_i == "B"

ii. x_i is the last word of an NP

```
    && T_i != "O"          // x_i is in a dish name
    && T_{i+1} != "I"      // and is the last word of that name
                           //    (should treat this test as true in
                           //    the case i==n, where T_{i+1} is undef)
```

In each case, the feature that counts copies of this configuration will probably get positive weight.

8. (a) 3 edits (1 deletion, 2 insertions):

```
meter--
met-ric
```

(b)

```
       m:m/0   e:e/0   t:t/0   e:eps/1  r:r/0   eps:i/1  eps:c/1
(start)----> ()----> ()----> ()-----> ()----> ()-----> ()-----> (final)
```

(c) i. the Viterbi approximation to Expectation Maximization (Viterbi EM)
ii. sum_i log p(x_i, y_i)
iii. gets stuck in a local maximum
iv. Viterbi EM is not guaranteed to find even a local maximum

9. The problem setup gave you examples ("Mary loves John") that showed how main sentences would translate into predicates over worlds, and it even explained explicitly why this should be so. This is why your answers to (a), (d), and (e) should start with %w.
(As in the simplify script, I'll use % to mean lambda, %E to mean "there exists," %A to mean "for all.")

(a) %w %Et ((t > now) ^ president[w,t](Elmo) ^ ~president[w',t-1](Elmo))

Some people imagined that there was a moment called "future" (analogous to "now"), but of course there are MANY future moments. Not everything that happens in the future will happen at the same time! So you need a "there exists" quantifier to pick a particular future moment.

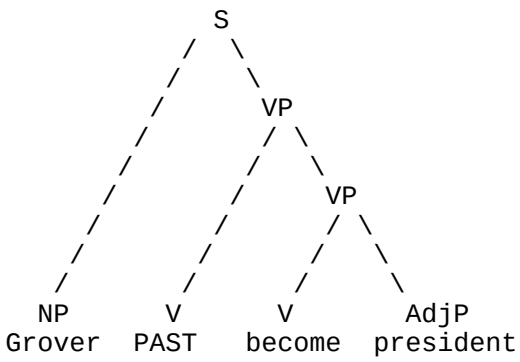
Some other people wanted Elmo to become president at time now+1. But the statement just says he'll become president some time in the future, not necessarily tomorrow!

(b) will become president
= %subj %w %Et (t > now ^ president[w,t](subj) ^ ~president[w,t-1](subj))
become president
= %t %subj %w (president[w,t](subj) ^ ~president[w,t-1](subj))
will
= %vp %subj %w %Et (t > now ^ vp(t)(subj)(w))

This one involves a pop/push maneuever, because
"will" applies to a untensed VP that has not
yet received its t argument but also has not
yet received its subject and world arguments.
You can verify that applying this function to "become
president" produces the above semantics for "will become
president."

(c) The main point is that you should run a morphological analyzer
that turns the sentence into "Grover PAST become president,"
with PAST playing the same role as "will." (You could also
write PAST as "-ed" (affix hopping) or "did" if you like.)

The leaves of the syntax tree are these morphemes, rather than
the words. We adopted this analysis of tense when covering
semantics in class.



(d) %w %Asubj %Ew' (abilities in w' are like those in w) ^
%Et (t > now ^ president[w',t](subj) ^ ~president[w,t-1](subj))

Note the order of quantifiers. For any particular person,
there is some accessible world in which they do become
president.

(The reverse order %Ew' %Asubj would imply that there's a
single world in which everyone does become president. That's
a possible reading of "Everyone can become president.")

There are many crazy possible worlds. As with all modals, we
have to restrict to some relevant set off those. In the above
semantics, I've interpreted "anyone can become president"
as "anyone has the ability to become president." There are
other interpretations, such as "anyone has the legal right
to become president" or "for any person, it might happen
that they become president."

(e) %w %Aw' (w' extends w into the future) =>
%Et (t > now ^ president[w',t](Elmo) ^ ~president[w,t-1](Elmo))

Notice that "will become president" requires that Elmo becomes
president in EVERY accessible world, whereas "can become
president" requires only that he becomes president in SOME
accessible world. The other big difference is the
accessibility relation that describes how w' is like w.

10. (a) The entire parse. As a rule of thumb, as a constituent gets
larger, its inside probability goes down.[*]

[*] Note: This would be strictly true for the VITERBI INSIDE
probability. The most probable derivation tree of the entire
sentence can never be more probable than one of its subtrees,
because it is a more specific event. (Its probability is the

subtree's probability times some extra rule probabilities that are <= 1.)

However, for the INSIDE probability, this argument doesn't quite hold. Rewriting the root nonterminal as the sentence (summing over all derivations) is no longer a more specific event than rewriting a nonterminal in the best derivation as the substring that it spans, because it doesn't commit to using that nonterminal at all. So for the INSIDE probability, [*] is only a rule of thumb. To see where it might go wrong, consider the grammar

```
1   S -> A A
0.9 A -> x
0.1 A -> x x
```

Then S has probability 0.18 of generating x x x, since it can do so via two equally likely trees of probability 1 * 0.9 * 0.1 each. But A has probability only 0.1 of generating x x. So in the best parse, the S node has inside probability of 0.18 but the A node that covers x x has even smaller inside probability of 0.1.

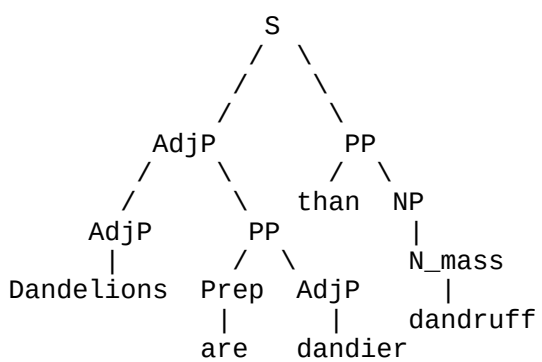
(b) The verb subconstituent. As a rule of thumb, as the material outside the constituent gets larger, its outside probability goes down.[**]

So we expect the answer to be a single-word constituent. Because the verb in this case is a common one, the unlikely material in the sentence is outside the verb, and so the outside probability of the verb will be low: the outside probability must account for several rare words as well as choice to add an optional PP.

[**] Note: Again, this would be strictly true for the VITERBI OUTSIDE probability, but for the OUTSIDE probability it is only a rule of thumb. For example, suppose we had a billion variants of the nonterminal PP, which behaved almost identically in the grammar. Call them PP, PP_1, PP_2, ... Then there would be a billion variants of the best parse shown, all almost as likely. The outside probability of the PP constituent would be extremely low because it would be required to generate the specific nonterminal PP (with probability 1e-9), whereas the outside probability of any other node would either not have to generate the PP at all (because the PP is inside the other node instead of outside it) or would be able to sum over all nonterminal variants PP, PP_1, PP_2, etc. So in that case, the PP would end up with a lower outside probability even than its subconstituents.

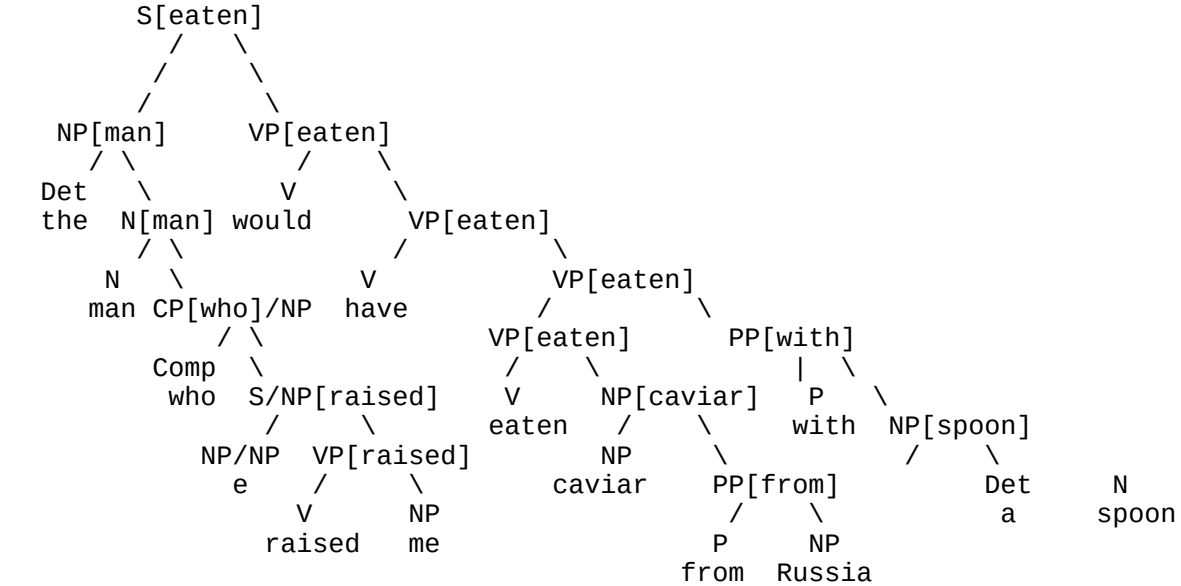
(c) The three constituents rooted at NP, N_plural, and V.

(d) Because of smoothing or errors in the grammar, there may be very low-probability parses that include AdjP -> AdjP PP at other positions. For example, this awful tree:



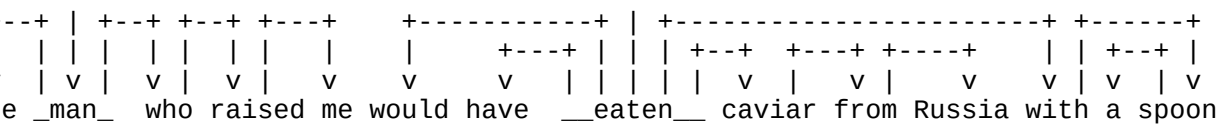
Raising the probability of the rule will raise the inside probability of S's left child, which will raise the outside probability of S's right child and its subconstituents.

11. (a)



(b) To find the parent of "caviar", we find the largest constituent headed by "caviar" (namely NP) and look at the head of *its* parent in the phrase-structure tree.

In this case, the parent is "eaten". In other words, NP[caviar] is an argument or modifier to the verb "eaten", and helps build up the meaning of S[eaten]. So we draw an arrow to "caviar" from its parent "eaten".



12. (a) Prof. Boozle is wrong because the plural NP "whose apples" is NOT the subject of the sentence. It has simply moved to the front, because that's what wh-phrases do. "Is" has also moved to the front because that's what auxiliary verbs do in questions.

The actual subject and verb are "she is," which are both singular and therefore agree. This is clear from the version where nothing has moved: "She is eating whose apples?"

We accepted any reasonable tree as an answer. Important things:

- Show how "she" is the subject of "is" and how they agree in singular number (whereas "whose apples" is plural).

- Use a slashed category to move "whose apples" to the front.

  Thus, "eating" should explicitly have a direct object.
  In "She is eating whose apples?" the direct object is "whose apples."
  In "Whose apples is she eating?" the direct object is an NP/NP that rewrites as epsilon. The /NP gap should propagate upward until it is matched by the NP "whose apples" at the start of the sentence.

As the gap propagates upwards, nonterminals along the way should also have a /NP. In particular, "Is she eating" is an S/NP: a question missing a noun phrase object. Basically it is the question "Is she eating my apples?" but with the NP "my apples" removed.

(Many of you claimed that "is she eating" was a VP, so that you could use S -> NP VP to combine it with the NP "whose apples." But "is she eating" doesn't look like an VP, act like a VP, or treat "whose apples" syntactically like its subject.)

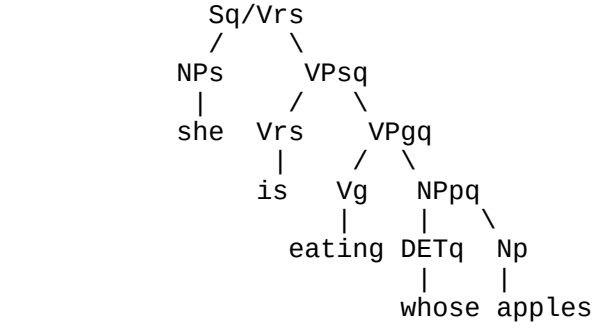
- Show how the V "is" and the VP "is eating ..." are in present tense, while the V "eating" and the VP "eating ..." are in progressive tense.

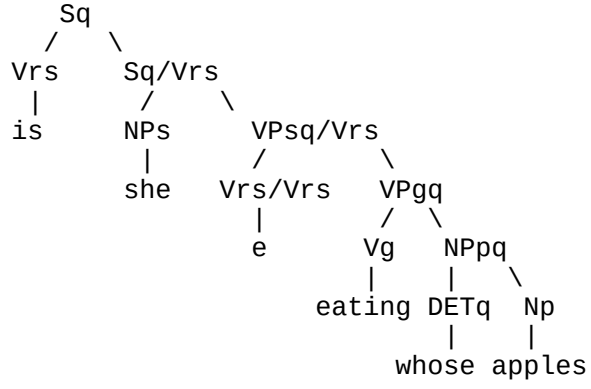
- If possible, use a slashed category with a /V gap, or some other mechanism, to move "is" to the front of the question. In other words, the question "Is she eating my apples?" should be related to the statement "She is eating my apples."

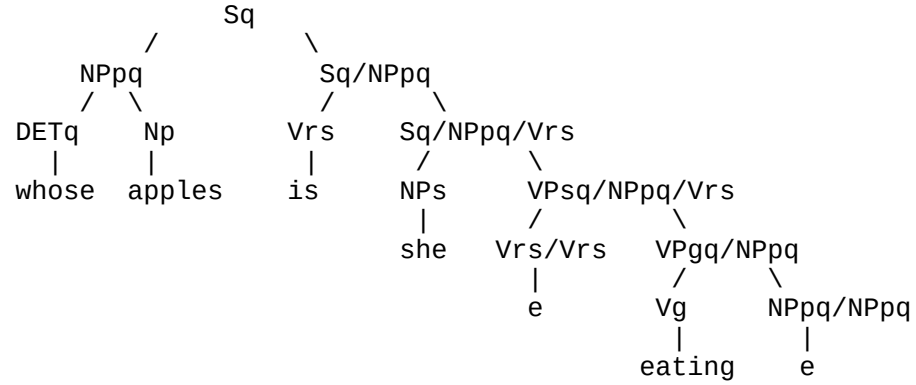
Here's an especially detailed answer that shows the features one would really use. I'll use the following lowercase suffixes to abbreviate the features:

s for num=singular
p for num=plural
r for tense=present
g for tense=progressive
q for a feature indicating that we have a question

(q stands for wh=true or inv=true or both. You actually need more than one binary feature to ensure that inversion and wh-movement both happen exactly when they're supposed to, i.e., to prevent sentences like *Whose apples she is eating? or *Is she eating whose apples? But I'm being sloppy about that here since it's not the focus of this question.)



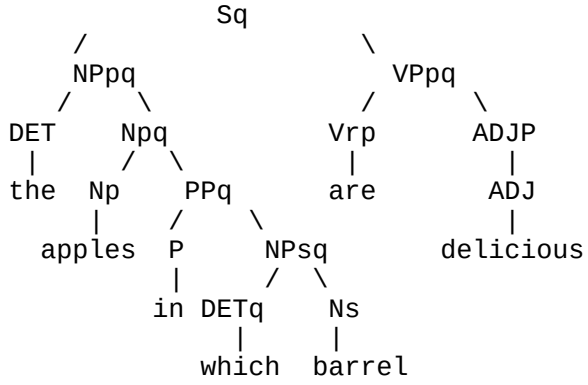


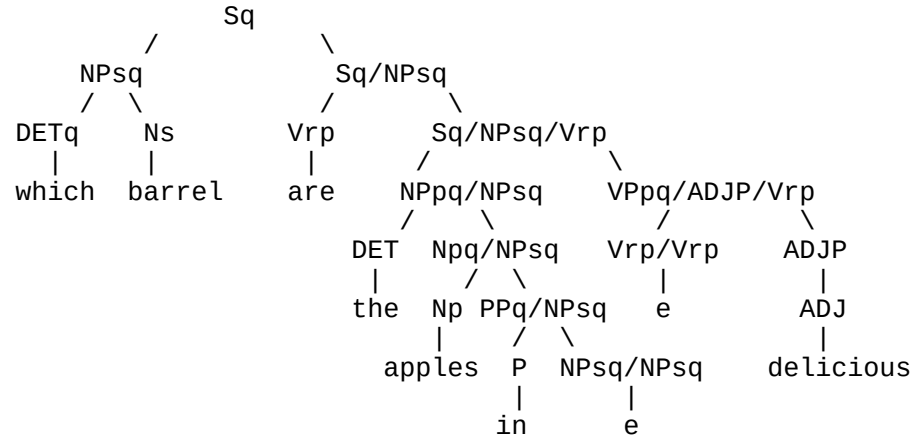


(b) The equivalent sentence is
"The apples in WHICH barrel are delicious?"

The point is that the NPq "which barrel" should be able to move to the front of the sentence, using /NPq categories, just as in the previous sentence. The surprising fact that it can't is due to a so-called "island effect" that limits what can move where.

Here are the trees before and after movement (with all features included, for your interest):





OFFICIAL ANSWERS TO THE 2002 NLP PRACTICE FINAL

1. (a) Increases; No Change.
   (b) p(d | a,b,c,e,f) > 0.99
   (c) Decreases; No Change.
   (d) No. The approximated "probabilities" will sum to more than 1.

2. Correct answers: d, f, h.
   (a) No. Not a greedy algorithm - considers all the data before making any decisions.
   (b) No. Not greedy.
   (c) No. Not greedy.
   (d) Yes. The transformation-based learning algorithm is greedy. At every step, it adds the transformation rule that most increases performance on training data. The resulting sequence of rules is not necessarily optimal. (I suspect that finding the optimal rule sequence is NP-complete.)
   (e) No. Not greedy. In particular, the choice of tag for word i can be affected by words after i as well as before i.
   (f) Yes. The competitive linking algorithm tries to pair up words from a sentence with words from a translation of that sentence. It starts by matching the word pair that matches best in isolation, and eliminating it from further consideration. This is not necessarily optimal: there may be a set of matches that is better overall, but which doesn't include the single best match.
   (g) No. OT optimization can be regarded as a greedy algorithm, because each constraint winnows the candidate set without paying attention to what the lower-ranked constraints want. However, OT defines the optimal answer to be the one produced by this greedy behavior. So the greediness is not an approximation to some other optimization problem: it is supposed to be a correct description of human language.
   (h) Yes. The decision tree is grown greedily from the top down. At each node, a question is chosen whose answer reduces the entropy of the classification as much as possible. The interaction with subsequent questions is not considered. This does not necessarily get the optimal decision tree (which is known to be NP-complete).
   (i) No. SVD is optimal: it produces the best possible rank-k approximation to the original matrix, under a least-squares error measure. If you are a linear algebra expert, you may know that the simplest SVD algorithm happens to be a greedy algorithm: it chooses the k basis vectors one at a time, at each step choosing the vector that best captures the remaining variance in the data. But it is optimal anyway.

3. e. We saw in class how under the simple permissive grammar, the number of parse trees for a sentence of length n grows exponentially: 1, 2, 5, 14, 42, 132, ... By the way, these are called "Catalan numbers":
   http://mathworld.wolfram.com/BinaryBracketing.html
   http://mathforum.org/advanced/robertd/catalan.html

4. e. If the sentence has length n, then there can be k^n possible ways just to tag those n words with preterminals. So there can certainly be exponentially many parses.

5. (a) Choose y that maximizes
   $$p(y \mid x) = p(y) * p(x \mid y) / p(x)$$
   $$= p(y) * p(x \mid y) / \sum_i p(y_i) * p(x \mid y_i)$$
   (b) Construct the machine [X .o. XY .o. WY].l
   or [WY .o. XY.i .o. X].u and find its best path.

   (c) Best answer (smallest possible machine):

257 states for the different one-letter contexts (256 characters plus BOS).

The state reached by reading a 0 character is a final state with no arcs, but each of the other 256 states has 256 arcs, for a total of 256*256 = 65536.

(d) Best answer (smallest possible machine):

We only need one final state, reached by reading a 0 character. We also need one start state at the beginning of the sentence. Any other bigram context consists of a non-0 character (or BOS) followed by a non-0 character, so there are 256*255 of them. So the total number of states is 256*255+2 = 65282.

The final state has no arcs. Any non-final state can be followed by any of the 256 characters, so it has 256 arcs, for a total of 65281*256 = 16711936 arcs.

Note - in practice, many of these arcs correspond to trigrams that have never been observed. It's possible to get away without explicitly storing these arcs.

6. Correct answers: c, d.
   (a) No: it's only 3-4 saccades per second on average.
   (b) No: people are capable of both modes.
   (This was discussed in class the year the exam was given.)
   (c) Yes: eye-tracking experiments were major evidence against multi-pass theories of comprehension. People seem to integrate all kinds of information incrementally as they hear a sentence.
   (d) Yes. We saw several examples on the videotape in class.
   (e) No: this used to be the case, but head-mounted eye trackers are now available and have enabled these experiments.
   (f) No: the previous dominant paradigm was self-paced reading.

7. The new centroid of A is a weighted average of the six points,

$$\frac{(1,1)*a + (1,2)*b + (1,3)*c + (2,4)*d + (3,4)*e + (4,4)*f}{a+b+c+d+e+f}$$

where each point is weighted by its probability of coming from cluster A rather than cluster B. These weights are denoted a,b,c,d,e,f above.

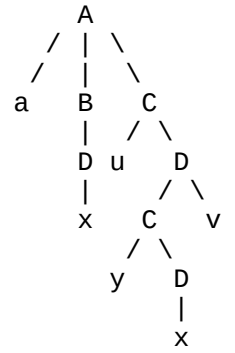
For example the first point (1,1) has probability a = 0.165/(0.165+0.001) of coming from cluster A rather than cluster B.

(Formally, that expression computes p(A | point) from p(point | A) and p(point | B), using Bayes' Theorem and the modeling assumption that the clusters have equal prior probabilities: p(A)=p(B).)

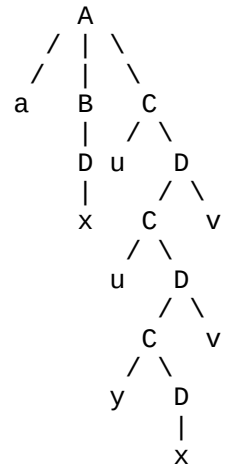
To write it all out, the x coordinate of the new centroid of A is the quotient

$$\begin{aligned}
&1 * 0.165/(0.165+0.001) \;/\; 0.165/(0.165+0.001) \\
&+ 1 * 0.100/(0.100+0.004) \;/\; + 0.100/(0.100+0.004) \\
&+ 1 * 0.044/(0.044+0.008) \;/\; + 0.044/(0.044+0.008) \\
&+ 2 * 0.008/(0.008+0.044) \;/\; + 0.008/(0.008+0.044) \\
&+ 3 * 0.004/(0.004+0.100) \;/\; + 0.004/(0.004+0.100) \\
&+ 4 * 0.001/(0.001+0.165) \;/\; + 0.001/(0.001+0.165)
\end{aligned}$$

8. The second tree must be adjoined into the first tree.
   The only solution is



unless the second tree is adjoined multiple times, giving e.g.



9. There are many possible answers ...
   (a) bank
   (b) severer
   (c) Time flies.
       It is good to eat.
   (d) A woman gives birth every 15 minutes.
       All cows are not black.

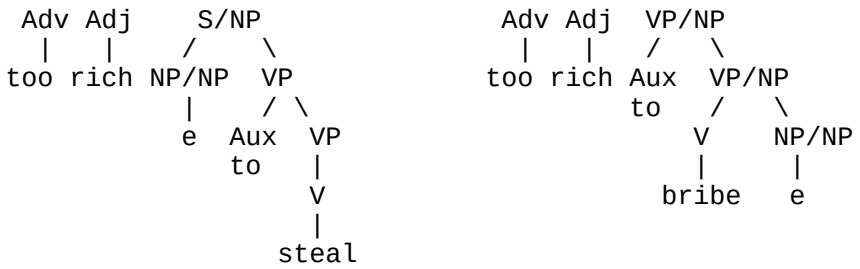
10. Factor[sem=1] -> a
    Factor[sem=1] -> b
    Factor[sem=1] -> c
    Factor[sem=1] -> d
    Factor[sem=2] -> (Expr)
    Factor[sem=star(1)] -> Factor *

    Term[sem=1] -> Factor
    Term[sem=concat(1,2)] -> Term Factor

    Expr[sem=1] -> Term
    Expr[sem=plus(1,3)] -> Expr + Term

    START[sem=1] -> Expr

11. The key point here is that Bloomberg is the subject of "steal,"
    but the object of "bribe." This affects where the /NP gap goes.
    Note also that "steal" is being used intransitively, while
    "bribe" is being used transitively. We didn't much care how
    you handled the other details.



12. Correct answers: a, d.

    (a) True. In EM generally, 0 probabilities stay 0 as the algorithm proceeds. If the initial model says that X can never happen, then X will never be allowed to happen in the reconstruction of hidden data according to that model. So a new model estimated from that reconstructed hidden data will say that X has probability 0. (Smoothing would fix this, but the basic forward-backward or EM algorithm is defined without smoothing.)
    (b) False. Even if the transition probabilities such as p(C | H) are equal, the algorithm can still learn as long as the emission probabilities such as p(3 | H) are unequal. It's only if *everything* is symmetric that we have a symmetry-breaking problem.
    (c) False. Which local maximum is found by EM depends on the initial model. One of those local maxima is in fact the global maximum, and the system will find it if the initial model is close enough to it.
    (d) True. The EM algorithm tries to maximize the likelihood of training data (or equivalently minimize its perplexity). But that may overfit the training data. It provides no guarantees about test data.
    (e) False. The reestimation step of the EM algorithm always increases the likelihood on training data, up to a local maximum. It never decreases the training likelihood (i.e., increases perplexity).

13. (a) Part of speech tagging.
    (b) Viterbi algorithm.
    (c) states = tags
        days = words
        dollars = costs (negative log probabilities)
    (d) A state of the trellis corresponds to being at a particular festival on a particular day. Let mu(y) represent the worth of the optimal path to state y in the trellis.

        For each state y, you must compute
            mu(y) = (maximum over all arcs x -> y
                        of (mu(x) - (cost of getting from x to y)))
                    + benefit of being in state y

        So each state y requires an addition ("maximum ... + benefit") and each arc x -> y requires a subtraction ("mu(x) - cost ..."). The trellis has 50*30 states, since you can be in any of 50 places on each of the next 30 days. It has 1*50 + 50*50*29 arcs, since you can go from Baltimore to anywhere on the first day, and you can go from anywhere to anywhere on the remaining 29 days. So the total number of additions is 50*30 + 1*50 + 50*50*29 = 74,050.

14. p(VP[head=swim] -> VP[head=swim] PP[head=to] | VP[head=swim])
    ---
    p(VP[head=watch] -> VP[head=watch] PP[head=to] | VP[head=watch])

    (This ratio is probably greater than 1, since you're more likely to "swim to ..." than to "watch to ..." In other words, the first tree has higher probability under the head-lexicalized model.)

15. (a) Many possible answers! Here are some.

Answer 1 (a general technique):

First we define U that will transduce "(foo)" to "foo":

```
define U [L .x. 0] C* [R .x. 0] ;
```

Now T simply applies U to a maximal set of substrings:

```
define T ~$[U.u] [U ~$[U.u]]* ;
```

Note that the untransduced substrings are required to match ~$[U.u], which means that they don't contain anything that U knows how to transduce.

(Note: It happens that ~$[U.u] is equivalent to \L* \R*, so you could substitute the latter for the former.)

More generally, instead of applying U to a maximal set of substrings as with ->, we could apply it to a set of substrings identified using the @-> operator:

```
define T     [ U.u @-> %{ ... %} ]   # bracket substrings to replace
         .o. ?* [ %{:0 U %}:0 ?* ]   # nondeterministically replace some
         .o. ~$%{ ;                  # make sure we replaced all
)
```

Answer 2:

This answer places brackets around instances of L C* R, and then deletes "{ L" and "R }" everywhere.

```
define T     [ L C* R -> %{ ... %} ]    # insert brackets
         .o. [ %{ L | R %} -> 0 ] ;     # delete brackets and parens
```

Note the use of the -> operator to bracket a *maximal* set of substrings that match L C* R. This is very much like answer 1. Directed replacement operators such as @> or @-> will also do the trick in this case.

A cute variant notation is inspired by the Porter stemmer extra credit problem on HW4. This inserts special Delete and Backspace symbols into the original string, and then processes them:

```
define T     [ L C* R -> Delete ... Backspace ]
         .o. [ Delete ? | ? Backspace -> 0 ] ;
```

Answer 3:

Similar to answer 2, this one puts the brackets inside the L and R instead of outside.

```
   define T     [ C* @-> %[ ... %] // L _ R ]
            .o. [L %[ | %] R -> 0] ;
```

Answer 4:

Or instead of placing curly brackets around L C* R, you can turn the L and R into curly brackets, and then delete them. This solution is not as general, but it's interesting because it uses nondeterminism.

```
define T     [ ?* [ [ L : %{ ] C* [ R : %} ] ?* ]* ]  # nondeterministically turn some L and R into brackets
         .o. ~$[L (C-%{-%})* R]                        # make sure you've done so everywhere
```

```
                .o. [ %{ | %} -> 0 ] ;                    # now delete the brackets
```

(b) define A [T .o. T .o. T .o. T .o. T .o. C*].u ;

The transducer [T .o. T .o. T .o. T .o. T .o. C*] applies T five times in a row and then checks whether the result is free of parentheses (i.e., matches C*).

A is defined to be the upper language of this -- the set of all parenthesized strings that would survive such a process. Since we are only taking the upper language, the process is not actually carried out, except hypothetically!

16.
```
Frank has really been building houses:  tense=pres
Frank                                :  num=sing
      has really been building houses:  tense=pres, num=sing
      has                            :  tense=pres, num=sing
          really been building houses:  tense=perf, num=any
                 been building houses:  tense=perf, num=any
                 been                :  tense=perf, num=any
                      building houses:  tense=prog, num=any
                      building       :  tense=prog, num=any
                             e houses:  num=plural
                             e       :  num=plural
                               houses:  num=plural
```

Note: The num feature for the lower VPs wasn't discussed in class, and we gave credit for any answer on these.

The best answer is probably to say num=any, as shown above. Only the tensed verb "has" shows any preference for a singular or plural subject. The lower VPs don't seem to have a preference. They can combine with either singular "has" or plural "have":

Frankie and Johnnie have really been building houses.

And if they can combine directly with subjects, they don't seem to care whether they're singular or plural subjects:

Frank building houses is a sight for sore eyes.
Frankie and Johnnie building houses is a sight for sore eyes.

17. (a) recall = 104/310, precision = 104/887

(b) recall = 40/310, precision = 40/75

(c) The formula we need is Bayes' Theorem:

```
p(compling | feats) = p(compling & feats) / p(feats)
                    = p(compling & feats)
                      / (p(compling & feats) + p(other & feats))
```

We have

```
p(compling) = 310/7M
p(has NLP=no | compling) = 206/310
p(has disambig=yes | compling) = 40/310
p(compling & feats) = 310/7M * 206/310 * 40/310

p(other) = about 1
p(has NLP=no | other) = about 1
p(has disambig=yes | other) = 35/7M
p(other & feats) = 1 * 1 * 35/7M

p(compling & feats) / (p(compling & feats) + p(other & feats))
  = 206*40/310 / (206*40/310 + 35)
This is < 0.5 since 206*40/310 ~= 27 < 35
```

18. All of the features here are binary features: they are present or they're not. So the feature vector f(x,c) consists of 1's and 0's. So when you compute the score of (x,c) by taking the dot product of the feature vector with the weight vector, it is equivalent to adding up the weights of just the features that are present.

(x,compling) has the first, third, fourth features shown in the
first column of the problem. Their weights are 0, -1, -3.
So score of (x,compling) = 0-1-3 = -4
So p(x,compling) = 1/Z * exp(-4)

(x,other) has the first, third, fourth features shown in the
second column of the problem. Their weights are 14, 0, -8.
So score of (x,other) =14-8=6
So p(x,other) = 1/Z * exp(6)

Therefore p(compling | x) = p(x,compling) / (p(x,compling)+p(x,other))
                          = exp(-4) / (exp(-4)+exp(6))
                          = 1 / (1+exp(10))
                         ~= 0

19. Several possible answers here. You could talk about decision
lists, decision trees, Naive Bayes classifiers, maximum entropy
classifiers, or a vector-space method like k-nearest-neighbor.

20. Correct answers: a, b.
(a) True, as can be shown by expanding the notation p(A|B)
into p(A,B)/p(B). This was a problem on HW2.
(b) Still true, for essentially the same reason.
(c) False. We're replacing c(w[i-1],w[i]) in the numerator
with c(w[i-1],w[i])+lambda*p(w[i]). In the reversed
computation, we'd be replacing it with
c(w[i-1],w[i])+lambda*p(w[i-1]).
(d),(e) False. As with (c), these are backoff methods.

21. (a) p(time flies like Superman)
(b) (10+3)/(15+6+1+10+3) = 13/35
(c) .6*.03*.04 + .2*.05*.02 = 0.00092
(d) .2*1*.07*.01 = .00014
(e) The relationship is (b) = (c)*(d)/(a)
(f) Enumerating the parses would take exponential time,
since there might be exponentially many parses (see question 3).
Computing all the alpha and beta probabilities only takes O(n^3)
time, because it takes advantage of the way the parses share
constituents with one another in the parse chart representation.

22. (a) %x E%e past(e), act(e,marriage), marrier(e,x), victim(e,Homer)

where as in HW3,
% = "lambda"
E% = "there exists"

(b) %y %x E%e past(e), act(e,marriage), marrier(e,x), victim(e,y)

(c) married Homer, from (a):
  %x E%e past(e), act(e,marriage), marrier(e,x), victim(e,Homer)
hopefully married Homer:
  %z hope(Speaker, E%e past(e), act(e,marriage), marrier(e,z),
                       victim(e,Homer))
hopefully:
  %f %z hope(Speaker, f(z))

(d) The semantic term doesn't really make any sense.
First, the variable e is not bound by anything -- it is
not related to the "E%e" in part (a).
Second, it doesn't make sense to write
  f ^ manner(e,hopeful)
since f is going to be a lambda term (from part (a)).
Only predicates can be conjoined with ^, not lambda terms.

(e) The problem is that "married Homer" has the semantics %x E%e. We can't attach this variable e to the variable in manner(e,hopeful).

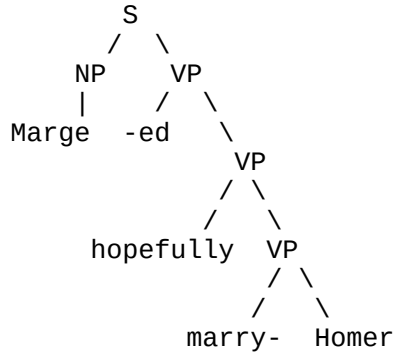
So we need to change the semantics of verb phrases. "married Homer" should be %e %x ... (or %x %e ...) rather than %x E%e ... That's what we did in class. For example, if "married Homer" is %e %x ... then the semantics of "hopefully" can be
%f %e f(e) ^ manner(e,hopeful)

We would still like E%e to show up in the final meaning of the entire sentence. Our approach in class was to make -ed convert %e to E%e:

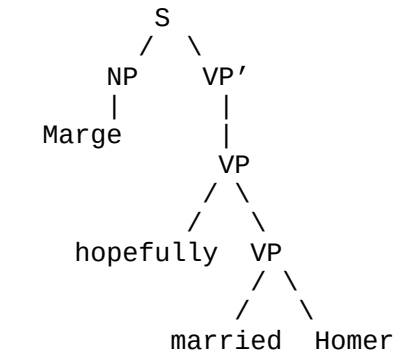
```
marry Homer             %e ...
married Homer           %e ...

hopefully marry Homer   %e ...
hopefully married Homer E%e ...
```

But this was awkward. It meant that we needed to use a structure like this with the -ed suffix way up high, so that it could be a function that applied to "hopefully marry Homer." (Chomskian linguists do assume such a structure!)



Another approach would be to assume a structure like this:

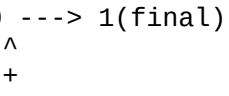


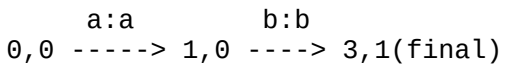
where the VP has semantics %e ...
but the VP' has semantics E%e ...
thanks to the following rule:
VP'[sem=E%e 1(e)] -> VP

23. (a) The construction was shown on a slide in lecture 17.

(b)

b



(c)


24. Correct answers: d.
    (a) False. This is dangerous with a fixed threshold; you want to lower the threshold upon failure (the iterative deepening method).
    (b) False. Doesn’t really make sense to talk about the probability of a or A. (The only reasonable way to interpret it is that one should drop low-probability rules, which isn’t a great method of pruning. Such rules are presumably in the grammar because they will be useful in some contexts.)
    (c) False. This changes the result of the parser, and has no effect on speed.

25. b. I love saying "Xhosa."