LING185A, Take-home final exam

Due date: Wed. 3/22/2017, 6:00pm

Instructions

Download `ProbCFG_Stub.hs` and `Util_Stub.hs` from the website, and rename these to `ProbCFG.hs` and `Util.hs`. You will need to submit modified versions of these files. You should also download `ProbFSAv2.hs`, but you should not modify this file.

This exam has three sections:

- For Section 1 you will write some code in `Util.hs`. You will need to submit this file online.
- For Section 2 you will write some code in `ProbCFG.hs`. You will need to submit this file online.
- For Section 3 you will answer some written (non-code) questions. You will need to submit a hard copy in person, either
  - to the front desk of the Linguistics Department main office (Campbell Hall 3125) during business hours, or
  - to my office (Campbell Hall 3103K) between 5:00pm and 6:00pm on Wednesday 3/22.

You are allowed to consult your notes from lectures, your answers to the course assignments, and anything posted on the course CCLE site. You may also consult any additional sources which have been *explicitly* mentioned in any of these materials (this includes the Manning & Schütze and Jurafsky & Martin textbooks mentioned in the syllabus), but this should not really be necessary and you should think carefully about whether you are wasting time by looking further afield like this.

You must not receive any assistance on this exam from anyone except the instructor or TA, and you must not provide any such assistance to anyone else.

Overview

The code for the table-based implementation of PFSA probability calculations that we saw in class on 3/15 is in `ProbFSAv2.hs` ("version 2" just to distinguish it from last week's file). This code calls upon certain general utility functions in `Util.hs` for which there are only stubs in the current version of this file. Your task in Section 1 will be to fill in these stubs so that `ProbFSAv2.hs` works as desired. You should not change anything in `ProbFSAv2.hs` itself.

The third file, `ProbCFG.hs`, contains stubs for the analogous PCFG probability calculations. Your task in Section 2 will be to fill in these stubs, guided by the provided implementations in `ProbFSAv2.hs`. Because of the similarities between the way the probability calculations are carried out for the two types of grammars, this code will also call upon the functions in `Util.hs`.

1 Warm up: Utility functions (6 point)

Fill in the stubs for the following functions in `Util.hs`.

A. Write the function `allPairs :: [a] -> [b] -> [(a,b)]` which produces all possible pairs containing one element from the first argument list and one element from the second argument list. The **order is important**: if `x` precedes `y` in the first argument list, then all pairs whose first element is `x` must precede all pairs whose first element is `y` in the result list.

```
*Util> allPairs [2,3] ["hello","world"]
[(2,"hello"),(2,"world"),(3,"hello"),(3,"world")]
*Util> allPairs [True,False,True] [7]
[(True,7),(False,7),(True,7)]
*Util> allPairs [[],["a"],["a","d"]] [10,20]
[([],10),([],20),(["a"],10),(["a"],20),(["a","d"],10),(["a","d"],20)]
```

B. Write the function `maxOver :: (a -> Double) -> [a] -> (Double,a)` such that `maxOver f xs` picks out the element of `xs` which is mapped to the largest number by `f`, and produces a pair containing that largest number and the picked-out element. If there are "ties", then the result should be the element that occurs earliest in the provided list. The result should be `undefined` if the given list is empty.[1]

```
*Util> maxOver (\x -> x * 10) [3,5,4]
(50.0,5.0)
*Util> maxOver (\x -> x * x) [3,5,4]
(25.0,5.0)
*Util> maxOver (\x -> x * x) [3,5,4,-6,6]
(36.0,-6.0)
*Util> maxOver (\x -> x * x) []
*** Exception: Prelude.undefined
```

C. Write the function `updateForAll :: (t -> c -> t) -> t -> [c] -> t` which takes three arguments: an updating function of type `t -> c -> t`, an initial value of type `t`, and a list of update-triggers of type `c`. The updating function produces a new value of type `t` from an old value of type `t` and a single update-trigger of type `c`. The result of `updateForAll f z xs` should be the result of successively updating the initial value `z` via the function `f` on the basis of each of the update-triggers in the list `xs` **in the order** that they appear there.

```
*Util> updateForAll (\x -> \y -> 10 * x + y) 1 [2,3,4]
1234
*Util> 10 * (10 * (10 * 1 + 2) + 3) + 4
1234
*Util> updateForAll (\x -> \y -> x ++ " " ++ y) "hello" ["one","two","three"]
"hello one two three"
*Util> updateForAll (\x -> \y -> y ++ " " ++ x) "hello" ["one","two","three"]
"three two one hello"
*Util> replicate 3 'a'
"aaa"
*Util> replicate 5 'a'
"aaaaa"
*Util> updateForAll (\s -> \n -> s ++ " " ++ replicate n 'a') "hello" [2,3,4]
"hello aa aaa aaaa"
```

[1]A few of you stumbled across a handy shortcut for doing this in certain cases that worked in last week's assignment, by using the built-in function `maximum` on a list of pairs. This won't work here because the type `a` might not be ordered, and if it's not then the type `(Double,a)` is not ordered either. You need to write this as a good old-fashioned recursive list function.

```
*Util> updateForAll (\s -> \n -> replicate n 'a' ++ " " ++ s) "hello" [2,3,4]
"aaaa aaa aa hello"
```

Peeking ahead a little bit: now everything in `ProbFSAv2` for efficiently calculating backward and Viterbi probabilities for a PFSA should be working. For example, the backward probability for the output `["a","d"]` for state 10 can be found like this:

```
*ProbFSAv2> Map.findWithDefault 0 (["a","d"],10) (buildTableBackward pfsa3 ["a","d"])
0.20500000000000002
```

2 Inside and Viterbi calculations for PCFGs (8 points)

In this section you might find the built-in functions `take` and `drop` useful. Both have type `Int -> [a] -> [a]`. The function `take` produces a sublist of the specified length starting from the beginning of the original list; `drop` produces that part of the original list that is left out by a corresponding call to `take`.

```
*ProbCFG> take 2 ["one","two","three","four","five"]
["one","two"]
*ProbCFG> take 4 ["one","two","three","four","five"]
["one","two","three","four"]
*ProbCFG> drop 2 ["one","two","three","four","five"]
["three","four","five"]
*ProbCFG> drop 4 ["one","two","three","four","five"]
["five"]
```

And you might also find it useful that we can write things like `[0..10]` for lists of integers:

```
*ProbCFG> [0..10]
[0,1,2,3,4,5,6,7,8,9,10]
*ProbCFG> [0 .. (length "hello")]
[0,1,2,3,4,5]
*ProbCFG> [1 .. (length "hello")]
[1,2,3,4,5]
```

Representing PCFGs

We will consider probabilistic context-free grammars (PCFGs) of a certain restricted sort.[2] There are only two kinds of rules allowed: binary transition rules of the form '$A \rightarrow B\ C$' where A, B and C are all nonterminals, and ending rules of the form '$A \rightarrow w$' where A is a nonterminal and w is a word. (So there are no unary rules.) Each rule has an associated probability, and for each nonterminal, the set of rules that have that nonterminal on their left-hand side must have probabilities that sum to one.

Here is an example grammar that is encoded as `pcfg1` in `ProbCFG.hs`.

S → NP VP	1.0	NP → NP PP	0.4
PP → P NP	1.0	NP → astronomers	0.1
VP → V NP	0.7	NP → ears	0.18
VP → VP PP	0.3	NP → saw	0.04
P → with	1.0	NP → stars	0.18
V → saw	1.0	NP → telescopes	0.1

[2]Grammars that conform to this pattern are said to be in "Chomsky Normal Form" (CNF). Any context-free grammar can be converted to an equivalent one in CNF that produces the same set of strings.

Exactly how these rules are encoded in the `ProbCFG` type is not important (although it's not complicated, either). What matters is that the probabilities of the binary transition rules can be retrieved using the function `trProb :: ProbCFG -> Cat -> (Cat,Cat) -> Double`, and that the probabilities of the ending rules can be retrieved using the function `endProb :: ProbCFG -> Cat -> String -> Double`. Note that these functions return zero when there is no corresponding rule shown in the table above.

```
*ProbCFG> trProb pcfg1 S (NP,VP)
1.0
*ProbCFG> trProb pcfg1 VP (VP,PP)
0.3
*ProbCFG> trProb pcfg1 VP (VP,NP)
0.0
*ProbCFG> endProb pcfg2 NP "stars"
0.18
*ProbCFG> endProb pcfg2 V "saw"
1.0
*ProbCFG> endProb pcfg2 VP "saw"
0.0
```

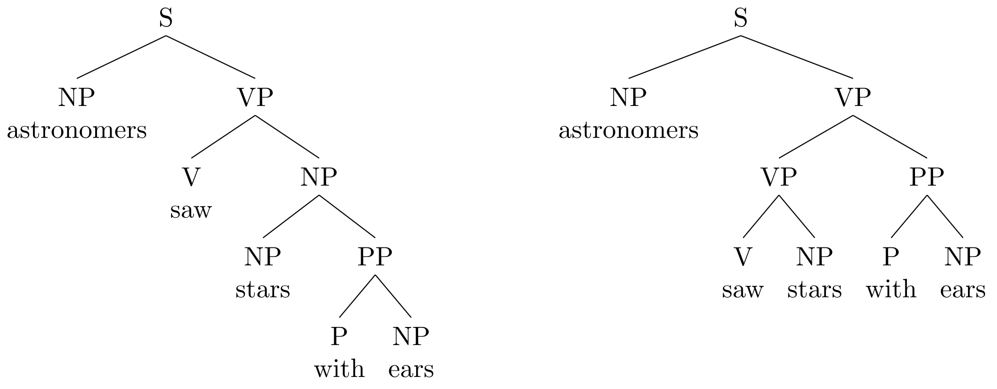
In addition to these rule probabilities, there is also a collection of starting probabilities, one for each category, which must sum to one. These can be retrieved from a grammar using the `startProb` function. In all the grammars here, the starting probabilities are simply 1.0 for the category S and 0.0 for all other categories. And finally, we can get a list containing all the categories used by a particular grammar via the `allCats` function.

We interpret the probability attached to the rule 'VP → V NP', for example, as the probability that some node will have V as the label of its left daughter and have NP as the label of its right daughter *given that* this node has VP as its own label. And we interpret the probability attached to the rule 'NP → stars', for example, as the probability that some node will be a leaf node with the word 'stars' *given that* this node has NP as its own label. If we take C_α to be the label of the node at address α, and take W_α to be the word(s) produced underneath the node at address α, then what the probabilities attached to these two rules are saying is:

$$\Pr(C_{\alpha 0} = \text{V}, C_{\alpha 1} = \text{NP} \mid C_\alpha = \text{VP}) = 0.7$$
$$\Pr(W_\alpha = \text{stars} \mid C_\alpha = \text{NP}) = 0.18$$

Notice that there are two distinct trees for the sentence 'astronomers saw stars with ears' according to this grammar:

(1)



It follows from the way we interpret the rules' individual probabilities that the probability of the first tree, with the PP modifying the NP, is

$$1.0 \times 0.1 \times 0.7 \times 1.0 \times 0.4 \times 0.18 \times 1.0 \times 1.0 \times 0.18 = 0.0009072$$

and the probability of the second tree is

$$1.0 \times 0.1 \times 0.3 \times 0.7 \times 1.0 \times 0.18 \times 1.0 \times 1.0 \times 0.18 = 0.0006804$$

(where in each case I have not bothered to write the starting probability of 1.0 for S). So the total probability of an S yielding this sequence of words, i.e. the inside probability, is

$$\Pr(W_\alpha = \text{astronomers saw stars with ears} \mid C_\alpha = \text{S}) = 0.0009072 + 0.0006804 = 0.0015876$$

and the probability of the *best* tree for this sequence of words with root label S is 0.0009072.

I have also defined `pcfg2` as a grammar which is like `pcfg1` but has the probabilities of the two rules for expanding VP nodes swapped. In `pcfg2`, the second tree, where the PP modifies the VP, has a higher probability than the first tree.

Naive calculation of inside probabilities

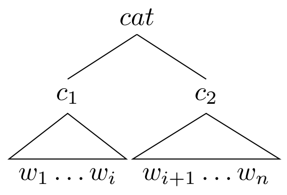
Recall that *inside probabilities* in a PCFG correspond closely to *backward probabilities* in PFSA. Backward probabilities can be computed like this:

(2)
$$\texttt{probBackward}\ [\,]\ st = \texttt{endProb}\ st$$
$$\texttt{probBackward}\ w_1 \dots w_n\ st = \sum_{next \in \texttt{allStates}} \Big[(\texttt{trProb}\ st\ next) \times (\texttt{emProb}\ (st, next)\ w_1) \times (\texttt{probBackward}\ w_2 \dots w_n\ next)\Big]$$

And similarly, inside probabilities can be computed like this:

(3)
$$\texttt{probInside}\ w\ cat = \texttt{endProb}\ cat\ w$$
$$\texttt{probInside}\ w_1 \dots w_n\ cat = \sum_{c_1 \in \texttt{allCats}} \sum_{c_2 \in \texttt{allCats}} \sum_{1 \le i < n} \Big[(\texttt{trProb}\ cat\ (c_1, c_2)) \times (\texttt{probInside}\ w_1 \dots w_i\ c_1) \times (\texttt{probInside}\ w_{i+1} \dots w_n\ c_2)\Big]$$

In that last equation we're summing over (i) all possible left daughter categories c_1, (ii) all possible right daughter categories c_2, and (iii) all possible lengths i for the part of $w_1 \dots w_n$ that gets covered by the left daughter. (The length of this part determines the length of the part that gets covered by the right daughter.) This is the picture to have in mind:



A. Write the function `naiveInside :: ProbCFG -> [String] -> Cat -> Double` which computes inside probabilities in the "obvious", recursive manner based on the equations above.[3] This function should be analogous to `naiveBackward` in `ProbFSAv2.hs`. The result can be `undefined` when the second argument is the empty list. You can use the `allTriples` function provided just below the stub, which (funnily enough) does for triples what `Util.allPairs` does for pairs.

```
*ProbCFG> naiveInside pcfg1 ["astronomers"] NP
0.1
*ProbCFG> naiveInside pcfg1 ["astronomers"] V
0.0
*ProbCFG> naiveInside pcfg1 ["saw"] V
1.0
*ProbCFG> naiveInside pcfg1 ["saw"] NP
4.0e-2
*ProbCFG> naiveInside pcfg1 ["saw","stars"] VP
0.126
*ProbCFG> naiveInside pcfg1 ["astronomers","saw","stars"] S
1.26e-2
```

This `naiveInside` function works fine for small sentence-chunks, e.g. up to three words, but with larger sentence-chunks things quickly get out of hand due to the way this implementation keeps *re-computing* certain inside probabilities. (Think about how many separate times something like `endProb V "saw"` is getting evaluated.) On the machine where I'm running things, evaluating

```
naiveInside pcfg1 ["saw","stars","with","ears"] VP
```

takes a noticable amount of time, and I didn't wait long enough to see how much time

```
naiveInside pcfg1 ["astronomers","saw","stars","with","ears"] S
```

would take.

Table-based calculation of inside probabilities

In `ProbFSAv2.hs` you'll find, in addition to the `naiveBackward` function corresponding to the `naiveInside` function that you've written, an implementation of the table-based approach to calculating backward probabilities. The relevant top-level function is `buildTableBackward`; see also `fillCellBackward`, `cellsToFill` and `chunks`. I've provided a handy function `Util.printMap` for inspecting tables of the sort constructed by `buildTableBackward`:

```
*ProbFSAv2> Util.printMap (buildTableBackward pfsa3 ["a","d"])
(([],40),0.5)
((["a","d"],10),0.20500000000000002)
((["d"],20),0.2)
((["d"],30),0.45)
```

Notice that we do not store probabilities of zero in the table. Due to the way we use `Map.findWithDefault` in `fillCellBackward`, if we ever look at a particular position in the table and find that it's empty we interpret that as a value of zero. This just makes the output from `Util.printMap` a bit more readable. (The code for deciding whether or not to insert a value into the table is a bit ugly and repetitive, but I didn't want to change things too much from what we saw in class.)

The general pattern that we can follow in these table-based calculations, whether computing backward probabilties with a PFSA (as is already done here for you) or inside probabilities with a PCFG (as you will need to do yourself), is this:

- There are certain parts of the sentence, called *chunks*, for which it is useful to compute re-usable intermediate values.
- Our aim is to calculate a probability for each chunk-state pair (for a PFSA) or chunk-category pair (for a PCFG). Each one of those pairs I'll call a *cell*.

[3]I think I said in class that I would provide this for you. I changed my mind. It's a good exercise.

- We begin by filling in cells for the smallest chunks. This corresponds to the base case in the naive implementations.
- We then fill in cells for progressively larger chunks, on the basis of the values found in the cells for smaller chunks. This corresponds to the recursive case in the naive implementations.

Here is a guide to understanding the differences between the way this general pattern applies to calculating backward probabilities, and the way it applies to calculating inside probabilities:

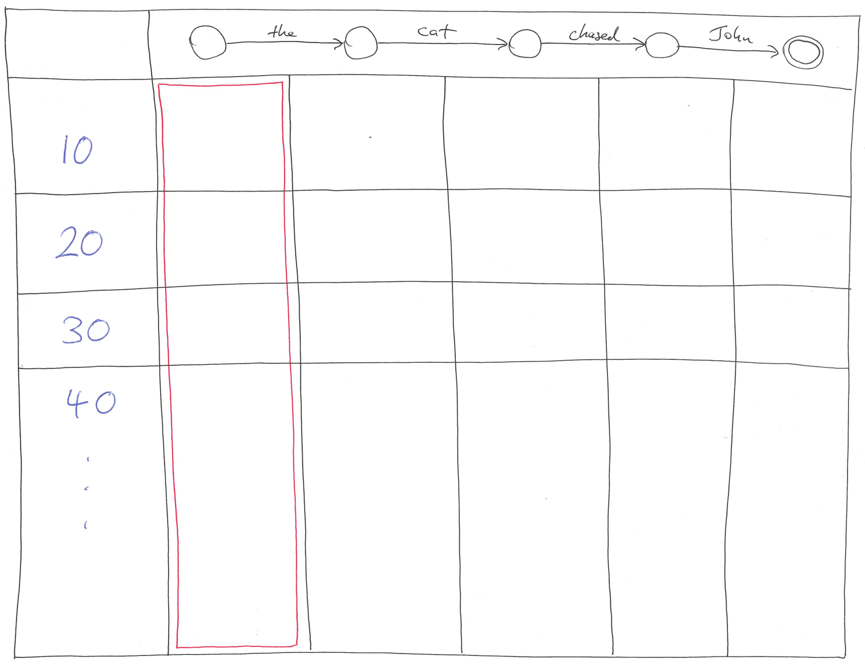
- Regarding chunks:
  - When calculating backward probabilities, the relevant chunks are all suffixes[4] of the sentence, all the way down to the empty suffix.
  - When calculating inside probabilities, you need to work out how to define the relevant chunks and their order.
- If you like thinking in terms of the drawings in Figure 1, it's useful to make a careful distinction between (i) the *table* that we are filling in according to the general pattern above, which is made up of a collection of *cells*, and (ii) the two-dimensional *grid* that we can draw on a page or blackboard, which is made up of a collection of *boxes*.
  - When calculating backward probabilities, each *box* in the two-dimensional *grid* that we can draw corresponds to one cell. (And each column in the grid corresponds to one chunk.)
  - When calculating inside probabilities, each *box* in the two-dimensional *grid* that we can draw corresponds to one chunk, and therefore corresponds to multiple cells (specifically, one cell for each category).
- Regarding the way (cells for) smaller chunks make contributions towards the values for (cells for) larger chunks:
  - When calculating a backward probability for the cell (ws, st) by summing, we must consider each state that we might transition to from st. There is no need to consider different possible ways in which subparts of ws could be generated from each such state. So if the grammar has N states, then there are N contributions to the sum in (2).
  - When calculating an inside probabilitiy for the cell (ws, cat) by summing, we must consider each pair of categories that we might transition to from cat, and also each of the different possible ways in which subparts of ws could be generated from each such pair of states. So if the grammar has N categories and there are L different ways to break up ws, then there are $N^2 \times L$ contributions to the sum in (3). Each of the L different ways to break up ws corresponds to a choice of *two contributing boxes* in a grid of the sort in Figure 1, one somewhere to the left of the box representing the chunk ws, and one somewhere underneath it; and for each choice of two contributing boxes, we must consider each pair of categories.

Now, over to you ...

B. Write the function `chunks :: [String] -> [[String]]` which produces all the chunks of the given sentence for which we want to calculate inside probabilities, in an order appropriate for the way inside probabilities can be efficiently computed. When you've done this, `cellsToFill` will provide (with the help of the `Util.allPairs` function that you have written) the list of *cells* that need to be filled, in order. These are analogous to the functions of the same names in `ProbFSAv2.hs`.

C. Write the functions `buildTableInside` and `fillCellInside`. There are analogous to `buildTableBackward` and `fillCellBackward` in `ProbFSA.hs`. All the interesting work is in `fillCellInside`: given a grammar, a partially-filled table, and a cell that remains to be filled in, this function should fill that cell in with the appropriate inside probability (and produce the appropriately updated table). This function should *not* be recursive: all inside probabilities that it needs should be drawn from the table;

[4]In the mathematical sense (not the linguistic sense): using Haskell-ish notation, `t` is a suffix of `xs` iff there is some `s` such that `s ++ t == xs`.



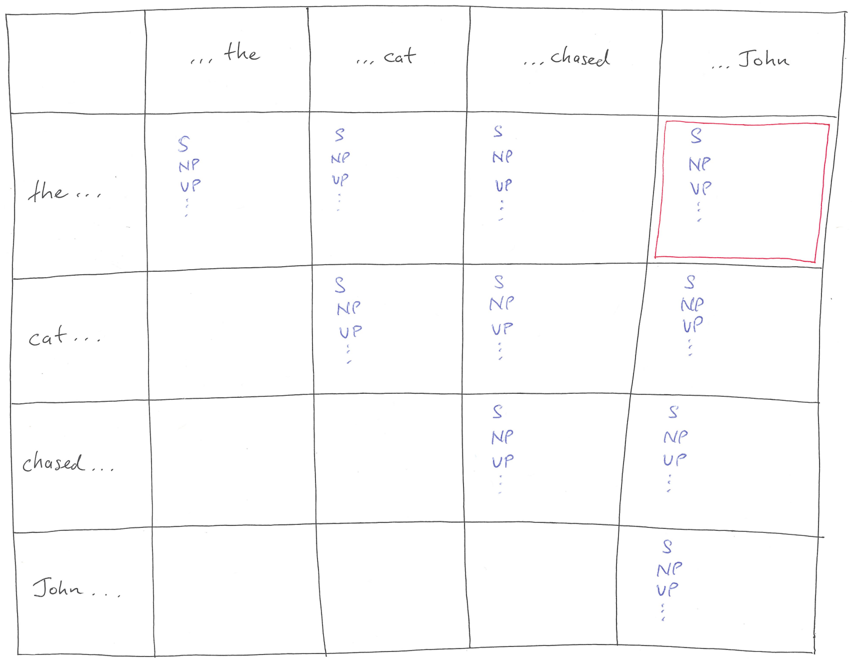


Figure 1: Generic schemas for computing probabilities for the word-sequence `["the","cat","chased","John"]`. The top grid would be used for FSA-based calculations; the bottom grid for CFG-based calculations. The *column* in the top grid that is marked in red corresponds to the complete word-sequence, and contains information about the relationship between it and the various *states* (written in blue); the *box* in the bottom grid that is marked in red corresponds to the complete word-sequence, and contains information about the relationship between it and the various *categories* (written in blue).

your use of cellsToFill in buildTableInside should ensure that this is possible. The result from fillCellInside can be undefined if the cell (its third argument) corresponds to an empty chunk, just like we said for naiveBackward.

```
*ProbCFG> Util.printMap (buildTableInside pcfg1 ["astronomers","saw","stars","with","ears"])
((["astronomers"],NP),0.1)
((["astronomers","saw","stars"],S),1.26e-2)
((["astronomers","saw","stars","with","ears"],S),1.5875999999999998e-3)
((["ears"],NP),0.18)
((["saw"],NP),4.0e-2)
((["saw"],V),1.0)
((["saw","stars"],VP),0.126)
((["saw","stars","with","ears"],VP),1.5875999999999998e-2)
((["stars"],NP),0.18)
((["stars","with","ears"],NP),1.296e-2)
((["with"],P),1.0)
((["with","ears"],PP),0.18)
*ProbCFG> Util.printMap (buildTableInside pcfg2 ["astronomers","saw","stars","with","ears"])
((["astronomers"],NP),0.1)
((["astronomers","saw","stars"],S),5.4e-3)
((["astronomers","saw","stars","with","ears"],S),1.0692e-3)
((["ears"],NP),0.18)
((["saw"],NP),4.0e-2)
((["saw"],V),1.0)
((["saw","stars"],VP),5.4e-2)
((["saw","stars","with","ears"],VP),1.0692e-2)
((["stars"],NP),0.18)
((["stars","with","ears"],NP),1.296e-2)
((["with"],P),1.0)
((["with","ears"],PP),0.18)
```

Note that this is now calculating the same values as naiveInside, but without the serious speed problems. Celebrate!

Table-based calculation of Viterbi probabilities

In addition to the table-based calculation of backward probabilities, ProbFSAv2.hs contains code for carrying out table-based calculation of Viterbi probabilities, where we choose the largest contribution rather than summing them. This is the non-naive version of what you did recursively in Assignment #9. The functions that do this are buildTableViterbi and fillCellViterbi. (Since the order in which cells' Viterbi probabilities should be calculated is the same as the order in which their inside probabilities should be calculated, the cellsToFill function is used here again.) Actually, these functions produce not only a Viterbi probability for each cell, but also a *backpointer* indicating the state that contributed the winning probability, as you saw in Assignment #9.

In Assignment #9, we wrote a function that returned a (Double,State) pair for a given cell, comprising the Viterbi probability and the backpointer together. So one can imagine implementing the table-based version of this by gradually filling values into a Map of type

```
Map.Map ([String],Cat) (Double,State)
```

but instead I've decided to use a pair comprised of two separate Maps, one for the probabilities and one for the backpointers:

```
type ViterbiTable = (Map.Map ([String],State) Double, Map.Map ([String],State) State)
```

Among other things, this allows us to simply not record a backpointer in the cases where we find a backward probability simply via endProb.

You can get your hands on the individual components of a pair with `fst` (which is just `(\(x,y) -> x)`) and `snd` (which is just `(\(x,y) -> y)`). So you can see the results of `buildTableViterbi` like this:[5]

```
*ProbFSAv2> Util.printMap (fst (buildTableViterbi pfsa3 ["a","d"]))
(([],40),0.5)
((["a","d"],10),0.189)
((["d"],20),0.2)
((["d"],30),0.45)
*ProbFSAv2> Util.printMap (snd (buildTableViterbi pfsa3 ["a","d"]))
((["a","d"],10),30)
((["d"],20),40)
((["d"],30),40)
```

So what we have is an efficient implementation of the calculation of PFSA Viterbi probabilities according to these formulas:

(4)
$$\texttt{probViterbi}\ [\,]\ st \quad = \quad \texttt{endProb}\ st$$
$$\texttt{probViterbi}\ w_1 \dots w_n\ st = \max_{next \in \texttt{allStates}} \big[(\texttt{trProb}\ st\ next) \times (\texttt{emProb}\ (st, next)\ w_1) \times (\texttt{probViterbi}\ w_2 \dots w_n\ next)\big]$$

These formulas are a minor modification of the formulas in (2) for backward probabilities. By the same logic that justified this step last week, we can apply the analogous modification to our formulas in (3) for inside probabilities to get best-analysis probabilities for our PCFG! These are often also called "Viterbi probabilities", perhaps unfortunately, but usually it's not hard to tell whether we're talking about the PFSA version of the PCFG version from the context, so I'll go ahead and use the same names for both. Specifically, if we take `probViterbi` *ws cat* to be the probability of the most likely tree yielding the words *ws* with root label *cat*, then these probabilities can be computed according to these formulas:[6]

(5)
$$\texttt{probViterbi}\ w\ cat \quad = \quad \texttt{endProb}\ cat\ w$$
$$\texttt{probViterbi}\ w_1 \dots w_n\ cat = \max_{c_1 \in \texttt{allCats}} \max_{c_2 \in \texttt{allCats}} \max_{1 \leq i < n} \big[(\texttt{trProb}\ cat\ (c_1, c_2)) \times (\texttt{probViterbi}\ w_1 \dots w_i\ c_1) \times (\texttt{probViterbi}\ w_{i+1} \dots w_n\ c_2)\big]$$

There's just one more thing: in order to actually find the best tree for a sentence we need to add backpointers that record which contribution provided the winning probability. With a PFSA, there is just one competing contribution for each state, so the backpointer just needs to identify that state. With a PCFG there is one competing contribution for each `(Cat,Cat,Int)` pair. So the relevant backpointers are triples of this type, and the type `ViterbiTable` that we will use to record both Viterbi probabilities and backpointers for PCFGs is:

```
type ViterbiTable = (Map.Map ([String],Cat) Double, Map.Map ([String],Cat) (Cat,Cat,Int))
```

Now, over to you ...

[5] Fun fact: A backpointer, in general, points "back" from bigger chunks' probabilities towards the smaller chunks' probabilities which were used to compute them. But since our table gets filled in right-to-left here, this means that our backpointers now point left-to-right! This is unfortunately confusing for PFSAs but corresponds to the directionality that makes perfect sense for PCFGs, which is what's important here, so feel free to ignore this remark.

[6] Note that multiple 'max' operations can be absorbed into one:

$$\max_{x \in X} \max_{y \in Y} \max_{z \in Z} \big[f(x,y,z)\big] = \max_{(x,y,z) \in X \times Y \times Z} \big[f(x,y,z)\big]$$

just as you probably already worked out was possible with sums:

$$\sum_{x \in X} \sum_{y \in Y} \sum_{z \in Z} \big[f(x,y,z)\big] = \sum_{(x,y,z) \in X \times Y \times Z} \big[f(x,y,z)\big]$$

D. Write the functions `buildTableViterbi` and `fillCellViterbi`. These are analogous to the functions of the same name in `ProbFSA.hs`. Again, all the interesting work is in `fillCellViterbi`, and this function should *not* be recursive. The result from `fillCellViterbi` can be `undefined` if the cell (its third argument) corresponds to an empty chunk, just like we said for `naiveBackward` and `fillCellInside`.

```
*ProbCFG> Util.printMap (fst (buildTableViterbi pcfg1 ["astronomers","saw","stars","with","ears"]))
((["astronomers"],NP),0.1)
((["astronomers","saw","stars"],S),1.26e-2)
((["astronomers","saw","stars","with","ears"],S),9.071999999999999e-4)
((["ears"],NP),0.18)
((["saw"],NP),4.0e-2)
((["saw"],V),1.0)
((["saw","stars"],VP),0.126)
((["saw","stars","with","ears"],VP),9.071999999999998e-3)
((["stars"],NP),0.18)
((["stars","with","ears"],NP),1.296e-2)
((["with"],P),1.0)
((["with","ears"],PP),0.18)
*ProbCFG> Util.printMap (snd (buildTableViterbi pcfg1 ["astronomers","saw","stars","with","ears"]))
((["astronomers","saw","stars"],S),(NP,VP,1))
((["astronomers","saw","stars","with","ears"],S),(NP,VP,1))
((["saw","stars"],VP),(V,NP,1))
((["saw","stars","with","ears"],VP),(V,NP,1))
((["stars","with","ears"],NP),(NP,PP,1))
((["with","ears"],PP),(P,NP,1))
```

E. Write the function `extractTree :: ([String],Cat) -> ViterbiTable -> StrucDesc` which uses the backpointer information in the given table (second argument) to produce the highest-probability tree for the given "cell" (first argument), i.e. the highest-probability tree with the given word-sequence along its leaves and the given category at its root. (You do not actually need to work with any probabilities here.) The result can be `undefined` if the given word-sequence is empty. The result can also be `undefined` if the provided table doesn't have all the information you need in it; a handy way to make this happen is to pass `undefined` as the first argument to `Map.findWithDefault`.

```
*ProbCFG> let ws = ["astronomers","saw","telescopes"] in
          extractTree (ws,S) (buildTableViterbi pcfg1 ws)
Binary S (Leaf NP "astronomers") (Binary VP (Leaf V "saw") (Leaf NP "telescopes"))
```

```
*ProbCFG> let ws = ["astronomers","saw","stars","with","ears"] in
          extractTree (ws,S) (buildTableViterbi pcfg1 ws)
Binary S (Leaf NP "astronomers")
         (Binary VP (Leaf V "saw")
                    (Binary NP (Leaf NP "stars") (Binary PP (Leaf P "with") (Leaf NP "ears")))
         )
```

```
*ProbCFG> let ws = ["astronomers","saw","stars","with","ears"] in
          extractTree (ws,S) (buildTableViterbi pcfg2 ws)
Binary S (Leaf NP "astronomers")
         (Binary VP (Binary VP (Leaf V "saw") (Leaf NP "stars"))
                    (Binary PP (Leaf P "with") (Leaf NP "ears"))
         )
```

So what we've built here could be described as a parser, although it's not a model that lets us ask questions about "what happens when" during incremental processing of a sentence (say, by a human being) in the way that the systems we talked about in Weeks 6 and 7 did. The approach we've implemented here is known as *chart-based parsing* or *tabular parsing*, as opposed to the other style which is known as *transition-based parsing* (or sometimes *stack-based parsing*).

If you're bored over spring break, you might like to think about how the system you've implemented here could be modified to allow retrieval of *all* the possible parses of a sentence, by recording values of type

[(Cat,Cat,Int)] instead of (Cat,Cat,Int) in the table of backpointers.

3 The big picture (6 points)

In this section, assume that we are working with grammars where the individual symbols (e.g. things emitted on an arc, or things appearing at a leaf node, etc.) are the letters {a,b,c,...,x,y,z}. So when I write 'abc', for example, this is a "word-sequence" of length three, which would be represented as `["a","b","c"]` in the way we've been doing things in Haskell.

Although I'm using probabilities as a kind of launching-off point, the important ideas here really do not relate to probabilities.

Bigram grammars

A. Suppose I have a particular probabilistic bigram grammar which assigns a probability of 0.24 to 'abxpqxcd', and assigns a probability of 0.144 to 'abxpqxpqxcd'. What probability does this grammar assign to 'abxpqxpqxpqxcd'? What probability does this grammar assign to 'abxcd'?
Hint: The probability it assigns to 'abc' is

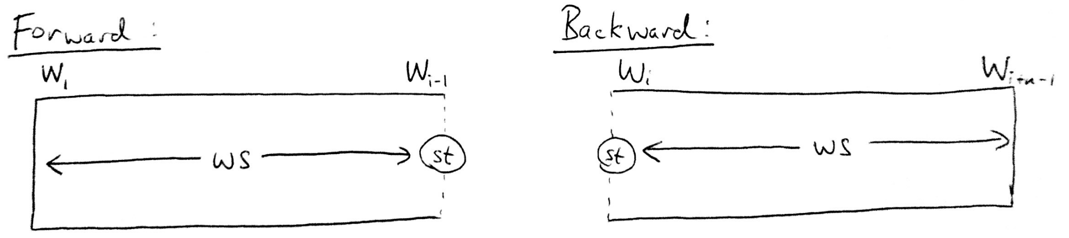
$$\Pr(W_{\text{curr}} = \text{a} \mid W_{\text{prev}} = \texttt{<s>}) \times \Pr(W_{\text{curr}} = \text{b} \mid W_{\text{prev}} = \text{a}) \times \Pr(W_{\text{curr}} = \text{c} \mid W_{\text{prev}} = \text{b}) \times \Pr(W_{\text{curr}} = \texttt{</s>} \mid W_{\text{prev}} = \text{c})$$

FSAs

Reminders:

$$\texttt{probForward}\ ws\ st = \Pr(W_1 \dots W_{i-1} = ws, S_i = st)$$
$$\texttt{probBackward}\ ws\ st = \Pr(W_i \dots W_{i+n-1} = ws, \text{end at } S_{i+n} \mid S_i = st)$$

Here are the pictures to have in mind:



B. Suppose I have a particular probabilistic FSA (which has 10 and 20 as two of its states), such that:

`probForward` 'abc' 10 = 0.2
`probForward` 'abc' 20 = 0.2
`probForward` 'def' 10 = 0.1
`probBackward` 'def' 10 = 0.3
`probBackward` 'pqr' 20 = 0.2

What strings can we conclude are assigned non-zero probabilities by this probabilistic FSA?

C. Explain why we cannot reach conclusions about the exact probabilities of the strings you gave in your answer to the previous question, whereas we could reach conclusions about exact probabilities in the first question above on bigram grammars.

CFGs

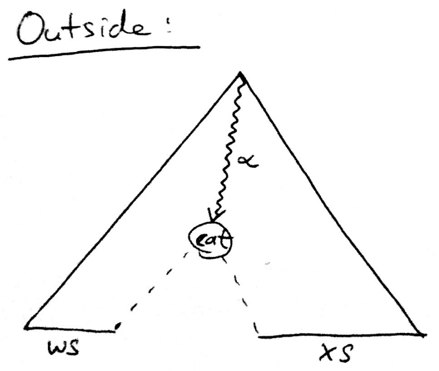
We've talked a lot about inside probabilities which are analogous to backward probabilities. Recall that if C_α is the category of the node at address α in a generated tree and W_α is the word(s) at the leaves dominated by the node at address α, then we can write the following for inside probabilities.

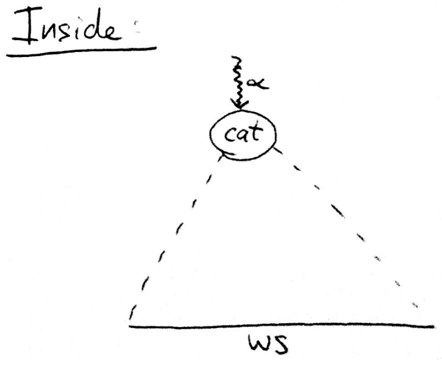
$$\texttt{probInside}\ ws\ cat = \Pr(W_\alpha = ws \mid C_\alpha = cat)$$

We have not talked as much about outside probabilities, which are (roughly) analogous to forward probabilities. Let us write $W_{\overleftarrow{\alpha}}$ for the words at the leaves that are outside of address α on the left, and $W_{\overrightarrow{\alpha}}$ for the words at the leaves that are outside of address α on the right. Then we can write the following for outside probabilities.

$$\texttt{probOutside}\ (ws, xs)\ cat = \Pr(W_{\overleftarrow{\alpha}} = ws, W_{\overrightarrow{\alpha}} = xs, C_\alpha = cat)$$

Here are the pictures to have in mind:





D. Suppose I have a particular probabilistic CFG (which has A and B as two of its categories), such that:

$\texttt{probOutside}$ ('abc', 'xyz') A > 0

$\texttt{probOutside}$ ('tuv', 'pqr') B > 0

$\texttt{probOutside}$ ('def', 'ijk') A > 0

$\texttt{probInside}$ 'ghi' A > 0

$\texttt{probInside}$ 'ijk' B > 0

What strings can we conclude are assigned non-zero probabilities by this probabilistic CFG?

Linking probabilities

Let's define a *linking probability for FSAs* as follows:

$$\texttt{probLinkingFSA}\ ws\ st_1\ st_2 = \Pr(W_i \ldots W_{i+n-1} = ws, S_{i+n} = st_2 \mid S_i = st_1)$$

A linking probability says something about the relationship between some word-sequence, ws, and two states, st_1 and st_2. It is conditioned upon, or "starts from", an assumption that a certain state (st_1) appears at a certain position (position i) in the state-sequence, like a backward probability. But it is also the probability that (among other things) a certain state (st_2) ends up appearing at a certain position (position $i + n$) in the state-sequence, like a forward probability.

Similarly, let's define a *linking probability for CFGs* as follows:

$$\texttt{probLinkingCFG}\ (ws, xs)\ cat_1\ cat_2 = \Pr(W_{\overleftarrow{\alpha}} = ws, W_{\overrightarrow{\alpha}} = xs, C_{\beta\alpha} = cat_2 \mid C_\beta = cat_1)$$

This says something about the relationship between a pair of word-sequences, (ws, xs), and two categories cat_1 and cat_2. It is conditioned upon, or "starts from", an assumption that a certain category (cat_1) appears at a certain address (address β) in the tree, like an inside probability. But it is also the probability that (among other things) a certain category (cat_2) ends up appearing at a certain address (address $\beta\alpha$) in the tree, like an outside probability.

E. Suppose I have a particular probabilistic FSA (which has 10 and 20 as two of its states), such that:

`probForward` 'abc' $10 > 0$

`probForward` 'ghi' $10 > 0$

`probForward` 'abc' $20 > 0$

`probBackward` 'def' $20 > 0$

`probLinkingFSA` 'pqr' $10\ 20 > 0$

What strings can we conclude are assigned non-zero probabilities by this probabilistic FSA?

F. Suppose I have a particular probabilistic CFG (which has A and B as two of its categories), such that:

`probOutside` ('abc', 'xyz') A > 0

`probInside` 'ghi' A > 0

`probInside` 'ijk' B > 0

`probLinkingCFG` ('xyz', 'tuv') A B > 0

`probLinkingCFG` ('def', 'pqr') A A > 0

What are six strings that we can conclude are assigned non-zero probabilities by this probabilistic CFG?

If the one idea that stays with you from this course is the very general notion of *interchangeability* that these last few questions have been getting at, and the way it applies across various different kinds of grammars, that probably wouldn't be too bad.