LING185A, Assignment #9

Due date: Wed. 3/14/2018

Download `ProbCFG_Stub.hs` from the course website, and rename it to `ProbCFG.hs`.[1] You will need to submit a modified version of this file. There are `undefined` stubs for each of the functions you need to write.

Some Haskell conveniences

You might find the built-in functions `take` and `drop` useful below. 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`.

```
Prelude> take 2 ["one","two","three","four","five"]
["one","two"]
Prelude> take 4 ["one","two","three","four","five"]
["one","two","three","four"]
Prelude> drop 2 ["one","two","three","four","five"]
["three","four","five"]
Prelude> 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:

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

More interestingly, Haskell also allows *list comprehensions*. These closely mirror the familiar notation that you are familiar with for sets, such as $\{x^2 \mid x \in \{1, 2, 3, 4, 5\}\}$. For example:

```
Prelude> [x * x | x <- [1,2,3,4,5]]
[1,4,9,16,25]
Prelude> [length s | s <- ["one","two","three","four"]]
[3,3,5,4]
Prelude> [take n [5,6,7,8] | n <- [0,1,2,3]]
[[],[5],[5,6],[5,6,7]]
```

Underlyingly, these are really just uses of `map`:

[1] You know the drill.

```
Prelude> map (\x -> x * x) [1,2,3,4,5]
[1,4,9,16,25]
Prelude> map (\s -> length s) ["one","two","three","four"]
[3,3,5,4]
Prelude> map (\n -> take n [5,6,7,8]) [0,1,2,3]
[[],[5],[5,6],[5,6,7]]
```

They can also be used with two (or more) "source lists"; underlyingly, these get de-sugared into additional uses of `map` combined with `concat`:

```
Prelude> [x + y | x <- [10,20,30,40], y <- [5,6,7]]
[15,16,17,25,26,27,35,36,37,45,46,47]
Prelude> concat (map (\x -> map (\y -> x + y) [5,6,7]) [10,20,30,40])
[15,16,17,25,26,27,35,36,37,45,46,47]
```

And later source lists can be specified in ways that depend on the value (`s`, in the example below) drawn from earlier source lists.

```
Prelude> [take n s | s <- ["hello","world"], n <- [3 .. length s]]
["hel","hell","hello","wor","worl","world"]
```

But this is obvious when you look at the conversion into good old-fashioned `map` and `concat` (note the scope of `s`):

```
Prelude> concat (map (\s -> map (\n -> take n s) [3 .. length s]) ["hello","world"])
["hel","hell","hello","wor","worl","world"]
```

Background: Table-based calculation of backward probabilities

Also download `ProbFSAv2.hs` ("version 2" just to distinguish it from last week's file), which will be useful to look at but which you do not need to modify. It contains a table-based implementation of backward probability calculations (i.e. storing intermediate results in a table to avoid recomputing them repeatedly). Your main task below will be to write table-based implementations of some analogous calculations for PCFGs, using `ProbFSAv2.hs` as a guide.

To begin, have a look at the `naiveBackward` function in `ProbFSAv2.hs`. This calculates backward probabilities in the style we saw last week: it's the recursive function we get by directly "translating" the relevant mathematical equation into Haskell. It works, but is very slow for larger inputs because of the way it recalculates backwards probabilities of smaller substrings many times over. On my laptop, these are basically instant:

```
*ProbFSAv2> naiveBackward pfsa3 (words "a d") 10
0.20500000000000002
*ProbFSAv2> naiveBackward pfsa3 (words "a d e a d") 10
4.2025e-2
```

But there's a small noticeable delay with one more loop around the grammar:

```
*ProbFSAv2> naiveBackward pfsa3 (words "a d e a d e a d") 10
8.615125e-3
```

And this one takes fifteen seconds or so:

```
*ProbFSAv2> naiveBackward pfsa3 (words "a d e a d e a d e a d") 10
1.766100625e-3
```

I also defined `pfsa4` to be a heavily ambiguous grammar, which generates all strings of `"a"` and `"b"` via all available state-sequences. Play around with some of these examples to get a sense of the slowdown.

```
*ProbFSAv2> replicate 10 "a"
["a","a","a","a","a","a","a","a","a","a"]
*ProbFSAv2> naiveBackward pfsa4 (replicate 5 "a") 10
5.7912224e-4
*ProbFSAv2> naiveBackward pfsa4 (replicate 10 "a") 10
3.0546346738042877e-6
*ProbFSAv2> naiveBackward pfsa4 (replicate 15 "a") 10
1.5216620890080294e-8
*ProbFSAv2> naiveBackward pfsa4 (replicate 20 "a") 10
7.612080932572332e-11
```

A solution to this speed problem is to "memoize" each calculated backward probability so that it does not need to be recomputed. Think of a table like the ones we saw in class, with one row for each state in the grammar and one column for each position in the input where there's a "hidden" state. (So if the input is two words long, there are three columns, corresponding to S_0, S_1 and S_2.) Or, equivalently, there's one column for each suffix of the input, including the full input and the empty suffix. An example for input `["the","cat","chased","John"]` is shown at the top of Figure 1.

To compute backward probabilities, we begin with the right-most column: this represents the empty suffix, and therefore the base case of the recursive specification of backward probabilities: the values here are just the grammar's ending probability for each state. Then we work from right to left (hence the term "backward probabilities") through the table, following the logic of the recursive step: each value in each of the other columns can be computed from the collection of values in the column to its immediate right.

The function `buildBackwardTable` constructs a table of this sort, with values of all backward probabilities filled in, given a grammar to work with (this provides the states to be listed down the left-hand side, and all the associated probabilities) and a word-sequence to analyze (this provides the words to be listed across the top). More specifically, note the following types:

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

buildBackwardTable :: ProbFSA -> [String] -> BackwardTable
```

So the result takes the form of a lookup table that maps a `([String],State)` pair to a probability. A value of the type `([String],State)` — for example, `(["chased","John"],20)` or `(["cat","chased","John"],30)` — picks out a particular cell in the table, and the value associated with that key will be the backward probability that we would write in that cell.

I've provided a handy helper function `printMap`, which will be useful just for debugging purposes. We can use it to see what `buildBackwardTable` is doing. Notice that we don't store zero probabilities in the table; if there's no value stored for a particular `([String],State)` pair, we interpret that as probability zero.

```
*ProbFSAv2> printMap (buildBackwardTable pfsa3 ["e","a","d"])
(([],40),0.5)
((["a","d"],10),0.20500000000000002)
((["d"],20),0.2)
((["d"],30),0.45)
((["e","a","d"],40),0.10250000000000001)
*ProbFSAv2>
*ProbFSAv2>
```

```
*ProbFSAv2>
*ProbFSAv2> printMap (buildBackwardTable pfsa1 ["these","buffalo","damaged","stuff"])
(([],5),0.5)
((["buffalo","damaged","stuff"],2),1.68e-2)
((["buffalo","damaged","stuff"],3),1.0079999999999999e-2)
((["damaged","stuff"],3),2.4e-2)
((["damaged","stuff"],4),1.68e-2)
((["stuff"],4),6.0e-2)
((["these","buffalo","damaged","stuff"],1),6.7535999999999999e-3)
*ProbFSAv2> printMap (buildBackwardTable pfsa4 (replicate 3 "a"))
(([],10),0.2)
(([],20),0.1)
((["a"],10),2.6000000000000002e-2)
((["a"],20),9.4e-2)
((["a","a"],10),1.796e-2)
((["a","a"],20),2.032e-2)
((["a","a","a"],10),4.376e-3)
((["a","a","a"],20),9.5752e-3)
```

And `fastBackward` uses `buildBackwardTable` in the straightforward way to do what `naiveBackward` does, but without the serious speed problems:

```
*ProbFSAv2> fastBackward pfsa3 (words "a d e a d e a d e a d") 10
1.766100625e-3
*ProbFSAv2> fastBackward pfsa4 (replicate 20 "a") 10
7.612080932572332e-11
*ProbFSAv2> fastBackward pfsa4 (replicate 40 "a") 10
4.761273414848784e-20
*ProbFSAv2> fastBackward pfsa4 (replicate 100 "a") 10
1.1652233032190522e-47
```

How does the table actually get constructed by `buildBackwardTable`? A key part is the function `cellsToFill`.

```
*ProbFSAv2> cellsToFill pfsa1 ["e","a","d"]
[([],10),([],20),([],30),([],40),(["d"],10),(["d"],20),(["d"],30),(["d"],40),
 (["a","d"],10),(["a","d"],20),(["a","d"],30),(["a","d"],40),
 (["e","a","d"],10),(["e","a","d"],20),(["e","a","d"],30),(["e","a","d"],40)]
*ProbFSAv2> cellsToFill pfsa4 ["a","a","a","a"]
[([],10),([],20),(["a"],10),(["a"],20),
 (["a","a"],10),(["a","a"],20),(["a","a","a"],10),(["a","a","a"],20),
 (["a","a","a","a"],10),(["a","a","a","a"],20)]
```

So what this function does is compute a list of all the cells of the table, *in the order that they should be filled in with values*. The calculations of values for cells later in this list depend on values that belong in cells earlier in the list. The `suffixes` function does part of the work here.

The actual values are computed by `fillCellsBackward`. This function calculates and stores values for a list of specified cells (its third argument), given a table that is assumed to already contains values for the relevant "earlier" cells. The calculation of the value that gets the name `result` in `fillCellsBackward` is analogous to the simple `naiveBackward` function, but note the local definition of `probBackward` which retrieves previously-computed values from the table. Also notice that the line

```
sum (map (\next -> trProb g st next * emProb g (st,next) w * probBackward ws next) (allStates g))
```

could equivalently be written as

```
sum [trProb g st next * emProb g (st,next) w * probBackward ws next | next <- allStates g]
```

which will be useful to bear in mind when it comes to the analogous PCFG calculations.

Now if you look at `buildBackwardTable` you should be able to see how the pieces fit together: we provide `fillCellsBackward` with an empty table (that's `Map.empty`) and a list of all the cells that need values, beginning with the "base case" cells and proceeding up towards the cells that describe longer and longer word-sequences. (Note that `fillCellsBackward` gets its name from the fact that it computes *backward probabilities*, not from anything relating to the order in which cells get filled in; the order is dictated by `cellsToFill`, not `fillCellsBackward`.)

1 Inside probabilities for PCFGs

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

S → NP VP	1.0	NP → NP PP	0.4
PP → P NP	1.0	NP → dogs	0.1
VP → V NP	0.7	NP → telescopes	0.18
VP → VP PP	0.3	NP → saw	0.04
P → with	1.0	NP → cats	0.18
V → saw	1.0	NP → hamsters	0.1

Exactly how these rules are encoded in the `ProbCFG` type is not important (although it's not complicated, either). What matters is just 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 pcfg1 NP "cats"
0.18
*ProbCFG> endProb pcfg1 V "saw"
1.0
*ProbCFG> endProb pcfg1 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.

Notice that there are two distinct trees for the sentence 'dogs saw cats with telescopes' according to `pcfg1`. The probability of one of these trees is

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

and the probability of the other tree is

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

So the total probability of an S yielding this sequence of words, i.e. the inside probability, is

$$\Pr(W_{\widehat{\alpha}} = \text{dogs saw cats with telescopes} \mid C_\alpha = \text{S}) = 0.0009072 + 0.0006804 = 0.0015876$$

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

A. To begin, write the function `naiveInside :: ProbCFG -> [String] -> Cat -> Double` which computes inside probabilities in the "obvious", recursive manner based on the equations above. This function should be analogous to `naiveBackward` in `ProbFSAv2.hs`. The result can be `undefined` when the second argument is the empty list.

```
*ProbCFG> naiveInside pcfg1 ["dogs"] NP
0.1
*ProbCFG> naiveInside pcfg1 ["dogs"] V
0.0
*ProbCFG> naiveInside pcfg1 ["saw","cats"] VP
0.126
*ProbCFG> naiveInside pcfg1 ["dogs","saw","cats"] VP
0.0
*ProbCFG> naiveInside pcfg1 ["dogs","saw","cats"] S
1.26e-2
*ProbCFG> naiveInside pcfg1 ["saw","cats","with","telescopes"] VP
1.5875999999999998e-2
```

But you'll notice that this (like `naiveBackward`, but even worse) gets unusably slow.

Table-based calculation of inside probabilities

You'll solve this speed problem by following the pattern of what we saw for `fastBackward` in `ProbFSAv2.hs`. A function `fastInside` is already written but you will need to fill in stubs for a couple of other functions before it will work.

The general pattern that we can follow in these table-based calculations, whether computing backward probabilties with a PFSA or inside probabilities with a PCFG, is this:

- There are certain parts of the sentence, called *chunks*, for which it is useful to compute re-usable intermediate values.
  - When calculating backward probabilities, the relevant chunks are all suffixes 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.
- 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*.
  - When calculating backward probabilities, each cell corresponds to one box in a two-dimensional grid of the sort shown at the top of Figure 1. (And each column corresponds to one chunk.)
  - When calculating inside probabilities, there are multiple cells — specifically, one cell for each category — in each box in a two-dimensional grid of the sort shown at the bottom of Figure 1. (And each box corresponds to one chunk.)

- 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.

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 the list of *cells* that need to be filled, in order; this is analogous to the function of the same name in `ProbFSAv2.hs`.

C. Write the function `fillCellsInside`, which is analogous to `fillCellsBackward` in `ProbFSA.hs`. This function should *not* recursively compute any probabilities: all inside probabilities that it needs should be drawn from the table. The result from `fillCellsInside` can be `undefined` if one of the cells specified by its third argument corresponds to an empty chunk, just like we said for `naiveBackward`.

When you've written these, you'll be able to use `fastInside` like this:

```
*ProbCFG> fastInside pcfg1 ["dogs","saw","cats","with","telescopes"] S
1.5875999999999998e-3
*ProbCFG> fastInside pcfg2 ["dogs","saw","cats","with","telescopes"] S
1.0692e-3
*ProbCFG> fastInside pcfg1 ["dogs","saw","cats","with","telescopes","with","telescopes"] S
2.6535599999999996e-4
```

2 Viterbi probabilities for PCFGs

Recall from last week that:

- we can calculate the probability of the *most likely* analysis of a word-sequence, rather than the total probability of that word-sequence, by taking the largest of a collection of alternative probabilities rather than their sum; and
- by recording a backpointer in each table cell that identifies the "contributor" of this winning highest probability, we can subsequently reconstruct the most likely structural description.

This logic applies just as well to PCFGs as to PFSAs (and to many other kinds of grammars ...). The only real difference is what form the backpointers take: for PFSAs a backpointer was simply a `State`; for PCFGs it's a `(Cat,Cat,Int)` triple, where the three components represent the category of the contributing left daughter, the category of the contributing right daughter, and the number of words contributed by the left daughter.

We might therefore imagine constructing a table of type

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

which would map each cell to a probability and a backpointer. That would make sense too, but we'll do things slightly differently and represent the table by two separate `Map`s, one for the probabilities and one for the backpointers. (It turns out to make things easier.)

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

D. Write the function `buildViterbiTable` which produces a correctly filled-in `ViterbiTable` for a given grammar and word-sequence, analogous to `buildInsideTable`. You'll probably want to do this by writing a `fillCellsViterbi` function, analogous to `fillCellsInside`; and again, this function should not recursively compute any probabilities. The map containing backpointers (the second component of the `ViterbiTable`) should only be non-empty for the cell `(ws,c)` if (i) the map containing probabilities

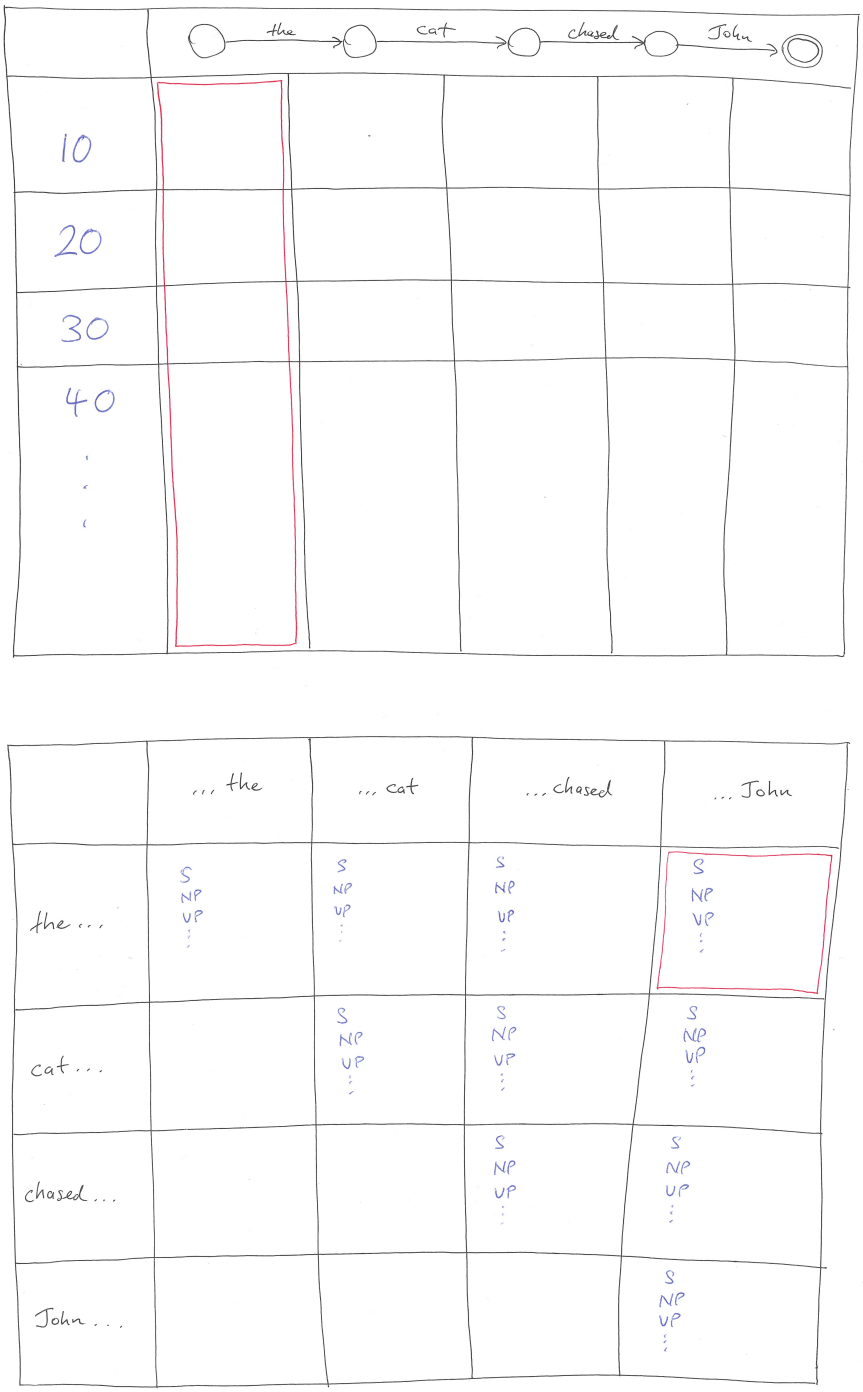

Figure 1: Generic schemas for computing probabilities for the word-sequence `["the","cat","chased","John"]`. The top grid would be used for FSA-based calculations; the bottom grid for CFG-based calculations. The *column* in the top grid that is marked in red corresponds to the complete word-sequence, 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).

(the first component of the `ViterbiTable`) has a non-zero value for the cell `(ws,c)`, and (ii) the length of `ws` is greater than one. Remember the `fst` and `snd` functions for getting the components of a pair.

```
*ProbCFG> printMap (fst (buildViterbiTable pcfg1 ["dogs","saw","cats","with","telescopes"]))
((["cats"],NP),0.18)
((["cats","with","telescopes"],NP),1.296e-2)
((["dogs"],NP),0.1)
((["dogs","saw","cats"],S),1.26e-2)
((["dogs","saw","cats","with","telescopes"],S),9.071999999999999e-4)
((["saw"],NP),4.0e-2)
((["saw"],V),1.0)
((["saw","cats"],VP),0.126)
((["saw","cats","with","telescopes"],VP),9.071999999999998e-3)
((["telescopes"],NP),0.18)
((["with"],P),1.0)
((["with","telescopes"],PP),0.18)
```

```
*ProbCFG> printMap (snd (buildViterbiTable pcfg1 ["dogs","saw","cats","with","telescopes"]))
((["cats","with","telescopes"],NP),(NP,PP,1))
((["dogs","saw","cats"],S),(NP,VP,1))
((["dogs","saw","cats","with","telescopes"],S),(NP,VP,1))
((["saw","cats"],VP),(V,NP,1))
((["saw","cats","with","telescopes"],VP),(V,NP,1))
((["with","telescopes"],PP),(P,NP,1))
```

```
*ProbCFG> printMap (snd (buildViterbiTable pcfg2 ["dogs","saw","cats","with","telescopes"]))
((["cats","with","telescopes"],NP),(NP,PP,1))
((["dogs","saw","cats"],S),(NP,VP,1))
((["dogs","saw","cats","with","telescopes"],S),(NP,VP,1))
((["saw","cats"],VP),(V,NP,1))
((["saw","cats","with","telescopes"],VP),(VP,PP,2))
((["with","telescopes"],PP),(P,NP,1))
```

E. Write the function `extractStrucDesc :: ViterbiTable -> ([String],Cat) -> StrucDesc` which uses the backpointer information in the given table (first argument) to produce the highest-probability tree for the given "cell" (second 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 = ["dogs","saw","cats"] in
          extractStrucDesc (buildViterbiTable pcfg1 ws) (ws,S)
Binary S (Leaf NP "dogs") (Binary VP (Leaf V "saw") (Leaf NP "cats"))
```

```
*ProbCFG> let ws = ["dogs","saw","cats","with","telescopes"] in
          extractStrucDesc (buildViterbiTable pcfg1 ws) (ws,S)
Binary S (Leaf NP "dogs")
         (Binary VP (Leaf V "saw")
                    (Binary NP (Leaf NP "cats") (Binary PP (Leaf P "with") (Leaf NP "telescopes"))))
```

```
*ProbCFG> let ws = ["dogs","saw","cats","with","telescopes"] in
          extractStrucDesc (buildViterbiTable pcfg2 ws) (ws,S)
Binary S (Leaf NP "dogs")
         (Binary VP (Binary VP (Leaf V "saw") (Leaf NP "cats"))
                    (Binary PP (Leaf P "with") (Leaf NP "telescopes")))
```

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 Week 6 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*).