Assignment #5

Due date: Sat. 2/13/2021, 11:59pm

Download `SemiringFSA.hs` and `Assignment05_Stub.hs` from the CCLE site, save them in the same directory, and rename the latter to `Assignment05.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 `SemiringFSA.hs`.

Some background and reminders:

- `SemiringFSA.hs` defines `GenericAutomaton` types, and a handful of generalized FSAs that are copied from the class notes. Notice how these use different types (`Float`, `Double`, `Cost`, etc.) for the "values" or "weights" they associated with various possible starts, ends and transitions.
- I've decided, completely arbitrarily, to use the Haskell type `Float` for real-number weights that accumulate via maximizing, and use the type `Double` for real-number weights that accumulate via summing. This means that the values assigned to strings by `gfsa8` will be computed by taking maximums across possible paths (i.e. they will represent the weight of the best analysis of the string), and the values assigned by `gfsa25` will be computed by taking sums across possible paths (i.e. they will represent total probabilities of strings).[1]
- `SemiringFSA.hs` also defines a *type class* for semirings, and declares `Bool`, `Float` and `Double` to be members of this class by providing definitions for `(&&&)`, `(|||)`, `gtrue` and `gfalse` for each of these types.[2] The upshot of all this is that we can do things like the following:

```
*SemiringFSA> True &&& False
False
*SemiringFSA> True ||| False
True
*SemiringFSA> 0.5 &&& 0.2
0.1
*SemiringFSA> 0.5 ||| 0.2 :: Float
0.5
*SemiringFSA> 0.5 ||| 0.2 :: Double
0.7
```

```
*SemiringFSA> gtrue &&& False
False
*SemiringFSA> gtrue ||| False
True
*SemiringFSA> gtrue &&& 0.6
0.6
*SemiringFSA> gtrue ||| 0.6 :: Float
```

[1] This is a bit of a quirk of the way, in a programming language, we need to define a single interpretation of an overloaded operator like `(|||)` per type; whereas in mathematics on paper, we just say that there are two semirings based on the same set, and not worry too much about the fact that $0.5 \veebar 0.2$ is ambiguous out of context.

[2] We can also call this "providing a `Semiring` instance for `Bool/Float/Double`", or "providing a `Semiring` implementation for `Bool/Float/Double`".

```
1.0
*SemiringFSA> gtrue ||| 0.6 :: Double
1.6
*SemiringFSA> gfalse &&& 0.6
0.0
```

- If you want to understand better how `&&&` and `|||` are working as infix operators here, look at the "Haskell conveniences" handout.
- `SemiringFSA.hs` also defines the "extended" (or "big") variants of `(&&&)` and `(|||)`, namely `gen_and` and `gen_or`. Notice that we define these functions once and for all, making use only of the four semiring primitives, which lets them be used with any semiring type — even types that it might not have occurred to us yet to think of as semirings. (In practice, we will have more use for `gen_or` than `gen_and`; the latter is there more for completeness.)

```
*SemiringFSA> gen_and [True, True, False]
False
*SemiringFSA> gen_and [True, True, True]
True
*SemiringFSA> gen_and [0.25, 0.5, 0.1]
1.25e-2
*SemiringFSA> gen_or [0.25, 0.5, 0.1] :: Float
0.5
*SemiringFSA> gen_or [0.25, 0.5, 0.1] :: Double
0.85
```

- The table below might be a useful reference for keeping track of how things fit together. The left half of the table contains the pencil-and-paper notation used in the class notes and occasionally in the instructions here; the right half contains the Haskell equivalents.

Set	combine two	combine many	neutral element	Type	combine two	combine many	neutral element
$\mathbb{R}_{\geq 0}$	$\times$	$\prod$	1.0	`Float`	`(*)`	`product`	`1.0`
$\mathbb{R}_{\geq 0}$	max	max	0.0	`Float`	`max`	`maximum`	`0.0`
$\mathbb{R}_{\geq 0}$	$\times$	$\prod$	1.0	`Double`	`(*)`	`product`	`1.0`
$\mathbb{R}_{\geq 0}$	$+$	$\sum$	0.0	`Double`	`(+)`	`sum`	`0.0`
{true, false}	$\wedge$	$\bigwedge$	true	`Bool`	`(&&)`	`and`	`True`
{true, false}	$\vee$	$\bigvee$	false	`Bool`	`(||)`	`or`	`False`
generalized	$\circledwedge$	$\bigcircledwedge$	$\top$	generalized	`(&&&)`	`gen_and`	`gtrue`
generalized	$\circledvee$	$\bigcircledvee$	$\bot$	generalized	`(|||)`	`gen_or`	`gfalse`

Notice that only the things in the "combine two" and "neutral element" columns are primitives — the things in the "combine many" columns are defined out of them. When we want to set up a Haskell type to be used as a semiring, all we need to do is specify `(&&&)`, `(|||)`, `gtrue` and `gfalse` for that type.

- We can work out the value assigned to the length-one string 'V' by `gfsa8`, like this:

```
*SemiringFSA> let (states,syms,i,f,delta) = gfsa8 in (i 1 * delta (1,'V',1) * f 1)
0.9
```

Or like this:

```
*SemiringFSA> let (states,syms,i,f,delta) = gfsa8 in (i 1 &&& delta (1,'V',1) &&& f 1)
0.9
```

But notice that the latter is better because by only swapping `gfsa7` in for `gfsa8`, we can do the analogous calculation in the boolean semiring; the expression after the `in` doesn't change.

```
*SemiringFSA> let (states,syms,i,f,delta) = gfsa7 in (i 1 &&& delta (1,'V',1) &&& f 1)
True
```

- Remember that `"CV"` is just a handy abbreviation for the list `['C','V']`, so its type is `[Char]`. A synonym for this type is `String`; I'll avoid using this synonym, but I mention it just in case you see it in Haskell's error messages.

1 Writing functions that work for any semiring

These first few questions have nothing to do with FSAs yet, they are only to get you used to the idea of writing semiring-general code.

A. An important property that all semirings must have is that the generalized conjunction and generalized disjunction operations relate to each other in such a way that $x \oslash (y \otimes z) = (x \oslash y) \otimes (x \oslash z)$, i.e. $\oslash$ distributes over $\otimes$ (see (29f) on the handout). I've written the function `distrib_lhs` for you which calculates the left hand side of this equation for any three arguments x, y and z (given in that order). Write a corresponding function `distrib_rhs`, with the same type, which calculates the right hand side, i.e. $(x \oslash y) \otimes (x \oslash z)$. So we can use this as a sort of sanity-check: for all types for which `(&&&)` and `(|||)` are correctly set up, these two functions should give the same result. This isn't too exciting for real numbers and booleans perhaps but will get more interesting for other cases we see below.

```
*Assignment05> distrib_lhs True False True
True
*Assignment05> distrib_rhs True False True
True
*Assignment05> distrib_lhs 0.5 0.25 0.3 :: Double
0.275
*Assignment05> distrib_rhs 0.5 0.25 0.3 :: Double
0.275
*Assignment05> distrib_lhs 0.5 0.25 0.3 :: Float
0.15
*Assignment05> distrib_rhs 0.5 0.25 0.3 :: Float
0.15
```

B. Write a function `dotprod :: (Semiring v) => [v] -> [v] -> v` which computes what we might call the "semiring dot product"[3]: given two sequences of values $x_1 x_2 \ldots x_n$ and $y_1 y_2 \ldots y_n$ in any semiring, it should compute $(x_1 \oslash y_1) \otimes (x_2 \oslash y_2) \otimes \ldots \otimes (x_n \oslash y_n)$. (This should feel eerily familiar.) If the lists are not of equal length, then the function should just ignore the extra elements at the end of the longer list. The result for two empty lists should be the element which "leaves things alone" when combined with them via the $\otimes$ operation, i.e. `gfalse`. (This is the "natural" option: anything else would make the function harder to write.) It might seem tempting to use `gen_or` here but it's actually easier to just write this directly with recursion, pattern-matching on both input lists.

```
*Assignment05> dotprod [True, False, False] [False, False, True]
False
*Assignment05> dotprod [True, False, False] [True, False, True]
True
*Assignment05> dotprod [0.5, 0.75] [0.25, 0.75] :: Float
0.5625
*Assignment05> dotprod [0.5, 0.25] [0.25, 0.75] :: Double
0.3125
```

C. Write a function `expn :: (Semiring v) => v -> Numb -> v` which computes what we might call the "semiring exponential": given an element x of a semiring and a number n, it should combine n "copies" of x via generalized conjunction, i.e.:

$$\underbrace{x \otimes x \otimes \ldots \otimes x}_{n \text{ times}}$$

If n is zero, then the result should be `gtrue`.[4] (Again, this one is boring for booleans and real numbers, but will be more interesting for others later.)

```
*Assignment05> expn 0.5 (S (S Z))
0.25
*Assignment05> expn 0.5 (S (S (S Z)))
0.125
*Assignment05> expn 0.5 Z
1.0
*Assignment05> expn True (S (S Z))
True
*Assignment05> expn False (S (S Z))
False
*Assignment05> expn False Z
True
```

2 Now for semiring-based FSAs

We saw for basic, boolean-valued FSAs that it's convenient to "break down" the work of checking whether a string is generated into the properties of the string's subparts — specifically, if we're using *backward* values, properties of the string's *suffixes*. Exactly the same trick works for other semirings. So here you'll write a function `f` that calculates the final value associated with an entire string by an FSA with the use of a helper function `backward`.

D. Write a function

```
backward :: (Semiring v) => GenericAutomaton st sy v -> [sy] -> st -> v
```

which computes semiring-general backward values, following equation (33) on this week's handout. So for a given string `str` and a given state `q`, this function computes, in a semiring-general way, the value that an automaton associates with "finishing from `q`" in a way that produces `str`. (Don't worry about floating-point ugliness.[5])

```
*Assignment05> backward gfsa8 "VCV" 2
1.0
*Assignment05> backward gfsa25 "VCV" 2
2.2500000000000003e-2
*Assignment05> backward gfsa7 "VCV" 2
True
```

E. Now write a function

```
f :: (Semiring v) => GenericAutomaton st sy v -> [sy] -> v
```

[3] `https://en.wikipedia.org/wiki/Dot_product`. This has nothing to do with the "dot" operator we saw with regular expressions, or the associated lifted concatenation operation.

[4] It might have been tempting at first to have this function just take an `Int` argument ... but then what would we do with negative numbers? Negative exponents only make sense if we have multiplicative inverses (i.e. division), so that we can take x^{-n} to be $(x^n)^{-1}$. Since negative exponents aren't meaningful, `Numb` really is the "right" type here.

[5] `https://en.wikipedia.org/wiki/Floating-point_arithmetic#Accuracy_problems`

which computes values of the function f_M, using equation (36) in this week's handout. (This should remind you of what we did before with equation (6), of course.)

```
*Assignment05> f gfsa7 "VCV"
True
*Assignment05> f gfsa7 "CVCV"
True
*Assignment05> f gfsa8 "VCV"
0.9
*Assignment05> f gfsa8 "CVCV"
1.0
*Assignment05> f gfsa25 "VCV"
9.000000000000001e-3
*Assignment05> f gfsa25 "CVCV"
1.1250000000000001e-2
```

(See (10), (11), (26) and (27) on the handout for debugging these test cases.)

3 Adding the cost semiring

While the `f` function is defined in a nice semiring-general way, so far the only types that we can actually use it with are `Double`, `Float` and `Bool`, because these are the only types that are members of the `Semiring` type class at the moment. So we can't use `f` (or `backward`) on `gfsa37` yet, for example, because `Cost` is not a member of this type class. That's what this error message is saying:

```
*Assignment05> f gfsa37 "VCV"

<interactive>:1:1: error:
      No instance for (Semiring Cost) arising from a use of 'f'
      In the expression: f gfsa37 "VCV"
      In an equation for 'it': it = f gfsa37 "VCV"
```

And similarly:

```
*Assignment05> (TheInt 3) &&& (TheInt 4)

<interactive>:2:1: error:
      No instance for (Semiring Cost) arising from a use of '&&&'
      In the expression: (TheInt 3) &&& (TheInt 4)
      In an equation for 'it': it = (TheInt 3) &&& (TheInt 4)
```

```
*Assignment05> (TheInt 3) ||| Inf

<interactive>:3:1: error:
      No instance for (Semiring Cost) arising from a use of '|||'
      In the expression: (TheInt 3) ||| Inf
      In an equation for 'it': it = (TheInt 3) ||| Inf
```

So, let's fix this ...

F. Write a function `addCost :: Cost -> Cost -> Cost` that adds up two costs. Adding anything to an infinite cost produces an infinite cost.

```
*Assignment05> addCost (TheInt 3) (TheInt 4)
TheInt 7
```

```
*Assignment05> addCost (TheInt 3) Inf
Inf
*Assignment05> addCost Inf (TheInt 4)
Inf
*Assignment05> addCost Inf Inf
Inf
```

G. Write a function `minCost :: Cost -> Cost -> Cost` which selects the smaller of two costs (or, if the two given costs are the same, returns that cost). An infinite cost is larger than any other cost.

```
*Assignment05> minCost (TheInt 3) (TheInt 4)
TheInt 3
*Assignment05> minCost (TheInt 3) Inf
TheInt 3
*Assignment05> minCost Inf (TheInt 4)
TheInt 4
*Assignment05> minCost Inf Inf
Inf
```

H. Now you can use these two functions to make `Cost` an instance of the `Semiring` class. This is what was done already for `Bool` and `Float` and `Double` in `SemiringFSA.hs` — to do the same for `Cost` you need to write something like the following (in `Assignment05.hs`), with the gaps filled in appropriately.

```
instance Semiring Cost where
    x &&& y = ...
    x ||| y = ...
    gtrue = ...
    gfalse = ...
```

Remember that, for any cost `x`, `gtrue &&& x` should be `x`, and `gfalse ||| x` should be `x`. (And `gfalse &&& x` should be `gfalse`.) We're assuming that costs will never be negative remember.[6]

Then you'll be able to use all the existing semiring-related functions on `Cost`s ...

```
*Assignment05> dotprod [TheInt 3, TheInt 4, Inf] [Inf, TheInt 10, Inf]
TheInt 14
*Assignment05> dotprod [TheInt 3, Inf, Inf] [Inf, TheInt 10, Inf]
Inf
*Assignment05> expn (TheInt 2) (S (S (S Z)))
TheInt 6
*Assignment05> gen_and [TheInt 0, TheInt 5, TheInt 0, TheInt 0, TheInt 0]
TheInt 5
*Assignment05> gen_or [TheInt 5, TheInt 7, Inf]
TheInt 5
```

... including `f`!

```
*Assignment05> f gfsa37 "VCV"
TheInt 1
*Assignment05> f gfsa37 "CVCVCV"
TheInt 0
*Assignment05> f gfsa37 "CVCVCVC"
TheInt 2
*Assignment05> f gfsa37 "VCVCVC"
```

[6] So arguably we should have been using `Numb` here too rather than `Int` ...

```
TheInt 3
*Assignment05> f gfsa37 "VCVCVCV"
TheInt 1
*Assignment05> f gfsa37 "VVVV"
TheInt 4
*Assignment05> f gfsa37 "CC"
Inf
```

4 Adding the set-of-strings semiring

I. Now do the same thing for the type `[[a]]` as you did above for the type `Cost`: add the type `[[a]]` to the `Semiring` class, using the operations described in section 7.2 of the handout. Notice that the alphabet for the output strings, Γ, corresponds to the type variable `a` here: the set of strings Γ^* corresponds to the type `[a]`, so $\mathcal{P}(\Gamma^*)$ corresponds to the type `[[a]]`, and that's the type of the relevant semiring elements here. Here are some examples of how things should behave, using the concrete type `[[Char]]` for `[[a]]` — but since we're really using `[[a]]` to represent *sets* of lists-of-as, the order of elements in the resulting lists does not matter, and nor do any duplicates. And make sure it also works for lists of other types, e.g. `[[1,2,3],[4,5,6]] &&& [[7,8],[9,10]]`.

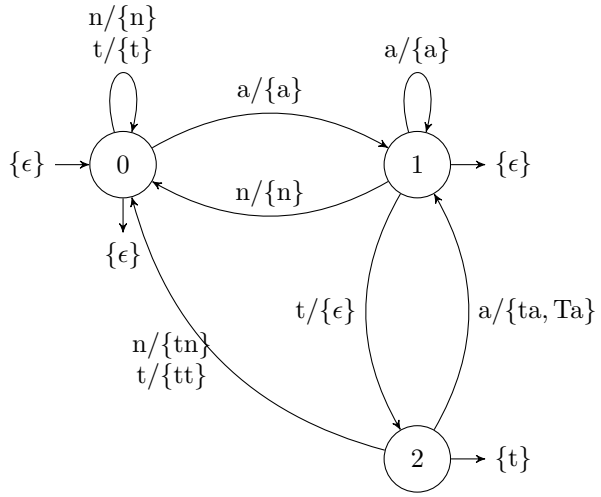
```
*Assignment05> ["hello","world"] ||| ["foo","bar"]
["hello","world","foo","bar"]
*Assignment05> ["hello","world"] &&& ["foo","bar"]
["hellofoo","hellobar","worldfoo","worldbar"]
*Assignment05> ["hello","world"] ||| []
["hello","world"]
*Assignment05> ["hello","world"] &&& []
[]
*Assignment05> ["hello","world"] &&& [""]
["hello","world"]
*Assignment05> gen_and [["tic","tac","toe"],["do","re","mi"],["foo"]]
["ticdofoo","ticrefoo","ticmifoo","tacdofoo","tacrefoo","tacmifoo","toedofoo",
 "toerefoo","toemifoo"]
*Assignment05> gen_or [["tic","tac","toe"],["do","re","mi"],["foo"]]
["tic","tac","toe","do","re","mi","foo"]
*Assignment05> dotprod [["hello","world"],["one","two"]] [["x"],["foo","bar"]]
["hellox","worldx","onefoo","onebar","twofoo","twobar"]
*Assignment05> expn ["aaa","bb"] (S (S (S Z)))
["aaaaaaaaa","aaaaaabb","aaabbaaa","aaabbbb","bbaaaaaa","bbaaabb","bbbbaaa","bbbbbb"]
```

J. Define `gfsa39 :: GenericAutomaton Int Char [[Char]]` to encode the automaton in (13) on this week's handout. So the output alphabet (Γ) here consists of the `Chars` `'C'` and `'V'`, just like the input alphabet we've been assuming. A finite state automaton with output strings like this is known as a "finite state transducer". When this is set up correctly you'll be able to use it like this:

```
*Assignment05> f gfsa39 "VCV"
["VCV","VCVV","VV"]
*Assignment05> f gfsa39 "CV"
["CV","CVV"]
*Assignment05> f gfsa39 "VC"
["V"]
*Assignment05> backward gfsa39 "VCV" 1
["VCV","VCVV","VV"]
*Assignment05> backward gfsa39 "VCV" 2
```

```
["VCV","VCVV","VVCV","VVCVV","VV","VVV"]
*Assignment05> backward gfsa39 "VCV" 3
[]
```

K. Define `gfsa_flap :: GenericAutomaton Int Char [[Char]]` to encode the finite-state transducer shown below. Two different 'input/output' pairs on a single arrow is just a notational shortcut for two different transitions. The input alphabet here is {`'a'`, `'n'`, `'t'`}, and the output alphabet is {`'a'`, `'n'`, `'t'`, `'T'`}. Think of the input alphabet as *phonemes*, and the output alphabet as *phones* (or allophones). Then this transducer encodes the phonological rule that says that a 't' optionally becomes a flap, here represented as 'T', when it appears in between two vowels (i.e. the rule that could be written as: t → T / a__a).



Here are some examples of how it should behave.

```
*Assignment05> f gfsa_flap "atan"
["atan","aTan"]
*Assignment05> f gfsa_flap "atat"
["atat","aTat"]
*Assignment05> f gfsa_flap "atnat"
["atnat"]
*Assignment05> f gfsa_flap "tatatnat"
["tatatnat","taTatnat"]
```

5 Just when you thought it couldn't get any more amazing . . .

L. Define `gfsa7_count :: GenericAutomaton Int Char Double` to encode an automaton that calculates the number of possible paths for a string in the boolean-valued automaton `gfsa7`. Notice that since you know the relevant semiring is the one associated with the type `Double`, the ⩓ and ⩔ operations that will be used are fixed for you; all you need to do is choose appropriate values of type `Double` to go on the automaton's transitions. This is easier than it might appear at first.

```
*Assignment05> f gfsa7_count "VC"
1.0
*Assignment05> f gfsa7_count "CC"
0.0
*Assignment05> f gfsa7_count "VCV"
2.0
*Assignment05> f gfsa7_count "CVVCV"
```

```
2.0
*Assignment05> f gfsa7_count "VCVCV"
4.0
```

M. Define `gfsa7_paths :: GenericAutomaton Int Char [[Int]]` to encode an automaton that calculates the list of possible state-sequences for a string in the boolean-valued automaton `gfsa7`. There are two very-slightly-different ways to do this, but the key idea is the same either way. (Notice that, the way we were interpreting these states, this is working out all the possible ways to split a string into syllables!)

```
*Assignment05> f gfsa7_paths "VC"
[[1,3,1]]
*Assignment05> f gfsa7_paths "CC"
[]
*Assignment05> f gfsa7_paths "VCV"
[[1,1,2,1],[1,3,1,1]]
*Assignment05> f gfsa7_paths "CVVCV"
[[1,2,1,1,2,1],[1,2,1,3,1,1]]
*Assignment05> f gfsa7_paths "VCVCV"
[[1,1,2,1,2,1],[1,1,2,3,1,1],[1,3,1,1,2,1],[1,3,1,3,1,1]]
```