Assignment #6

Due date: Thu. 2/20/2020

Download `SemiringCFG.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 `SemiringCFG.hs` or `Memoization.hs`.

Background and reminders

Have a look at `SemiringCFG.hs`. Things to notice:

- There is a declaration of the `Semiring` type class, and related functions `gen_and` and `gen_or`, and instance declarations for this class for `Bool` and `Double`. This should all be familiar from last week.
- We declare the type `GenericCFG nt t v` for CFGs with associated values of type `v`, and the function `makeGCFG` for making a grammar of this sort out of one comprised of "lookup tables". These are also analogous to things we saw last week.
- The grammar `cfg7` is the boolean grammar from the class handout; `cfg8` and `cfg9` are two probabilistic variants of it.
- There are three functions for calculating inside values. The first one, `insideCheck`, is the one that we wrote in class. It only works for boolean-valued grammars. The second one, `inside`, is the straightforwardly generalized version that works for values in any semiring. Unfortunately `inside` gets impractically slow for strings longer than a few terminal symbols. The `fastInside` function does *exactly* the same thing, but more efficiently: it works by recording inside values for smaller substrings in a table like the one we drew on the blackboard, so that they don't have to be recomputed each time they are needed, but you can completely ignore these details (which are implemented in `Memoization.hs`). Make sure you understand the code for `inside`, but then just use `fastInside` in its place.[1] Try these for example:

```
*Assignment06> inside cfg8 ["spies","with","telescopes"] VP
9.0e-3
*Assignment06> inside cfg8 ["watches","spies","with","telescopes"] VP
1.2000000000000002e-2
*Assignment06> fastInside cfg8 ["watches","spies","with","telescopes"] VP
1.2e-2
*Assignment06> fastInside cfg8 ["watches","spies","with","telescopes","with","telescopes"] VP
2.088e-3
```

- The one difference between `inside` and `fastInside` which you might is in constraints on the types: `fastInside` imposes an additional requirement that the types used for nonterminal and terminal symbols belong to the type class `Ord`. This is a requirement inherited from the inner workings of the

[1] If you're curious to look into how this works, one useful piece of background will be understanding how to "factor out" the recursion from a recursive function, as described in the appendix to the recursion handout from early in the course.

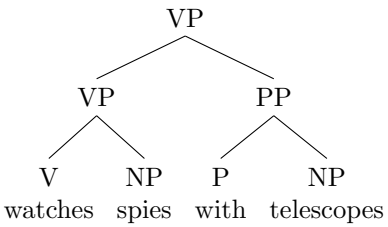
table-based memoization. You can essentially ignore it — I'll make sure that all the types you need to use do satisfy this requirement. (See e.g. the `deriving Ord` on our declaration of the `Cat` type.)

1 Leftmost derivations

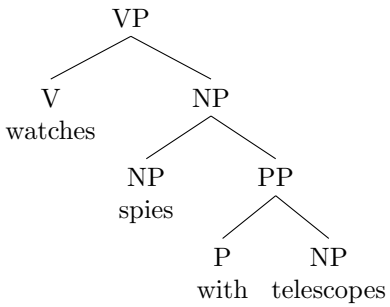
In `Assignment06.hs` I've also defined a type

```
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`

Recall that a *derivation* in a context-free grammar is a sequence of rewriting steps (see (2) and (5) on the class handout). If we restrict ourselves to derivations where each step rewrites the leftmost "unexpanded" nonterminal symbol, it turns out there's exactly one such derivation corresponding to each tree structure. These derivations are known as *leftmost derivations*. 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"]
```

2 Semirings and CFGs

First a relatively straightforward warm-up.

B. Write a function

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

that computes, in a semiring-general way, the "complete" value that the given grammar assigns to the given string, i.e. taking into account what the grammar says about starting nonterminals as well as inside values. For boolean-valued grammars, this essentially amounts to (12) on this week's handout.

```
*Assignment06> f cfg7 ["watches","spies","with","telescopes"]
True
*Assignment06> f cfg7 ["spies","with","telescopes"]
True
*Assignment06> f cfg7 ["telescopes","with","telescopes"]
False
*Assignment06> f cfg8 ["watches","spies","with","telescopes"]
1.2e-2
*Assignment06> f cfg8 ["telescopes","with","telescopes"]
0.0
*Assignment06> f cfg9 ["watches","spies","with","telescopes"]
6.0e-3
*Assignment06> f cfg9 ["telescopes","with","telescopes"]
9.0e-3
*Assignment06> f cfg9 ["telescopes","with","telescopes","with","spies","with","spies"]
7.200000000000002e-5
```

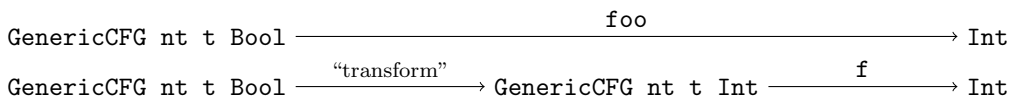
Now for the more interesting part.

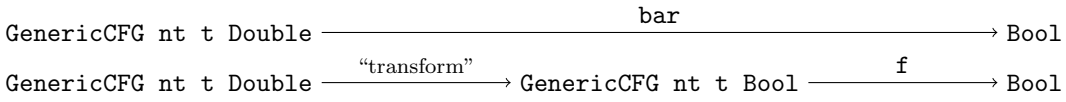
The type of the `f` function tells us that whenever we pass in a grammar that has, say, booleans associated with its rules, the value computed by `f` will also be a boolean; and whenever we pass in a grammar that has probabilities (i.e. `Double`s) associated with its rules, the value computed by `f` will also be a probability. If we want to compute some different kind of thing — for example, if we want to do some calculation whose result is an `Int` using `cfg7`, or if we want to do some calculation whose result is a `Bool` using `cfg8` — then this is not something that we can do by using `f` with `cfg7` or `cfg8` as its first argument. Maybe we'd just have to write some other functions, say called `foo` and `bar`, with these types:

```
foo :: GenericCFG nt t Bool -> Int
bar :: GenericCFG nt t Double -> Bool
```

But a neater strategy that's often available is to *transform the grammar* into some other grammar that has the same rules, but associates new values of the desired type with those rules, in such a way that running our existing, flexible `f` function on this new grammar does just what we need.

Here's a picture of the idea:





Here's an example. Given a probabilistic grammar such as `cfg8`, which associates *probabilities* with its rules, we'd like to be able to answer the *boolean* question of whether a particular string of terminal symbols is generated with any non-zero probability. To do this we can write a function that transforms a probabilistic grammar into a boolean-valued grammar:

```
probToBool :: GenericCFG nt t Double -> GenericCFG nt t Bool
probToBool (nts, ts, i, r) =
    let newi = \nt -> (i nt > 0) in
    let newr = \rule -> (r rule > 0) in
    (nts, ts, newi, newr)
```

and then use the resulting boolean-valued grammar with `f`:

```
*Assignment06> f (probToBool cfg8) ["spies","with","telescopes"]
True
*Assignment06> f (probToBool cfg8) ["telescopes","with","spies"]
False
```

The basic idea of how to write something like `probToBool` is to (i) define the `newi` function which has type `nt -> Bool`, *using* the input grammar's `i` function which has type `nt -> Double`, and (ii) similarly define `newr` using the input grammar's `r` function. You can poke around at the outputs of a function like `probToBool` like this:

```
*Assignment06> let (nts,ts,newi,newr) = probToBool cfg8 in (newi VP)
True
*Assignment06> let (nts,ts,newi,newr) = probToBool cfg8 in (newi NP)
False
*Assignment06> let (nts,ts,newi,newr) = probToBool cfg8 in (newr (NTRule VP (V,NP)))
True
```

This example isn't particularly exciting perhaps because we could just use `f` to calculate an actual probability and then check whether that probability is non-zero. But the same general pattern can be used to do some more interesting things.

C. Write a function

```
boolToCount :: GenericCFG nt t Bool -> GenericCFG nt t Int
```

and define semiring operations for $\texttt{Int}^2$ so that the resulting grammar can be used with `f` to calculate the number of distinct analyses (i.e. tree structures) for a string. (Hint: Just try the first thing that comes to mind.)

```
*Assignment06> f (boolToCount cfg7) ["watches","spies","with","telescopes"]
2
*Assignment06> f (boolToCount cfg7) ["watches","spies"]
1
*Assignment06> f (boolToCount cfg7) ["watches","spies","with","telescopes","with","telescopes"]
5
*Assignment06> f (boolToCount cfg7) ["spies","with","telescopes","with","telescopes"]
2
*Assignment06> f (boolToCount cfg7) ["with","telescopes","with","telescopes"]
0
*Assignment06> f (boolToCount (probToBool cfg8)) ["spies","with","telescopes","with","telescopes"]
2
*Assignment06> f (boolToCount (probToBool cfg9)) ["spies","with","telescopes","with","telescopes"]
4
```

D. Write a function

```
probToProbList :: GenericCFG nt t Double -> GenericCFG nt t [Double]
```

and define semiring operations for `[Double]` so that the resulting grammar can be used with `f` to calculate the probabilities of the individual tree structures for a string. So if there is only one tree structure for the given string, then the result should be a one-element list containing the probability of that tree structure, which will also be the probability of the string itself; if there are multiple tree structures for the given string, there should be one probability for each. The order in which the probabilities appear in this list is not important.

```
*Assignment06> f (probToProbList cfg8) ["watches","spies","with","telescopes"]
[4.8000000000000004e-3,7.200000000000001e-3]
*Assignment06> f (probToProbList cfg8) ["watches","spies","with","telescopes","with","telescopes"]
[2.88e-4,2.88e-4,4.3200000000000004e-4,4.3200000000000004e-4,6.48e-4]
*Assignment06> f (probToProbList cfg8) ["spies","with","telescopes","with","telescopes"]
[5.4e-4,8.099999999999998e-4]
*Assignment06> f (probToProbList cfg9) ["spies","with","telescopes","with","telescopes"]
[2.7e-4,4.049999999999999e-4,3.6e-4,3.6e-4]
```

Notice that the length of the list should be the same number as we can calculate using `boolToCount`, and the sum of the probabilities in the list should be the same number as we would get be running `f` on the original probabilistic grammar. These are actually useful hints.

```
*Assignment06> f (probToProbList cfg8) ["watches","spies","with","telescopes"]
[4.8000000000000004e-3,7.200000000000001e-3]
*Assignment06> sum (f (probToProbList cfg8) ["watches","spies","with","telescopes"])
1.2e-2
*Assignment06> f cfg8 ["watches","spies","with","telescopes"]
1.2e-2
*Assignment06> f (boolToCount (probToBool cfg8)) ["watches","spies","with","telescopes"]
2
```

E. Write a function

```
boolToDerivList :: GenericCFG nt t Bool -> GenericCFG nt t [[RewriteRule nt t]]
```

and define semiring operations for `[[RewriteRule nt t]]` so that the resulting grammar can be used with `f` to find the rule sequences used in the leftmost derivations of a string. Since the rule sequence for a single derivation has type `[RewriteRule]`, a list of such rule sequences has type `[[RewriteRule nt t]]`. The order in which the rule sequences appear in the list does not matter. The length of this list should, once again, be the same as the number of distinct derivations of the given string. The mechanism here is really the same as what we saw with output strings on FSAs; don't get distracted by the fact that we're producing lists *of rules*, it's just an output alphabet. Note that the order of the three elements that are "generalized-conjoined" in `inside` will play a role here.[3]

```
*Assignment06> f (boolToDerivList cfg7) ["watches","spies","with","telescopes"]
[[NTRule VP (V,NP),TRule V "watches",NTRule NP (NP,PP),TRule NP "spies",NTRule PP (P,NP),
  TRule P "with",TRule NP "telescopes"],
 [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> f (boolToDerivList (probToBool cfg8)) ["spies","with","telescopes"]
[[NTRule VP (VP,PP),TRule VP "spies",NTRule PP (P,NP),TRule P "with",TRule NP "telescopes"]]
*Assignment06> f (boolToDerivList (probToBool cfg9)) ["spies","with","telescopes"]
[[NTRule VP (VP,PP),TRule VP "spies",NTRule PP (P,NP),TRule P "with",TRule NP "telescopes"],
 [NTRule NP (NP,PP),TRule NP "spies",NTRule PP (P,NP),TRule P "with",TRule NP "telescopes"]]
```

(This output won't be neatly formatted like this, I've just tried to make it readable.)

F. Write a function

[2]This means that you have to add a `instance Semiring Int` block. See `SemiringCFG.hs` for some examples.

[3]What other things could we do if the order there was different? Why does this question of how to order those conjuncts only arise for CFGs and not for FSAs?

```
probToProbDerivList :: GenericCFG nt t Double ->
                                    GenericCFG nt t [(Double, [RewriteRule nt t])]
```

and define semiring operations for `[(Double, [RewriteRule nt t])]` so that the resulting grammar can be used with `f` to find the probability and rule sequence for each derivation of a string. (This is sort of a combination of the previous two questions.)

```
*Assignment06> f (probToProbDerivList cfg8) ["watches","spies","with","telescopes"]
[(4.8000000000000004e-3,[NTRule VP (V,NP),TRule V "watches",NTRule NP (NP,PP),TRule NP "spies",
                         NTRule PP (P,NP),TRule P "with",TRule NP "telescopes"]),
 (7.200000000000001e-3,[NTRule VP (VP,PP),NTRule VP (V,NP),TRule V "watches",TRule NP "spies",
                        NTRule PP (P,NP),TRule P "with",TRule NP "telescopes"])]
```