Ling 185A: Assignment 5

Dylan Bumford, Winter 2024

due: Mon, Feb 19

1 Defining transducers

Take a look at the `FST.hs` module. There you'll see, among other things, the definition of a transducer:

```
type State = Int
type Arc c w = (State, (c, w), State)

data FST c w =
  FSTWith [(State, w)] [(State, w)] [Arc c w]
  --          I            F         Delta
```

Notice for the sake of brevity we are leaving the set of states and the alphabet implicit.

Also notice that there are two versions of `walk` and `transduce` defined. The plain versions require only that the output type `w` be a monoid. They do not try to do any summarization along convergent paths. They just walk them all and report the results. Then there are primed versions `walk'` and `transduce'` that require `w` to be a semiring. These versions compress results at every step, wherever possible.

As we said in class, you will sometimes want to see every possible path, and sometimes not, so use whichever one makes sense.

Finally, at the end of the file, you will see several example FSTs, based on the syllabification machines from the class slides. You will probably want to use these machines in testing your answers below.

(1) English phonology has a process of nasal place assimilation, which causes the 'n' sounds in 'in Paris', 'onboard', and 'ten males', to be pronounced as 'm' in casual speech. Simplifying greatly, we can capture this with the following phonological rule: [n] → [m] / _ {[p], [b], [m]}. In prose: 'n' becomes 'm' when it precedes 'p', 'b', or 'm'.

   (a) Craft an FST that implements this rule. To make things simpler, let's work with a restricted alphabet containing only `'i'`, `'t'`, `'n'`, and `'p'`. Your FST should transform `"inpit"` to `"impit"` (compare the pronunciation of 'input' in casual speech), but leave `"intipt"` and `"inipt"` unchanged (compare the casual pronunciations of 'intact' and 'inept'). Use `transduce` to check/debug your answer. Hint: use 3 states.

```
fstNasalAssimP :: FST Char String
fstNasalAssimP = undefined
```

   (b) Define a transducer that behaves in exactly the same way as the previous FST, but over a slightly different alphabet, one with `'m'` instead of `'p'`.

```
fstNasalAssimM :: FST Char String
fstNasalAssimM = undefined
```

   (c) If you run `transduce fstNasalAssimM "innmit"`, you should get `"inmmit"`. But notice the output here still violates the nasal assimilation rule, since there is an `n` immediately preceding an `m`. Modify `fstNasalAssimM` so that it turns any sequence of consecutive `n`s before an `m` into a sequence of `m`s. For instance, `transduce fstNasalAssimCascade "innmit"` should yield `"immmit"` (converting both of the pre-`'m'` nasals to `m`).

```
fstNasalAssimCascade :: FST Char String
fstNasalAssimCascade = undefined
```

(d) English (unlike, say, Italian and Japanese), does not actually allow doubled consonants, like `"mm"`, which are called **geminates**. So define a transducer for `'m'`-**degemination**. It should turn any occurrences of `"mm"` anywhere in a string into `"m"` (again using the `'i'`, `'t'`, `'n'`, and `'m'` alphabet). The output should not contain any `"mm"` sequences at all.

```
fstElimMM :: FST Char String
fstElimMM = undefined
```

(e) Nasal place assimilation, followed by degemination, should transform `"inmit"` to `"immit"`, and then to `"imit"` (compare the pronunciation in fast/casual speech of `"in_many"`). Chain together `fstNasalAssimCascade` and `fstElimMM` in the function `transduceBoth`. Then test your composite machine by running `transduceBoth "inmit"`.

```
transduceBoth :: String -> [String]
transduceBoth = undefined

inmitTest :: [String]
inmitTest = transduceBoth "inmit"
```

(2) Recall our definition of a finite-state automaton, in which a transition is labeled only with the symbol that is parsed in each step:

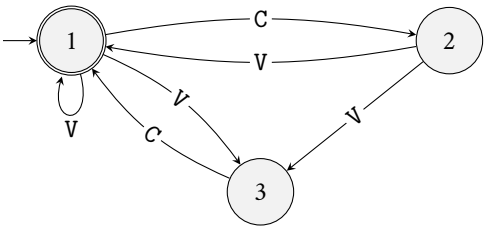
```
type State = Int
type Transition = (State, Char, State)

data FSA =
  FSAWith [State] [Char] [State] [State] [Transition]
  --         Q     Sigma    I       F        Delta
```

Define a function `fsaTrace`. This function should take in an FSA and give back an FST. The FST that it gives back should read in an input string and write out every path that the original FSA could have taken to parse that string. That is, for a string `u`, each output of the FST should be a list of the states visited along some path that the FSA takes while reading `u`.

```
fsaTrace :: FSA -> FST Char [State]
fsaTrace = undefined
```

For example, consider the FSA `fsa7` — coded up in `FST.hs` — depicted below.



Running `transduce (fsaTrace fsa7) "VC"` should yield the output `[[1,3,1]]`. This reflects that this string of segments is unambiguous; the only accepting path through `fsa7` for `"VC"` is that path from `1` to `3` back to `1`. Likewise:

```
ghci> transduce (fsaTrace fsa7) "VCV"
[[1,1,2,1],[1,3,1,1]]
ghci> transduce (fsaTrace fsa7) "CVVCV"
[[1,2,1,1,2,1],[1,2,1,3,1,1]]
```

```
ghci> transduce (fsaTrace fsa7) "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]]
```

Hint: pay close attention to the type given to `fsaTrace`. The `transduce` function will do most of the work for you, finding a path through the machine, and accumulating outputs for each state it visits along each successful path. You only need to ensure that each state is associated with the right output.

2 Algebra warm-up

In class, we looked in detail at a couple of semirings, one for aggregating weights by choosing the maximum and another for aggregating probabilities by summing them up. But there are many more semirings under the sun. Probably the simplest is the Boolean semiring defined by:[1]

```
instance Monoid Bool where
  (<>) = (&&)
  idty = True

instance Semiring Bool where
  summarize = or
```

(3) Write a function `dotprod :: Semiring w => [w] -> [w] -> w`. Given two lists of values `[x1, x2, ..., xn]` and `[y1, y2, ..., yn]`, it should compute `summarize [x1 <> y1, x2 <> y2, ..., xn <> yn]`. If the lists are of different lengths, then the function should just ignore the extra elements at the end of the longer list. You might want to write a helper function `zip :: [a] -> [a] -> [(a,a)]` that pairs the elements of the lists up index-wise (i.e., `zip [x1, ..., xn] [y1, ..., yn] == [(x1,y1), ..., (xn, yn)]`).

```
dotprod :: Semiring w => [w] -> [w] -> w
dotprod = undefined
```

Some test cases:

```
ghci> dotprod [True, False, False] [False, False, True]
False
ghci> dotprod [True, False, False] [True, False, True]
True
ghci> let w :: Weight = dotprod [0.5, 0.75] [0.25, 0.75] in w
0.5625
ghci> let p :: Prob = dotprod [0.5, 0.25] [0.25, 0.75] in p
0.3125
```

(4) Write a function `expn :: Monoid w => w -> Int -> w` which, when given an element `x` and a number `n`, combines `n` copies of `x` via `(<>)`. That is: `x <> x <> ... <> x` (`n` times). You can assume `n` is not negative.

```
expn :: Monoid w => w -> Int -> w
expn = undefined
```

```
ghci> expn 0.5 2
0.25
ghci> expn 0.5 3
0.125
ghci> expn 0.5 0
```

1 Remember, for *any* type to be a semiring, it must first be a monoid, and the `summarize` operation should distribute over the `(<>)`. In the Boolean case, we have `or [p && k, q && k] == k && or [p, q]`

```
1.0
ghci> expn True 2
True
ghci> expn False 2
False
ghci> expn False 0
True
```

3 Parsing with semirings

There are yet other simple semirings we might try out. If you think of the outputs along the arcs of a machine as indicating **costs**, then it's natural to think of paths as **adding up** all the costs along the way. When it comes time to summarize the various totals, we would then be interested in just the *best* possible path, meaning *the one the costs the least*.

There is, however, one catch. Costs can get arbitrarily high. The longer you walk around in the machine, the higher they get, and in principle there's no limit. But the semiring laws will demand that we have some element that is nevertheless "the worst". For this purposes, people often work with the "extended natural numbers": all the usual ones, plus one more called ∞. This new number has two notable properties:

- $x + \infty = \infty + x = \infty$
- $\min(x, \infty) = \min(\infty, x) = x$

In Haskell, we can encode this with the data type `Cost`, which is either a finite number `Fin Int` or is infinite `Inf` (this type is actually declared in the `Algebra.hs` library file).

```haskell
data Cost = Fin Int | Inf deriving (Eq, Show)
```

For instance,

```haskell
costA, costB :: Cost
costA = Fin 17
costB = Inf
```

(5) (a) Define a function `addCost :: Cost -> Cost -> Cost` that computes the sum of two costs.

```haskell
addCost :: Cost -> Cost -> Cost
addCost = undefined
```

(b) Define a function `minCost :: Cost -> Cost -> Cost` that selects the smaller of two costs.

```haskell
minCost :: Cost -> Cost -> Cost
minCost = undefined
```

(c) Provide `Monoid` and `Semiring` instances for `Cost` using these functions.

```haskell
instance Monoid Cost where
  (<>) = undefined
  idty = undefined

instance Semiring Cost where
  summarize []     = undefined
  summarize (c:cs) = undefined
```

When you're done, you should be able to use any of the `Semiring`-constrained functions on `Cost`s. For instance:

```
ghci> reduce fst37 "CVCVCVC"
Fin 2
ghci> reduce fst37 "VCVCVC"
Fin 3
ghci> reduce fst37 "VCVCVCV"
Fin 1
ghci> reduce fst37 "VVVV"
Fin 4
ghci> reduce fst37 "CC"
Inf
```

(6) Define a function that turns an FSA into an FST that calculates the number of possible paths for a given string in the original automaton. This is similar to the `fsaTrace` task above except this time we are going to take advantage of a numerical semiring.

To keep things clean, the `Algebra.hs` file defines a type synonym `Count` together with semiring operations equivalent to the ones for probabilities, but restricted to whole numbers (i.e., `Count`s).

```haskell
type Count = Int

instance Monoid Count where
  (<>) = (*)
  idty = 1

instance Semiring Count where
  summarize []     = 0
  summarize (n:ns) = n + summarize ns
```

The function `fsaCount` converts FSAs to FSTs whose outputs are `Count`s. Again, the `reduce` function (and underneath it, the `transduce'` function) will already do most of this work for you. You just have to specify how to assign `Count`s to the initial/final states and transitions that give you the right result.

```haskell
fsaCount :: FSA -> FST Char Count
fsaCount = undefined
```

Some tests:

```
ghci> reduce (fsaCount fsa7) "VC"
1
ghci> reduce (fsaCount fsa7) "CC"
0
ghci> reduce (fsaCount fsa7) "VCV"
2
ghci> reduce (fsaCount fsa7) "CVVCV"
2
ghci> reduce (fsaCount fsa7) "VCVCV"
4
```