Assignment #6

Due date: Sat. 2/20/2021, 11:59pm

Download `ContextFree.hs`, `Memoization.hs` and `Assignment06_Stub.hs` from the CCLE site, save them in the same directory, and rename the latter to `Assignment06.hs`. (Please use this name exactly.) For the questions below you'll add some code to this file, and then submit your modified version. Do not modify or submit `ContextFree.hs` or `Memoization.hs`.

Background and reminders

- Have a look through the code in `ContextFree.hs` and make sure you understand what's going on there. The generalization from boolean-valued CFGs to more generic semiring values works exactly the same way as what we saw last week for FSAs — but we don't deal with as many "fancy" semirings here as last week, just booleans and probabilities.
- Have a look at the provided example in `Fibonacci.hs` to see how the `Memoization` module can be used. You can ignore the implementation in `Memoization.hs`, but the comments at the top give you an overview of the functions that it provides.
- A predefined Haskell function that is handy here, just to save some keystrokes at the terminal, is `words`, which splits a string up into a list of strings, separating at spaces.

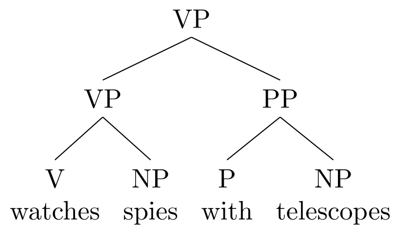
```
*Assignment06> :t words
words :: String -> [String]
*Assignment06> words "spies with telescopes"
["spies","with","telescopes"]
*Assignment06> insideBool cfg14 (words "spies with telescopes") VP
True
```

1 Leftmost derivations

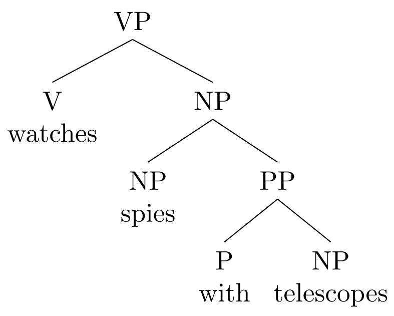
In `Assignment06.hs` I've also defined a type

```
data Tree nt t = Leaf nt t | NonLeaf nt (Tree nt t) (Tree nt t) deriving Show
```

for representing phrase-structure trees of the familiar sort, and two example trees `tree1` and `tree2`. These are our Haskell representations of the following trees:



tree1



tree2

Here are the leftmost derivations corresponding to tree1 and tree2; each derivation is shown in two columns, where the first column shows the rule being used at each step.

	VP
VP → VP PP	VP PP
VP → V NP	V NP PP
V → watches	watches NP PP
NP → spies	watches spies PP
PP → P NP	watches spies P NP
P → with	watches spies with NP
NP → telescopes	watches spies with telescopes

	VP
VP → V NP	V NP
V → watches	watches NP
NP → NP PP	watches NP PP
NP → spies	watches spies PP
PP → P NP	watches spies P NP
P → with	watches spies with NP
NP → telescopes	watches spies with telescopes

Notice that if you know the sequence of rules that applied in a leftmost derivation, then it's possible to reconstruct the entire derivation (i.e. the second column in one of the two derivations above). So such a sequence of rules can be taken as, in effect, another representation of a tree structure.

A. Write a function

```
treeToDeriv :: Tree nt t -> [RewriteRule nt t]
```

which computes the list of rewrite rules, in order, that are used in the leftmost derivation corresponding to the given tree. (Hint: The left hand side of the first rule in the output list will necessarily be the same as the root nonterminal of the input tree.)

```
*Assignment06> treeToDeriv tree1
[NTRule VP (VP,PP),NTRule VP (V,NP),TRule V "watches",TRule NP "spies",
 NTRule PP (P,NP),TRule P "with",TRule NP "telescopes"]
*Assignment06> treeToDeriv tree2
[NTRule VP (V,NP),TRule V "watches",NTRule NP (NP,PP),TRule NP "spies",
 NTRule PP (P,NP),TRule P "with",TRule NP "telescopes"]
*Assignment06> treeToDeriv (Leaf P "with")
[TRule P "with"]
```

Bonus (just for brownie points and glory, no course credit): Try writing a function to go in the other direction, converting a list of rules into a tree. This is kind of tricky, but fun! It will probably help to break this down into a few helper functions and enrich the types a bit. For example, the type `Tree nt (Either nt t)` is useful for representing "unfinished trees"; and since not all lists of rules correspond to a well-formed derivation, it will be useful to have `Maybe` types as return types. A handy trick here is that the imported `liftA` functions work on `Maybe` types, e.g. try `liftA (\x -> x + 1) (Just 3)` and `liftA (\x -> x + 1) Nothing`. (The type `Maybe a` is sort of like a restricted variant of `[a]` where the maximum length of a list is one.)

2 Inside, outside and memoization

B. Write a function

```
fastInside :: (Ord nt, Ord t, Semiring v) => GenericCFG nt t v -> [t] -> nt -> v
```

which computes the same values as the existing `inside` function (in `ContextFree.hs`), but uses the `Memoization` module to avoid the speed problems. The `Ord` restrictions on the types are inherited from the functions that you need to use from the `Memoization` module; you can essentially ignore them.

```
*Assignment06> fastInside cfg14a (words "spies with spies with telescopes") NP
9.600000000000002e-4
*Assignment06> fastInside cfg14a (words "spies with spies with telescopes") VP
9.0e-4
*Assignment06> fastInside cfg14a (words "watches spies with spies with telescopes") NP
0.0
*Assignment06> fastInside cfg14a (words "watches spies with spies with telescopes") VP
1.3920000000000004e-3
*Assignment06> fastInside cfg14a (words "watches spies with spies with telescopes with spies") VP
1.7376000000000004e-4
*Assignment06> fastInside cfg14 (words "watches spies with spies with telescopes with spies") VP
True
```

The next few functions make use of inside values. Use `fastInside` for these if you got it to work; otherwise, just use the slow `inside`.

C. Write a function

```
f :: (Ord nt, Ord t, Semiring v) => GenericCFG nt t v -> [t] -> v
```

(it's not a very good name, is it) which computes the semiring-general value that a generalized CFG assigns to a string. This is analogous to the function of the same not-very-good name last week for FSAs, i.e. we generalize from the case of boolean-valued CFGs where we said

$$w \in \mathcal{L}(G) \iff \bigvee_{n \in N} \Big[I(n) \wedge \text{inside}_G(w)(n) \Big]$$

to the semiring-general

$$f_G(w) = \bigoplus_{n \in N} \Big[I(n) \otimes \text{inside}_G(w)(n) \Big]$$

for any generalized CFG $G = (N, \Sigma, I, R)$. Notice that `cfg14a` and `cfg14b` *never differ on inside values, because they only differ in what they say about initial nonterminals*, but do produce different values here.

```
*Assignment06> f cfg14 (words "spies with telescopes")
True
*Assignment06> f cfg14a (words "spies with telescopes")
9.0e-3
*Assignment06> f cfg14b (words "spies with telescopes")
1.05e-2
*Assignment06> f cfg14 (words "telescopes with spies")
False
*Assignment06> f cfg14a (words "telescopes with spies")
0.0
*Assignment06> f cfg14b (words "telescopes with spies")
6.0e-3
```

D. Write a function

```
outside :: (Ord nt, Ord t, Semiring v) => GenericCFG nt t v -> ([t],[t]) -> nt -> v
```

which calculates the semiring-general outside value assigned to the given pair of strings for the given nonterminal, i.e. the generalized version of the recursive formulas for booleans in (25) on the CFGs handout. Notice that `cfg14a` and `cfg14b` produce different outside values, because outside values depend on what the grammar allows as initial nonterminal symbols.

```
*Assignment06> outside cfg14 (words "", words "with telescopes") NP
False
*Assignment06> outside cfg14 (words "", words "with telescopes") VP
True
*Assignment06> outside cfg14 (words "watches", words "") VP
False
*Assignment06> outside cfg14 (words "watches", words "") PP
True
*Assignment06> outside cfg14 (words "watches", words "with telescopes") NP
True
*Assignment06> outside cfg14 (words "watches", words "with telescopes") VP
False
*Assignment06> outside cfg14 (words "watches", words "with telescopes") PP
True
*Assignment06> outside cfg14a (words "watches", words "with telescopes") NP
6.0e-2
*Assignment06> outside cfg14a (words "watches", words "with telescopes") VP
0.0
*Assignment06> outside cfg14a (words "watches", words "with telescopes") PP
5.399999999999999e-3
*Assignment06> outside cfg14b (words "watches", words "with telescopes") NP
3.0e-2
*Assignment06> outside cfg14b (words "watches", words "with telescopes") VP
0.0
*Assignment06> outside cfg14b (words "watches", words "with telescopes") PP
4.5e-3
```

The relationship between the base case and the recursive case is, again, not as pretty as we're used to. As a hint, here's the beginning of what my `outside` looks like. The function `null` checks whether a list is empty.[1]

```
outside cfg (ys,zs) n =
    let (nts, ts, i, r) = cfg in
    case (null ys && null zs) of
    True -> ...
    False -> ...
```

E. Now write `fastOutside`, which has the same type and does the same thing as `outside` above, but applies memoization to its recursive calls in the same way that `fastInside` does. The natural way to do this, which is fine for our purposes, will make use of many different calls to `fastInside`, with no memoization of inside values from one call to the next. This is still a big improvement over not memoizing any outside values at all, but not blazingly fast – what one really "should" do, in principle, is build up a table of inside values just once, and draw on that for calculating outside values. But we won't worry about that here.

```
*Assignment06> fastOutside cfg14 (words "watches with", words "with spies with telescopes") NP
True
```

[1]Note that `null ys` is better than `ys == []`, because it doesn't require that the list element type is an `Eq` type.

```
*Assignment06> fastOutside cfg14a (words "watches with", words "with spies with telescopes") NP
1.0439999999999998e-3
*Assignment06> fastOutside cfg14b (words "watches with", words "with spies with telescopes") NP
8.820000000000001e-4
```

(These last two examples with `cfg14a` and `cfg14b` take about 12 seconds on my laptop.)

3 Retrieving trees

F. Write a function

```
insideTrees :: GenericCFG nt t Double -> [t] -> nt -> [(Double, Tree nt t)]
```

which finds all the tree structures which have the given string of terminals "along the bottom" and have the given nonterminal at the root, along with their probabilities according to the given grammar. Notice that this is not a semiring-general function, we're just interested in probabilistic grammars here. The list should not include any zero-probability entries. The order of the pairs in the list does not matter. The overall strategy here is closely analogous to the basic `inside` function. You should find that, by making sure that you only ever include the non-zero-probability trees in the relevant lists, this works reasonably fast even without memoization.

```
*Assignment06> insideTrees cfg14a (words "watches spies with telescopes") VP
[(4.800000000000001e-3,
  NonLeaf VP (Leaf V "watches")
             (NonLeaf NP (Leaf NP "spies") (NonLeaf PP (Leaf P "with") (Leaf NP "telescopes")))
 ),
 (7.200000000000001e-3,
  NonLeaf VP (NonLeaf VP (Leaf V "watches") (Leaf NP "spies"))
             (NonLeaf PP (Leaf P "with") (Leaf NP "telescopes"))
 )
]
```

(I've tidied up the output to make it more readable, it won't actually appear like this!)

G. Write a function

```
completeTrees :: GenericCFG nt t Double -> [t] -> [(Double, Tree nt t)]
```

which finds all the tree structures for the given string of terminals according to the given grammar, along with their probabilities — taking into account what the grammar says about initial values. In other words, this function differs from `insideTrees` in the same way that `f` differs from `inside`. Again, the order of the pairs in the list does not matter.

```
*Assignment06> map (\(x,y) -> x) (completeTrees cfg14a (words "watches spies with telescopes"))
[4.800000000000001e-3,7.200000000000001e-3]
*Assignment06> map (\(x,y) -> x) (completeTrees cfg14b (words "watches spies with telescopes"))
[2.4000000000000007e-3,3.6000000000000003e-3]
*Assignment06> map (\(x,y) -> x) (completeTrees cfg14a (words "spies with telescopes"))
[9.0e-3]
*Assignment06> map (\(x,y) -> x) (completeTrees cfg14b (words "spies with telescopes"))
[4.5e-3,6.000000000000001e-3]
```