

Assignment #5

Due date: Thu. 2/13/2020

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 (`gfsa1`, `gfsa3`, `gfsa4`, etc.). Notice how these use different types (`Bool`, `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 (see (7) and (8) on the handout), and use the type `Double` for real-number weights that accumulate via summing (see (9) and (10) on the handout).
- `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.¹ 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
1.0
*SemiringFSA> gtrue ||| 0.6 :: Double
1.6
*SemiringFSA> gfalse &&& 0.6
```

¹We can also call this “providing a `Semiring` instance for `Bool/Float/Double`”, or “providing a `Semiring` implementation for `Bool/Float/Double`”.

```
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
$[0, 1]$	\times	\prod	1.0	Float	<code>(*)</code>	product	1.0
$[0, 1]$	max	max	0.0	Float	max	maximum	0.0
$[0, 1]$	\times	\prod	1.0	Double	<code>(*)</code>	product	1.0
$[0, 1]$	+	\sum	0.0	Double	<code>(+)</code>	sum	0.0
$\{\text{true}, \text{false}\}$	\wedge	\bigwedge	true	Bool	<code>(&&)</code>	and	True
$\{\text{true}, \text{false}\}$	\vee	\bigvee	false	Bool	<code>()</code>	or	False
generalized	\otimes	\bigotimes	\top	generalized	<code>(&&&)</code>	<code>gen_and</code>	<code>gtrue</code>
generalized	\otimes	\bigotimes	\perp	generalized	<code>()</code>	<code>gen_or</code>	<code>gfalse</code>

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 the length-one string ‘V’, according to `gfsa7`, like this:

```
*SemiringFSA> let (states,syms,i,f,delta) = gfsa7 in (i 1 * delta (1,'V',1) * f 1)
0.75
```

Or like this:

```
*SemiringFSA> let (states,syms,i,f,delta) = gfsa7 in (i 1 &&& delta (1,'V',1) &&& f 1)
0.75
```

But notice that the latter is better because by only swapping `gfsa18` in for `gfsa7`, we can do the analogous calculation in the boolean semiring.

```
*SemiringFSA> let (states,syms,i,f,delta) = gfsa18 in (i 1 &&& delta (1,'V',1) &&& f 1)
True
```

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 $x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$, i.e. \otimes distributes over \oplus (see (20f) 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. $(x \otimes y) \oplus (x \otimes z)$. 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 probabilities 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”²: given two sequences of values $x_1x_2\dots x_n$ and $y_1y_2\dots y_n$ in any semiring, it should compute $(x_1 \otimes y_1) \oplus (x_2 \otimes y_2) \oplus \dots \oplus (x_n \otimes y_n)$. (This should feel eerily familiar from forward and backward calculations.) 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 \oplus 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.:

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

²https://en.wikipedia.org/wiki/Dot_product. This has nothing to do with the “dot” operator we saw with regular expressions.

If n is zero, then the result should be `gtrue`.³ (This one is rather boring for booleans, 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*. (Look back at equations (14), (21) and (24) from Handout #3.) 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 — see (22) on this week's handout — 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, defined as follows (also (29) on this week's handout):

$$\text{bwd}_M(\epsilon)(q) = F(q)$$

$$\text{bwd}_M(x_1 \dots x_n)(q) = \bigvee_{q_1 \in Q} \left[\Delta(q, x_1, q_1) \otimes \text{bwd}_M(x_2 \dots x_n)(q_1) \right]$$

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`.

```
*Assignment05> backward gfsa7 "CV" 1
1.0
*Assignment05> backward gfsa7 "CV" 2
0.0
*Assignment05> backward gfsa7 "CV" 3
0.375
*Assignment05> backward gfsa18 "CV" 1
True
*Assignment05> backward gfsa18 "CV" 2
False
*Assignment05> backward gfsa18 "CV" 3
True
```

³It 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, basically), 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.

E. Now write a function

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

which computes values of the function f_M , using the equation in (28) in this week's handout.

```
*Assignment05> f gfsa7 "VCV"
0.75
*Assignment05> f gfsa9 "VCV"
1.03125
*Assignment05> f gfsa18 "VCV"
True
```

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 two types that are members of the `Semiring` type class at the moment (or, “these are the only two types that are instances of the class `Semiring`”). So we can't use `f` (or `backward`) on `gfsa11` yet, for example, because `Cost` is not a member of this type class. That's what this error message is saying:

```
*Assignment05> f gfsa11 "VCV"

<interactive>:2:1:
  No instance for (Semiring Cost) arising from a use of 'f'
  In the expression: f gfsa11 "VCV"
  In an equation for 'it': it = f gfsa11 "VCV"
```

And similarly:

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

<interactive>:3:12:
  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>:4:12:
  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.

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

```

*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 gfsa11 "VCV"
TheInt 1
*Assignment05> f gfsa11 "CVCVCV"
TheInt 0
*Assignment05> f gfsa11 "CVCVCVC"
TheInt 2
*Assignment05> f gfsa11 "VCVCVC"
TheInt 3
*Assignment05> f gfsa11 "VCVCVCV"
TheInt 1
*Assignment05> f gfsa11 "VVVV"
TheInt 4

```

```
*Assignment05> f gfsa11 "CC"
Inf
```

4 Adding the set-of-strings semiring

- I. Now do the same thing for `[String]` as you did above for the type `Cost`: add the type `[String]` to the `Semiring` class, using the operations described in the discussion of (5.11) in the handout.⁴ Notice that in general there's no requirement that the alphabet of input symbols (which we've been calling Σ) is the same as the alphabet of output symbols — for example, perhaps the input symbols are *phonemes* and the output symbols are *phones*. A common convention is to take the input alphabet to be Σ and the output alphabet to be Γ , which is why the table on page 3 of the handout has $\mathcal{P}(\Gamma^*)$ as the set of possible output values. But for the purposes of this question, we'll just take the output alphabet Γ to be the set of characters, i.e. the type `Char`; so Γ^* corresponds to `[Char]`, and $\mathcal{P}(\Gamma^*)$ corresponds to `[[Char]]`. Here are some examples of how things should behave — but since we're really using `[[Char]]` to represent *sets* of strings, the order in which strings appear in the resulting lists does not matter.

```
*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","toedfoo",
 "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)))
["aaaaaaaa","aaaaaabb","aaabbaaa","aaabbbb","bbaaaaaa","bbaaabb","bbbbbaa","bbbbbb"]
```

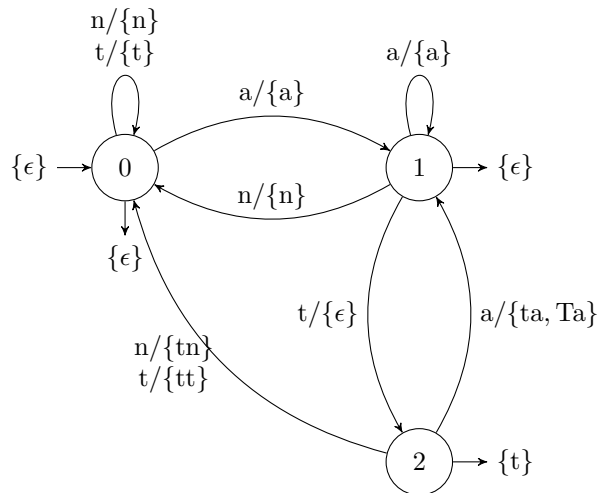
- J. Define `gfsa13 :: GenericAutomaton Int Char [String]` to encode the automaton in (13) on this week's handout. A finite state automaton with output strings like this is known as a “finite state transducer”. When this is set up correctly you'll be able to use it like this:

```
*Assignment05> f gfsa13 "VCV"
["VCVV","VV"]
*Assignment05> f gfsa13 "CV"
["CVV"]
*Assignment05> f gfsa13 "VC"
["V"]
*Assignment05> backward gfsa13 "VCV" 1
["VCVV","VV"]
*Assignment05> backward gfsa13 "VCV" 2
```

⁴The function we wrote for computing denotations of regular expressions a couple of weeks ago provides implementation hints!

```
["VVCVV","VVV"]
*Assignment05> backward gfsa13 "VCV" 3
[]
```

- K. Define `gfsa_flap :: GenericAutomaton Int Char [String]` 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 (as well as the output alphabet) should be `['a','n','t','T']`. 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.



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"]
```