LING185A, Assignment #8

Due date: Wed. 3/7/2018

Download `ProbFSA_Stub.hs` from the course website, and rename it to `ProbFSA.hs`.[1] This file contains the code for probabilistic FSAs that we saw in class, including the forward probability function. You will need to submit a modified version of this file. There are `undefined` stubs for each of the functions you need to write.

Review: Forward probabilities

There are two ways in which the grammar `pfsa3` can, from its start state, generate 'a d' and get to state 40. The `probForward` function computes for us the probability of *either* of these happening, i.e. the sum of the two routes' probabilities. This is known as a forward probability.

```
*ProbFSA> probForward pfsa3 ["a","d"] 40
0.41
```

This is the probability of starting off by generating 'a d' and getting to state 40, irrespective of what happens after that. If we wanted to know the probability of generating 'a d' and getting to state 40 and stopping there, we would include that additional ending probability.

```
*ProbFSA> probForward pfsa3 ["a","d"] 40 * endProb pfsa3 40
0.205
```

A useful way to think about how the `probForward` function works is to think of a table where there is one row for each state, and one column for each position in the state-sequence we are considering. If the sequence of output symbols we have is of length n, then there will be $n+1$ columns, for the $n+1$ positions in the state-sequence. For example, if we were considering the output sequence `["a","d"]`, we we would have three columns. Each cell in this table will contain a probability; specifically, the probability computed by `probForward` $w_1 \dots w_i$ st where st is the state corresponding to the cell's row, and $w_1 \dots w_i$ is that part of the input which is emitted by transitions that get us to the cell's column.[2] So the unshaded cell in Figure 1 corresponds to the probability computed by `probForward ["a","d"] 40`.

This probability gets computed recursively, building upon forward probabilities for `["a"]`, which correspond to the cells in the column immediately to the left of the cell we're trying to fill in. We set up the recursion in such a way that we can assume that we know those values; this is indicated by showing them shaded in the picture. So our task is to work out what should go in the unshaded box, given the probabilities in the shaded boxes and the probabilities of the grammar itself. What we do is consider, for each of the shaded cells, what "contribution" they make to the probability that needs to go into the unshaded cell; this contribution is the value in the shaded cell, multiplied by an appropriate transition probability and an appropriate emission probability. (In this example, two of the shaded cells make contributions of zero, since transitions from `10`

[1] You know the drill.

[2] To save space, I'll sometimes leave out the `ProbFSA` argument to functions like `probForward` when it's not relevant. Remember to put an appropriate argument in when you try actually running any examples.

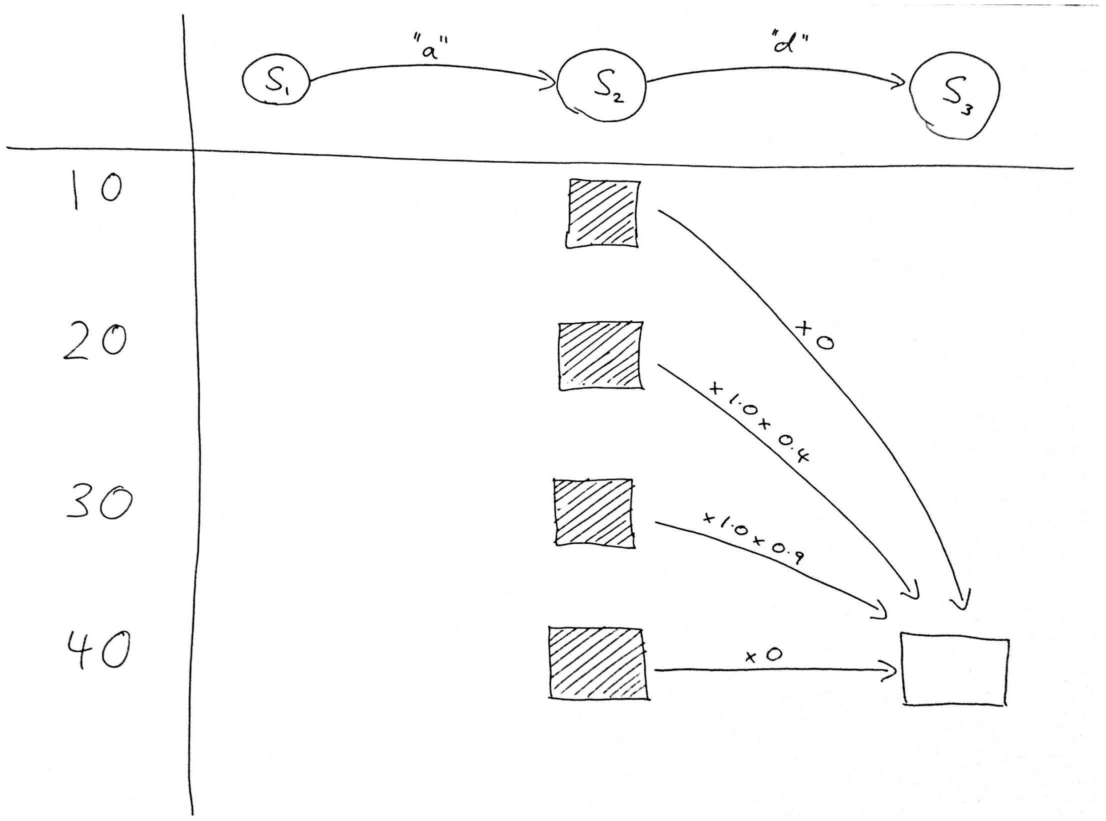


Figure 1

to 40 and from 40 to 40 are not possible.) The value that belongs in the unshaded cell is the *sum* of these contributions.

A more general version of the picture is shown in Figure 2.

Using an ad hoc mixture of Haskell and normal mathematical notation, we can expressed the recursive pattern here as in (1). And although it's not shown in the picture, the accompanying base case (i.e. the way we fill in the leftmost column of the table, which happens "first" from the dynamic programming perspective), is specified in (2).

(1) $\texttt{probForward}\ w_1 \dots w_{n-1} w_n\ st$

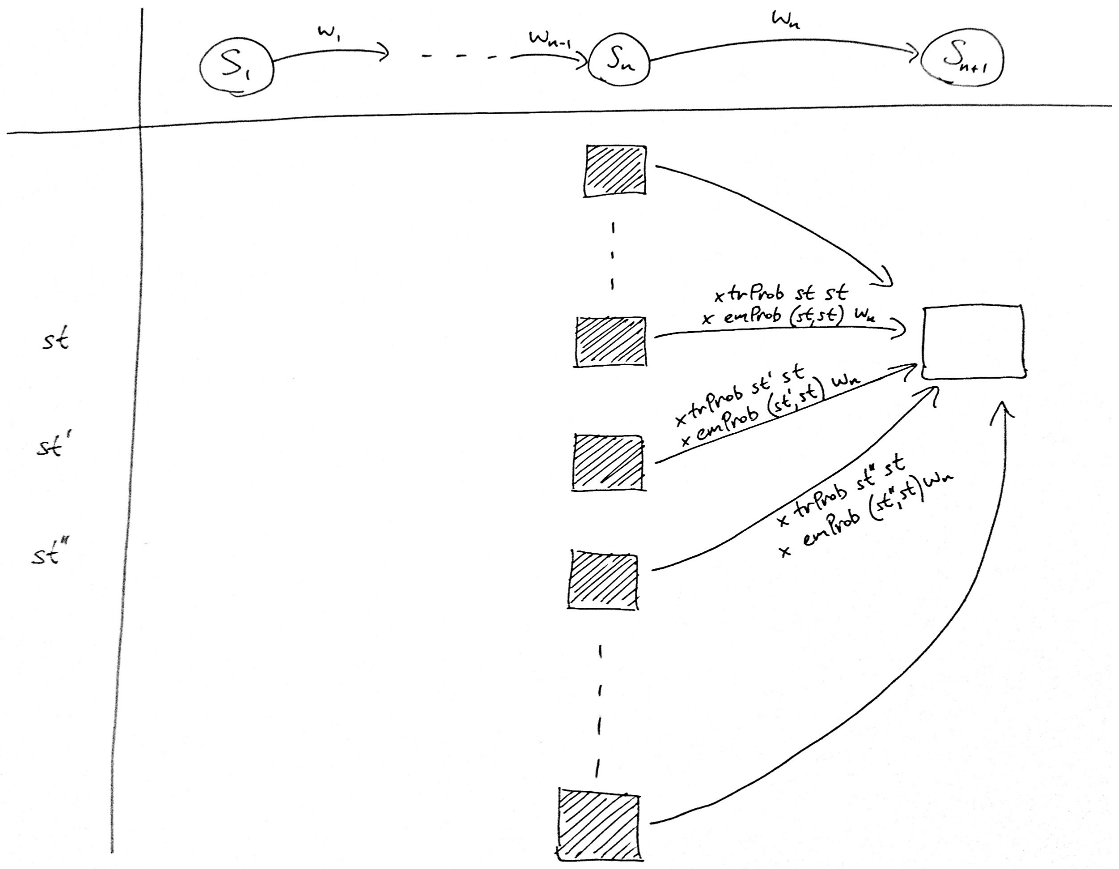
$$= \sum_{prev \in \texttt{allStates}} \big[(\texttt{probForward}\ w_1 \dots w_{n-1}\ prev) \times (\texttt{trProb}\ prev\ st) \times (\texttt{emProb}\ (prev, st)\ w_n) \big]$$

(2) $\texttt{probForward}\ [\,]\ st \quad = \quad \texttt{startProb}\ st$

Have a look at the provided implementation of `probForward` and make sure you completely understand how it (in particular, the use of `map`) connects to these equations and to the picture in Figure 2.

1 Backward probabilities

A forward probability is the probability of (i) starting in a way that the grammar allows, (ii) emitting some given sequence of symbols, and (iii) landing in a given state. A backward probability is the probability, given



w_1
w_{n-1}
w_n
S_1
S_n
S_{n+1}
st
st'
st''
× trProb st st × emProb (st,st) w_n
× trProb st' st × emProb (st',st) w_n
× trProb st'' st × emProb (st'',st) w_n

Figure 2

that (i) we are starting from a particular given state, of (ii) emitting some given sequence of symbols, and (iii) ending in a way that the grammar allows.

Your task here is to implement the function `probBackward :: ProbFSA -> [String] -> State -> Double` to compute backward probabilities. Notice that the recursion on the list argument is much less awkward here than it is with `probForward`, so although the equations look much the same the code itself is much more "Haskell-ish" here.

Here are some examples to check:

```
*ProbFSA> probBackward pfsa3 ["a","d"] 10
0.20500000000000002
*ProbFSA> probBackward pfsa3 ["d","e","a","d"] 20
4.1000000000000001e-2
*ProbFSA> probBackward pfsa3 ["d","e","a","d"] 30
9.2250000000000001e-2
*ProbFSA> probBackward pfsa3 ["c","e","a","d"] 20
6.15e-2
*ProbFSA> probBackward pfsa3 ["c","e","a","d"] 30
0.0
```

2 Viterbi probabilities

The forward probability `probForward ["a","d"] 40` is the *total* probability, across all possible state sequences ending at state 40, of emitting `["a","d"]`. Suppose now that we're interested not in that total probability, but rather in the single state sequence ending at state 40 that is most likely to have produced the output `["a","d"]`. We'll call the probability of that single best state sequence, *whatever it happens to be*, `viterbiProbForward ["a","d"] 40`.[3] (This name has `Forward` as a part of it because, like `probForward`, it relates to outcomes that include starting off in a grammatically-licensed way and moving forward from there; and as a result, we're going to be computing it "from left to right", with our base case on the far left, in the same way that we do `probForward`. But when we talk of "forward probabilities", we are talking about the results of `probForward`, not `viterbiProbForward`. Yes, there's `viterbiProbBackward` coming later.)

Working out values for `viterbiProbForward` can be thought of as filling in a table following exactly the same pattern as we used for forward probabilities. In other words, it turns out that knowing the values of `viterbiProbForward ["a"]` *prev* for each possible previous state *prev* provides us with everything we need to calculate `viterbiProbForward ["a","d"] 40`. Look back at Figure 1, bearing in mind that the value in each cell is now the probability of the *most likely* route leading to that position, rather than the total probability of reaching that position. As before, each number from one of the shaded cells needs to be multiplied by the appropriate transition probability and emission probability (perhaps zero), in order to determine that shaded cell's "contribution" to the unshaded cell. But this time we do not want to combine those contributions by *adding* them — this time they are "competing" to be the best analysis of the output sequence `["a","d"]`. So we want to take the *largest* of these four probabilities, and this is the number that goes in the shaded box.

The formula for the recursive calculation of `viterbiProbForward` is therefore eerily similar to the analogous one for forward probabilities:

$$\begin{aligned}&\texttt{viterbiProbForward}\ w_1 \dots w_{n-1} w_n\ st\\ &\quad = \max_{prev \in \texttt{allStates}} \left[(\texttt{viterbiProbForward}\ w_1 \dots w_{n-1}\ prev) \times (\texttt{trProb}\ prev\ st) \times (\texttt{emProb}\ (prev, st)\ w_n)\right]\end{aligned}$$

$$\texttt{viterbiProbForward}\ [\,]\ st \quad = \quad \texttt{startProb}\ st$$

If you're not convinced that this works — it is quite astounding at first — then here's something that might help. The four incoming numbers in Figure 1 (understood as a table of `viterbiProbForward` values, not `probForward` values) represent four candidate analyses of the input `["a","d"]` that all end at state 40: one has state 10 as the previous state, one has state 20 as the previous state, etc. The candidate analysis contributed by the shaded cell alongside state 20, i.e. the one with state 20 as the previous state, contains as a part *the best* analysis of the input `["a"]` that ends at state 20; and the candidate analysis contributed by the shaded cell alongside state 30 contains as a part the best analysis of the input `["a"]` that ends at state 30, and so on. The crucial insight is that the best analysis of `["a","d"]` *must* have one of *these four* analyses of the input `["a"]` as a subpart: if someone suggests to you an analysis of `["a","d"]` which does not have as a subpart one of those four analyses of `["a"]`, then you would be able to improve on that suggestion by swapping in the appropriate one of those four analyses for the initial subpart. (It's all about that interchangeability of *equivalent subparts* . . .)

Your task here is to implement the function `viterbiProbForward :: ProbFSA -> [String] -> State -> Double`. The function `maximum`, which takes a list of numbers (or any other sortable type, actually) and returns the largest member of the list, will be useful.

It should work like this:

```
*ProbFSA> viterbiProbForward pfsa3 ["a","d"] 40
0.378
```

[3]Named for Andrew Viterbi, who invented the technique explained below while he was a professor here at UCLA in the late 1960s: `https://en.wikipedia.org/wiki/Andrew_Viterbi`

```
*ProbFSA> viterbiProbForward pfsa3 ["a"] 10 * trProb pfsa3 10 40 * emProb pfsa3 (10,40) "d"
0.0
*ProbFSA> viterbiProbForward pfsa3 ["a"] 20 * trProb pfsa3 20 40 * emProb pfsa3 (20,40) "d"
3.2000000000000001e-2
*ProbFSA> viterbiProbForward pfsa3 ["a"] 30 * trProb pfsa3 30 40 * emProb pfsa3 (30,40) "d"
0.378
*ProbFSA> viterbiProbForward pfsa3 ["a"] 40 * trProb pfsa3 40 40 * emProb pfsa3 (40,40) "d"
0.0
```

And so, just for the record: we can now calculate the probability of the best *complete* derivation that emits `["a","d"]` and ends in state 40:

```
*ProbFSA> viterbiProbForward pfsa3 ["a","d"] 40 * endProb pfsa3 40
0.189
```

And by likewise computing `viterbiProbForward pfsa3 ["a","d"]` *st* `* endProb pfsa3` *st* for each state *st* (not just state 40) and taking the maximum, we can work out the probability of the best complete derivation that emits `["a","d"]`, period. Note that the state which maximizes this value might not be the same state as the one that maximizes `viterbiProbForward pfsa3 ["a","d"]` *st* alone!

3 Viterbi state-sequences

Now for something a bit trickier . . .

Of course we aren't usually very interested in just knowing the *probability of* the most likely state-sequence for a given output — we'd like to be able to find out what that most likely state-sequence is, i.e. which grammatical rules made that output happen, or what "structure" is assigned to that output by the grammar.[4] The relevant information is hiding there inside the workings of the `viterbiProbForward` function and the way it "fills in the table", but we don't have a convenient way to get it out. For example, we saw above that `viterbiProbForward pfsa3 ["a","d"] 40` comes out to be 0.378, and we could see there that the state-sequence that this is the probability of had state 30 in the position immediately before state 40. And we must have worked out what came before state 30 in that best sequence at some point, because we had to look at the range of candidates and choose the best one there too — but we only "recorded" (in the table) that winning probability, not the winning state that contributed it.

The solution to this is to imagine filling in each cell of the table with not only a probability, but also the state that contributed that winning probability. So in the scenario in Figure 1, what gets recorded in the unshaded cell is not just the probability 0.378, but also state 30. This extra information is sometimes known as a "backpointer": it points you back through the table along the path that was found to have the most likely probability. The way we are implementing things, this means that we will have a new function that works roughly like `viterbiProbForward` but returns a `(State,Double)` pair instead of simply a `Double`.

<u>Your task</u> is to write this function `viterbiPairForward :: ProbFSA -> [String] -> State -> (State,Double)`.

Some things to note:

- You can get the two components out of a pair using the `fst` ("first") and `snd` ("second") functions. For example `fst (10, 0.35)` is `10`, and `snd (10, 0.35)` is `0.35`.
- You may find it useful to define a helper function with type `[(a,Double)] -> (a,Double)`.
- There is a bit of a catch relating to the base case: in the leftmost column of the table, there is no previous state to point back to. So what should the result be for, say, `viterbiPairForward pfsa3 [] 10`? We know that the second member of the pair should be `startProb pfsa3 10`, just like before. What should the first member of the pair be? There's no good answer. You could just make it `0` or

[4]Some people will insist that this doesn't count as "structure", because it's list-shaped and not tree-shaped.

-1, or choose some number that doesn't correspond to a state by looking at the result of `allStates`. But a nice Haskell-ish solution here is just to leave it `undefined`. As long as we never ask Haskell to actually evaluate the *first member of* the result of something like `viterbiPairForward pfsa3 [] 10` — and it's OK to agree not to do this, because there's never any reason we would want to — this won't cause any problems. So my implementation of this function begins like this:

```
viterbiPairForward :: ProbFSA -> [String] -> State -> (State,Double)
viterbiPairForward pfsa output st =
    if output == [] then
        (undefined, startProb pfsa st)
    else
```

- Note that when you call `viterbiPairForward` recursively and "look back at the shaded cells" one column to the left, you never actually care about the state that is found in any of those shaded cells, only the probability that is found there. The state component of the result is not useful for these recursive calls. (This means that you could in principle get away with only calling `viterbiProbForward` on the subpart, instead of `viterbiPairForward` recursively. But instead of relying on that other function, try to write a single `viterbiPairForward` function that does all the work we need.)

We can now use this function to trace back through the optimal state-sequence one step at a time. For example:

```
*ProbFSA> viterbiPairForward pfsa3 ["a","d","e","a","d"] 40
(30,7.144199999999999e-2)
*ProbFSA> viterbiPairForward pfsa3 ["a","d","e","a"] 30
(10,7.937999999999999e-2)
*ProbFSA> viterbiPairForward pfsa3 ["a","d","e"] 10
(40,0.189)
*ProbFSA> viterbiPairForward pfsa3 ["a","d"] 40
(30,0.378)
*ProbFSA> viterbiPairForward pfsa3 ["a"] 30
(10,0.42)
```

Try it out with `pfsa1`, which is a probabilistic version of the ambiguous grammar from Assignment #4, and `["these","buffalo","damaged","stuff"]`.

4 Viterbi, done backwards

4.1 Probabilities only

We can also define "backwards Viterbi probabilities": that is, we'd like `viterbiProbBackward` $w_1 \dots w_n$ st to compute the probability of the *best* route, starting from state st, that emits $w_1 \dots w_n$ and then ends. This is an idea that has a quite natural linguistic interpretation, given the way we have thought about the relationship between FSAs and CFGs.

Your task here is to implement the function `viterbiProbBackward :: ProbFSA -> [String] -> State -> Double` to do this.

```
*ProbFSA> viterbiProbBackward pfsa3 ["a","d"] 10
0.189
*ProbFSA> viterbiProbBackward pfsa3 ["a","d","e","a","d"] 10
3.5720999999999996e-2
*ProbFSA> viterbiProbBackward pfsa3 ["a","d","e","a","d"] 30
0.0
*ProbFSA> viterbiProbBackward pfsa3 ["d","e","a","d"] 30
```

```
8.505e-2
*ProbFSA> viterbiProbBackward pfsa3 ["d","e","a","d"] 20
3.78e-2
```

4.2 Structural descriptions

Of course at this point we could "complete the paradigm" by writing `viterbiPairBackward`, which computes best paths-to-the-end but also records the backpointers, i.e. computes `(State,Double)` pairs.[5] But given the recursive structure of these "backward" calculations, we can do better ...

The stub file contains a copy of the definition of the `StrucDesc` type from week 4:

```
data StrucDesc = End State | Step State String StrucDesc deriving Show
```

<u>Your task</u> here is to implement a function `viterbiStrucDesc :: ProbFSA -> [String] -> State -> (StrucDesc,Double)`, which returns along with each probability, not just a backpointer indicating a single transition in the optimal state-sequence, but rather a structural description representing that complete state-sequence.[6] This function should "work backwards", in the Haskell-friendly way that `probBackward` and `viterbiBackward` do (but I'm leaving `Backward` out of the name because computing a `StrucDesc` any other way would not make sense). If there is no way to emit the given symbol-sequence starting from the given state, then the probability returned should naturally be zero, and in these cases it does not matter what `StrucDesc` appears as the first member of the pair.

```
*ProbFSA> viterbiStrucDesc pfsa3 ["a","d","e","a","d"] 10
(Step 10 "a" (Step 30 "d" (Step 40 "e" (Step 10 "a" (Step 30 "d" (End 40))))),3.5720999999999996e-2)
*ProbFSA> viterbiStrucDesc pfsa3 ["d","e","a","d"] 30
(Step 30 "d" (Step 40 "e" (Step 10 "a" (Step 30 "d" (End 40)))),8.505e-2)
*ProbFSA> viterbiStrucDesc pfsa3 ["d","e","a","d"] 20
(Step 20 "d" (Step 40 "e" (Step 10 "a" (Step 30 "d" (End 40)))),3.78e-2)
*ProbFSA> viterbiStrucDesc pfsa3 ["d","e","a","d"] 10
(Step 10 "d" (Step 40 "e" (Step 10 "a" (Step 30 "d" (End 40)))),0.0)
*ProbFSA> viterbiStrucDesc pfsa1 ["these","buffalo","damaged","stuff"] 1
(Step 1 "these" (Step 3 "buffalo" (Step 4 "damaged" (Step 4 "stuff" (End 5)))),4.2336e-3)
```

[5] Or perhaps they should be called "forwardpointers" in this context!

[6] This is *basically* answering question (10b) from the class handout. There's one tiny component missing from what we're doing here. Can you see what it is?