LING185A, Take-home final exam

Due date: Wed. 3/21/2018, 5:00pm

Instructions

Download `Exam_Stub.hs` from the website, and rename it to `Exam.hs`. You will need to write code in this file. You should also download `ExamProbCFG.hs` and `ExamProbFSA.hs` and save them in the same directory, but you should not modify these two files.

This exam has three sections:

- For Sections 1 and 2 you will write some code in `Exam.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 of your answers to these question in person, to the front desk of the Linguistics Department main office (Campbell Hall 3125) during business hours.

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.

1 Generic grammar computations (9 points)

1.1 Preview

Think back to the first time we wrote a function for calculating backward probabilities from a PFSA — in the straightforward recursive manner, without storing intermediate values in a table to be re-used. It looked something like this:

```
backward :: ProbFSA -> [String] -> State -> Double
backward g output st =
    case output of
    [] -> endProb g st
    (w:ws) -> sum [trProb g st next * emProb g (st,next) w * backward g ws next | next <- allStates g]
```

And we've seen that in order to compute the probability of the *most likely single analysis* of a word-sequence starting from a given state, also known as a Viterbi probability — rather than the total probability of all such analyses, which is what we call a backward probability — we can just replace the `sum` in this code with `maximum`.

That should give you the idea of factoring out the common parts that are shared by the calculation of backward probabilities and the calculation of Viterbi probabilities, and writing a generic function which can take either `sum` or `maximum` as an argument. Such a function would look like this:

```
generic :: ([Double] -> Double) -> ProbFSA -> [String] -> State -> Double
generic f g output st =
    case output of
    [] -> endProb g st
    (w:ws) -> f [trProb g st next * emProb g (st,next) w * generic f g ws next | next <- allStates g]
```

Then we could use this to write nice short definitions of functions to calculate backward and Viterbi probabilities:

```
backward :: ProbFSA -> [String] -> State -> Double
backward g output st = generic sum g output st

viterbi :: ProbFSA -> [String] -> State -> Double
viterbi g output st = generic maximum g output st
```

That's a nice start, and gives you the gist of what we're going to be doing below — but it turns out we can do *way* better! You'll discover that we can write something which is roughly like the `generic` function above, but which factors out an even more abstract pattern which underlies not only the calculation of backward and Viterbi probabilities, but all sorts of other (P)FSA-based computations as well.

And essentially the same thing will happen with CFGs: you can probably imagine a `genericCFG` function to which we would pass either `sum` to calculate inside probabilities or `maximum` to calculate Viterbi probabilities, but we'll go further than that, too.

1.2 Computing values other than probabilities

Suppose we had available the following functions, which are analogous to the `trProb`, `emProb` and `endProb` functions we have seen for PFSAs, but which simply return a boolean indicating whether a particular "event" (a transition, an emission, or the event of ending at a particular state) is *possible* rather than returning a probability. In other words, these functions let us effectively work with a non-probabilistic FSA.

```
trOK :: ProbFSA -> State -> State -> Bool
emOK :: ProbFSA -> (State,State) -> String -> Bool
endOK :: ProbFSA -> State -> Bool
```

Using these we could write a function to compute whether it's possible, given a particular state to start from, to reach an end state by emitting a particular given sequence of symbols. (This is the function `recognizeBackward` from Assignment #4.)

```
recognize :: ProbFSA -> [String] -> State -> Bool
recognize g output st =
    case output of
    [] -> endOK g st
    (w:ws) -> elem True [trOK g st next && emOK g (st,next) w && recognize g ws next | next <- allStates g]
```

Take a moment to make sure that you understand how this works. But then, notice that the general shape of it looks awfully familiar. This, as you may have guessed, is not a coincidence!

Before going further let me introduce a few new predefined Haskell functions:

- The function `and :: [Bool] -> Bool` computes the conjunction of a list of booleans. In other words, `and` is to `&&` as `sum` is to `+`.
- The function `or :: [Bool] -> Bool` computes the disjunction of a list of booleans. In other words, `or` is to `||` as `sum` is to `+`.
- The function `product :: [Double] -> Double` computes the product of a list of numbers. In other words, `product` is to `*` as `sum` is to `+`.

Then one way to clearly see the similarities between recognition and the calculation of backward probabilities is to rewrite `recognize` and `backward` like this:

```
recognize :: ProbFSA -> [String] -> State -> Bool
recognize g output st =
    case output of
    [] -> endOK g st
    (w:ws) -> or [and [trOK g st next, emOK g (st,next) w, recognize g ws next] | next <- allStates g]
```

```
backward :: ProbFSA -> [String] -> State -> Double
backward g output st =
    case output of
    [] -> endProb g st
    (w:ws) -> sum [product [trProb g st next, emProb g (st,next) w, backward g ws next] | next <- allStates g]
```

There are five points of variation here. For each of these two functions, there is:

- Some way to get, from a grammar, a *value* associated with the event of ending on a particular state. This is `endOK` for `recognize`, and `endProb` for `backward`.
- Some way to get, from a grammar, a *value* associated with the transition from one state to another. This is `trOK` for `recognize`, and `trProb` for `backward`.
- Some way to get, from a grammar, a *value* associated with the emission of a particular symbol on the transition between two particular states. This is `emOK` for `recognize`, and `emProb` for `backward`.
- Some way to combine a list of values that represent a collection of events, to produce a value associated with *all* of those events occurring. This is `and` for `recognize`, and `product` for `backward`.
- Some way to combine a list of values that represent a collection of events, to produce a value associated with *any one* of those events occurring. This is `or` for `recognize`, and `sum` for `backward`.

The type `BundleForFSA` represents bundles of five functions that do these five things, parametrized by the type of values we want to work with:

```
type BundleForFSA a = ( ProbFSA -> State -> a,                          -- ending values
                        ProbFSA -> State -> State -> a,                 -- transition values
                        ProbFSA -> (State,State) -> String -> a,        -- emission values
                        [a] -> a,                                       -- combine such that all happen
                        [a] -> a                                        -- combine such that one happens
                      )
```

Such a bundle can be used by a nice generic function as follows:

```
genericFSA :: BundleForFSA a -> ProbFSA -> [String] -> State -> a
genericFSA b g output st =
    let (endVal, trVal, emVal, allOf, oneOf) = b in
    case output of
    [] -> endVal g st
    (w:ws) -> oneOf [allOf [trVal g st next, emVal g (st,next) w, genericFSA b g ws next] | next <- allStates g]
```

Your task will be to define the appropriate bundles that will allow this sort of generic function to be used to do all sorts of interesting computations.

1.3 Now let's look at the code

Now have a look at `Exam.hs`. Look for the function called `genericFSA`. Although its implementation is different, its type is the same as the type of the `genericFSA` function show above, and *you can basically assume that it is that same function.* The difference is that the implementation of `genericFSA` in the file stores the values it computes in a table so that they can be retrieved later rather than recomputed, like we saw in Assignment #9, so it will still be reasonably fast for larger inputs. It does this in a modular, streamlined way that hides the manipulation of the underlying table as much as possible, and also abstracts completely over the result type `a`. You should not worry about the details of how it does this, but you should be able to see (if you squint a bit and look past the use of `tryRetrieveElse`, `pure` and `lift`) the outlines of the straightforward `genericFSA` function shown above.

Notice that there's an analogous `genericCFG` function, and an associated type `BundleForCFG`. A `BundleForCFG` has only four components: something to extract "ending values" (corresponding to rules like 'D → the'), something to extract "transition values" (corresponding to rules like 'VP → V NP'), and the same two combining functions. And again, you should be able to see the outlines of the familiar pattern we've used for calculating, for example, inside probabilities, in the implementation of `genericCFG`.

Immediately above the definitions of `genericFSA` and `genericCFG`, there's a section of the file containing definitions for the functions (such as `tryRetrieveElse`) that look after the way values are stored in a table and retrieved in order to avoid recomputing them. You can safely ignore this section. Above that, you will find a collection of `import` lines that import certain things from `ExamProbFSA.hs` and `ExamProbCFG.hs`. Notice that (because some of the imports are `qualified`), in order to refer to the functions `trProb`, `emProb`, `endProb` and `allCats` from `ExamProbFSA.hs` you need to put `FSA.` in front of their names; and similarly for the functions `trProb`, `endProb` and `allStates` from `ExamProbCFG.hs`.

Now look at the part down below the definitions of `genericFSA` and `genericCFG`. Find the definition of `backwardBundleFSA`. This specifies the five choices that are relevant for the calculation of backward probabilities, mentioned above. So we can calculate backward probabilities like this:

```
*Exam> genericFSA backwardBundleFSA pfsa3 ["a","d"] 10
0.20500000000000002
*Exam> genericFSA backwardBundleFSA pfsa3 ["d","e","a","d"] 20
4.100000000000001e-2
*Exam> genericFSA backwardBundleFSA pfsa3 ["d","e","a","d"] 30
9.225000000000001e-2
*Exam> genericFSA backwardBundleFSA pfsa1 ["damaged","stuff"] 3
2.4e-2
*Exam> genericFSA backwardBundleFSA pfsa1 ["damaged","stuff"] 4
1.68e-2
*Exam> genericFSA backwardBundleFSA pfsa1 ["damaged","stuff"] 5
0.0
```

Similarly, `recognizeBundleFSA` can be used to do the same work as the `recognize` function from above:

```
*Exam> genericFSA recognizeBundleFSA pfsa3 ["d","e","a","d"] 20
True
*Exam> genericFSA recognizeBundleFSA pfsa1 ["damaged","stuff"] 4
True
*Exam> genericFSA recognizeBundleFSA pfsa1 ["damaged","stuff"] 5
False
```

Finally, I've also defined `insideBundleCFG` for you, which can be used with `genericCFG` to calculate inside probabilities:

```
*Exam> genericCFG insideBundleCFG pcfg1 ["saw","cats","with","telescopes"] VP
1.5875999999999998e-2
*Exam> genericCFG insideBundleCFG pcfg1 ["dogs","saw","cats","with","telescopes"] VP
0.0
*Exam> genericCFG insideBundleCFG pcfg1 ["dogs","saw","cats","with","telescopes"] S
1.5875999999999998e-3
*Exam> genericCFG insideBundleCFG pcfg1 ["dogs","saw","cats","with","telescopes","with","telescopes"] S
2.6535599999999996e-4
*Exam> genericCFG insideBundleCFG pcfg2 ["dogs","saw","cats","with","telescopes"] S
1.0692e-3
*Exam> genericCFG insideBundleCFG pcfg2 ["dogs","saw","cats","with","telescopes","with","telescopes"] S
2.3969520000000002e-4
```

Your task here is to define various other bundles like these three, as described below in section 1.4.

Some general notes/hints:

- It's fine for the two `[a] -> a` functions that appear in all of these bundles to be `undefined` for the empty list.
- The two `[a] -> a` functions will never need to be sensitive to the order amongst the elements of the input list. (So for example, nothing will depend on your realizing that the result from `trVal` is the first of the three elements in the list that is passed to `allOf` in `genericCFG`; it's safe to pretend that you "don't know" these internal details about how `genericCFG` is written.)
- The file contains a definition of the type `EventFromCFG`; you may find it useful to define a bundle of type `BundleForCFG EventFromCFG`, for debugging purposes. But I'll leave you to work out exactly how.
- You can, of course, define the functions that you want to put into these bundles in the usual standalone way and then write the name of that function in the tuple. In `recognizeBundleFSA` the functions that needed to be defined just happened to be simple enough that writing a lambda term was convenient. (But if one of the functions we had to write there was recursive, for example, ...)
- You may find the predefined Haskell function `foldr1` useful. Its definition looks like this:

  ```
  foldr1 :: (a -> a -> a) -> [a] -> a
  foldr1 f []     = undefined
  foldr1 f (x:[]) = x
  foldr1 f (x:xs) = f x (foldr1 f xs)
  ```

  So you can think of this as something that transforms a function that combines exactly two things (type `a -> a -> a`) into a function that combines a list of things (type `[a] -> a`). For example, `product` is just

  ```
  foldr1 (\x -> \y -> x * y)
  ```

  so it may be convenient sometimes to define a "two at a time" function that combines things in the way that you want, and then use `foldr1` to convert that into a list-based function of the kind that needs to go into a bundle.

So the upshot of these last two points is that, for example, `insideBundleCFG` could be defined as follows, which is entirely equivalent to what's in the file:

```
insideBundleCFG :: BundleForCFG Double
insideBundleCFG = (CFG.endProb, CFG.trProb, foldr1 multiply, sum)

multiply :: Double -> Double -> Double
multiply x y = x * y
```

1.4 Over to you ...

These first few should be relatively easy warm-ups to ease you into things.

A. Define `recognizeBundleCFG :: BundleForCFG Bool` so that it can be used to perform recognition with CFGs, i.e. decide whether there's a structural description for the given word-sequence which has the given category at its root.[1]

```
*Exam> genericCFG recognizeBundleCFG pcfg1 ["saw","cats","with","telescopes"] VP
True
*Exam> genericCFG recognizeBundleCFG pcfg1 ["saw","cats","with","telescopes"] S
False
*Exam> genericCFG recognizeBundleCFG pcfg1 ["dogs","saw","cats","with","telescopes"] S
True
*Exam> genericCFG recognizeBundleCFG pcfg1 ["dogs","saw","cats"] S
True
```

B. Define `viterbiBundleFSA :: BundleForFSA Double` so that it can be used to compute Viterbi probabilities in an FSA.[2]

```
*Exam> genericFSA viterbiBundleFSA pfsa3 ["a","d"] 10
0.189
*Exam> genericFSA viterbiBundleFSA pfsa3 ["a","d","e","a","d"] 10
3.5720999999999996e-2
*Exam> genericFSA viterbiBundleFSA pfsa3 ["a","d","e","a","d"] 30
0.0
*Exam> genericFSA viterbiBundleFSA pfsa3 ["d","e","a","d"] 30
8.505e-2
*Exam> genericFSA viterbiBundleFSA pfsa3 ["d","e","a","d"] 20
3.78e-2
```

C. Define `viterbiBundleCFG :: BundleForCFG Double` so that it can be used to compute Viterbi probabilities in a CFG.

```
*Exam> genericCFG viterbiBundleCFG pcfg1 ["saw","cats","with","telescopes"] VP
9.071999999999998e-3
*Exam> genericCFG viterbiBundleCFG pcfg1 ["dogs","saw","cats","with","telescopes"] S
9.071999999999999e-4
*Exam> genericCFG viterbiBundleCFG pcfg2 ["dogs","saw","cats","with","telescopes"] S
6.804000000000001e-4
```

Now some more interesting ones . . .

D. Define `countBundleFSA :: BundleForFSA Int` and `countBundleCFG :: BundleForCFG Int` so that they can be used to compute the number of possible analyses of the given word-sequence that are consistent with the given category/state.

```
*Exam> genericFSA countBundleFSA pfsa3 ["a","d","e","a","d"] 10
4
*Exam> genericFSA countBundleFSA pfsa3 ["d","e","a","d"] 20
2
*Exam> genericFSA countBundleFSA pfsa3 ["d","e","a","d"] 30
2
*Exam> genericFSA countBundleFSA pfsa3 ["e","a","d"] 30
0
*Exam> genericFSA countBundleFSA pfsa3 ["e","a","d"] 40
2
*Exam> genericFSA countBundleFSA pfsa3 ["e","a","c"] 40
1
```

```
*Exam> genericCFG countBundleCFG pcfg1 ["dogs","saw","cats"] S
1
*Exam> genericCFG countBundleCFG pcfg1 ["dogs","saw","cats","with","telescopes"] S
2
*Exam> genericCFG countBundleCFG pcfg1 ["saw","cats","with","telescopes"] S
0
*Exam> genericCFG countBundleCFG pcfg1 ["saw","cats","with","telescopes"] VP
2
*Exam> genericCFG countBundleCFG pcfg1 ["cats","with","telescopes"] NP
1
*Exam> genericCFG countBundleCFG pcfg1 ["saw","cats","with","telescopes","with","telescopes"] VP
5
```

E. Define `probsBundleFSA :: BundleForFSA [Double]` and `probsBundleCFG :: BundleForCFG [Double]` so that they can be used to compute the list of the probabilities of the possible analyses of the given

[1] Doing this should clarify the sense in which (i) the relationship between what we call backward probabilities in an FSA and what we call inside probabilities in a CFG, is the same as (ii) the relationship between what we call recognition in an FSA and what we call — umm, recognition in a CFG.

[2] To be clear, these are what were called *backward Viterbi probabilities* in Assignment #8.

word-sequence that are consistent with the given category/state. The order of the elements in the result list does not matter.

```
*Exam> genericFSA probsBundleFSA pfsa3 ["a","d","e","a","d"] 10
[2.5600000000000001e-4,3.0240000000000006e-3,3.0240000000000006e-3,3.5720999999999996e-2]
*Exam> genericFSA probsBundleFSA pfsa3 ["e","a","c"] 40
[1.2e-2]
*Exam> genericFSA probsBundleFSA pfsa3 ["e","a","d"] 40
[8.000000000000002e-3,9.45e-2]
*Exam> genericFSA probsBundleFSA pfsa3 ["a","d"] 10
[1.6000000000000004e-2,0.189]
```

```
*Exam> genericCFG probsBundleCFG pcfg1 ["saw","cats","with","telescopes"] VP
[6.803999999999999e-3,9.071999999999998e-3]
*Exam> genericCFG probsBundleCFG pcfg2 ["saw","cats","with","telescopes"] VP
[6.803999999999999e-3,3.8879999999999995e-3]
*Exam> genericCFG probsBundleCFG pcfg1 ["cats","with","telescopes"] NP
[1.296e-2]
*Exam> genericCFG probsBundleCFG pcfg2 ["cats","with","telescopes"] NP
[1.296e-2]
```

F. Define `derivationBundleCFG :: BundleForCFG [[GrammarRule]]` so that it can be used to compute the list of *leftmost derivations* that derive the given word-sequence from the given category. The *leftmost derivation* corresponding to a particular tree structure is the sequence of rules that can be used to construct that tree structure in such a way that each step rewrites the leftmost remaining nonterminal; for a more detailed explanation and examples of this, see section 2 below. The order in which the derivations appear in the result list does not matter (but the order in which the rules appear in each derivation does matter, of course).

```
*Exam> genericCFG derivationBundleCFG pcfg1 ["cats","with","telescopes"] NP
[[Step NP (NP,PP),End NP "cats",Step PP (P,NP),End P "with",End NP "telescopes"]]
*Exam> genericCFG derivationBundleCFG pcfg1 ["saw","cats","with","telescopes"] VP
[[Step VP (VP,PP),Step VP (V,NP),End V "saw",End NP "cats",
  Step PP (P,NP),End P "with",End NP "telescopes"],
 [Step VP (V,NP),End V "saw",Step NP (NP,PP),End NP "cats",
  Step PP (P,NP),End P "with",End NP "telescopes"]]
```

G. Define `probDerivBundleCFG :: BundleForCFG [(Double,[GrammarRule])]` so that it can be used to compute the list of leftmost derivations from the previous question, but with each one paired up with its probability. The order in which these pairs appear in the result list does not matter.

```
*Exam> genericCFG probDerivBundleCFG pcfg1 ["saw","cats"] VP
[(0.126,[Step VP (V,NP),End V "saw",End NP "cats"])]
*Exam> genericCFG probDerivBundleCFG pcfg2 ["saw","cats"] VP
[(5.4e-2,[Step VP (V,NP),End V "saw",End NP "cats"])]
```

H. Define `countVPsBundleCFG :: BundleForCFG [Int]` so that it can be used to compute a list containing, for each derivation of the given word-sequence from the given category, the number of VP nodes in the corresponding tree structure. The order in the result list does not matter.

```
*Exam> genericCFG countVPsBundleCFG pcfg1 ["saw","cats","with","telescopes"] VP
[2,1]
*Exam> genericCFG countVPsBundleCFG pcfg1 ["dogs","saw","cats","with","telescopes"] S
[2,1]
*Exam> genericCFG countVPsBundleCFG pcfg1 ["saw","cats","with","telescopes"] VP
[2,1]
*Exam> genericCFG countVPsBundleCFG pcfg1 ["saw","cats"] VP
[1]
*Exam> genericCFG countVPsBundleCFG pcfg1 ["with","telescopes"] PP
[0]
*Exam> genericCFG countVPsBundleCFG pcfg1 ["with","telescopes"] VP
[]
```

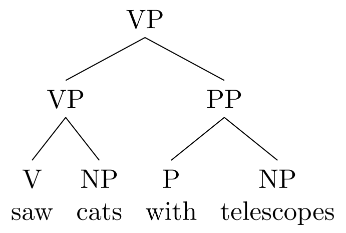
2 Leftmost derivations (5 points)

In general, given a particular tree generated by a context-free grammar, there are a number of different orders in which the rewriting steps might take place, all of which lead to the same tree. Consider, for example, carrying out a sequence of top-down rewrite steps to produce the tree in (1). The first rule used will necessarily be ‘VP $\rightarrow$ VP PP’, but the second rule might be either ‘VP $\rightarrow$ V NP’ or ‘PP $\rightarrow$ P NP’.

If we impose the constraint that at each point we rewrite the leftmost nonterminal symbol (or category), however, then for each tree there is a unique order in which the rewriting steps must have occurred. If we write down the rules that are used at each such step in an ordered list, we have what is known as a *leftmost derivation*.

For example, the leftmost derivation for the tree in (1) is shown in (2).

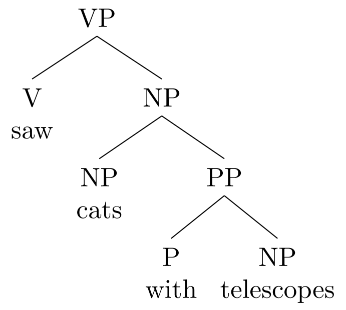
(1)



(2) [VP $\rightarrow$ VP PP, VP $\rightarrow$ V NP, V $\rightarrow$ saw, NP $\rightarrow$ cats, PP $\rightarrow$ P NP, P $\rightarrow$ with, NP $\rightarrow$ telescopes]

And the leftmost derivation for the tree in (3) is shown in (4).

(3)



(4) [VP $\rightarrow$ V NP, V $\rightarrow$ saw, NP $\rightarrow$ NP PP, NP $\rightarrow$ cats, PP $\rightarrow$ P NP, P $\rightarrow$ with, NP $\rightarrow$ telescopes]

Notice that, although carrying out rewrite steps in the order of the list in (5) will also lead to the tree in (1), it is not a leftmost derivation because it rewrites the PP before rewriting the lower VP. We can say that this is still a derivation of (1), but it is not the leftmost derivation of (1); there is only one leftmost derivation for any tree.[3]

(5) [VP $\rightarrow$ VP PP, PP $\rightarrow$ P NP, P $\rightarrow$ with, NP $\rightarrow$ telescopes, VP $\rightarrow$ V NP, V $\rightarrow$ saw, NP $\rightarrow$ cats]

And the list in (6), although it contains all the same rules, is not a derivation of (1), or of any other tree.

(6) [VP $\rightarrow$ VP PP, P $\rightarrow$ with, NP $\rightarrow$ telescopes, PP $\rightarrow$ P NP, VP $\rightarrow$ V NP, V $\rightarrow$ saw, NP $\rightarrow$ cats]

A. Write a function `leftmostCheck :: [GrammarRule] -> Bool` which returns `True` iff the given sequence of rules is a valid leftmost derivation (corresponding to any tree structure at all).

[3]Fun fact: The leftmost derivation also corresponds to the sequence of steps taken by a top-down parser.

```
*Exam> leftmostCheck [Step NP (NP,PP),End NP "cats",Step PP (P,NP),End P "with",End NP "telescopes"]
True
*Exam> leftmostCheck [Step NP (NP,PP),End NP "cats",Step PP (P,NP),End NP "telescopes",End P "with"]
False
```

B. Have a look at the type `Result`; an element of this type is either `No`, or `Yes x` for some `x` of type `StrucDesc`.[4] Write a function `leftmostToSD :: [GrammarRule] -> Result`. If the given sequence of rules is not a valid leftmost derivation then this function should return `No`, or if it is a valid leftmost derivation than it should return a `Yes` result with the `StrucDesc` for the corresponding tree structure.

```
*Exam> leftmostToSD [Step NP (NP,PP),End NP "cats",Step PP (P,NP),End P "with",End NP "telescopes"]
Yes (Binary NP (Leaf NP "cats") (Binary PP (Leaf P "with") (Leaf NP "telescopes")))
*Exam> leftmostToSD [Step NP (NP,PP),End NP "cats",Step PP (P,NP),End NP "telescopes",End P "with"]
No
```

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.

3.1 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'?

3.2 FSAs and CFGs

Reminders:

$$\begin{aligned}
\texttt{probForward}\ ws\ st &= \Pr(W_1 \dots W_{i-1} = ws, S_i = st) \\
\texttt{probBackward}\ ws\ st &= \Pr(W_i \dots W_{i+n-1} = ws, \text{end at } S_{i+n} \mid S_i = st) \\
\texttt{probInside}\ ws\ cat &= \Pr(W_{\widehat{\alpha}} = ws \mid C_\alpha = cat) \\
\texttt{probOutside}\ (ws, xs)\ cat &= \Pr(W_{\overleftarrow{\alpha}} = ws, W_{\overrightarrow{\alpha}} = xs, C_\alpha = cat)
\end{aligned}$$

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

$\texttt{probForward}$ 'abc' $10 = 0.2$

$\texttt{probForward}$ 'abc' $20 = 0.2$

$\texttt{probForward}$ 'def' $10 = 0.1$

$\texttt{probBackward}$ 'def' $10 = 0.3$

$\texttt{probBackward}$ 'pqr' $20 = 0.2$

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

[4]This type `Result` relates to `StrucDesc` in the way that the type called `Result` from Week #1 related to the type `Shape`.

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.

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

$\texttt{probOutside}\ (\text{‘abc’}, \text{‘xyz’})\ \mathrm{A} > 0$

$\texttt{probOutside}\ (\text{‘tuv’}, \text{‘pqr’})\ \mathrm{B} > 0$

$\texttt{probOutside}\ (\text{‘def’}, \text{‘ijk’})\ \mathrm{A} > 0$

$\texttt{probInside}\ \text{‘ghi’}\ \mathrm{A} > 0$

$\texttt{probInside}\ \text{‘ijk’}\ \mathrm{B} > 0$

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

3.3 Linking probabilities

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

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

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:

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

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 $\beta\alpha$, i.e. β concatenated with α) 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:

$\texttt{probForward}\ \text{‘abc’}\ 10 > 0$

$\texttt{probForward}\ \text{‘ghi’}\ 10 > 0$

$\texttt{probForward}\ \text{‘abc’}\ 20 > 0$

$\texttt{probBackward}\ \text{‘def’}\ 20 > 0$

$\texttt{probLinkingFSA}\ \text{‘pqr’}\ 10\ 20 > 0$

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:

$\texttt{probOutside}\ (\text{‘abc’}, \text{‘xyz’})\ \mathrm{A} > 0$

$\texttt{probInside}\ \text{‘ghi’}\ \mathrm{A} > 0$

$\texttt{probInside}\ \text{‘ijk’}\ \mathrm{B} > 0$

$\texttt{probLinkingCFG}\ (\text{‘xyz’}, \text{‘tuv’})\ \mathrm{A}\ \mathrm{B} > 0$

$\texttt{probLinkingCFG}\ (\text{‘def’}, \text{‘pqr’})\ \mathrm{A}\ \mathrm{A} > 0$

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