(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'.

Craft a FST that implements this rule. To make things simpler, we will work with a restricted alphabet containing only the `Char`'s `'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 -- the FST reads Char's, writes String's
fstNasalAssimP = undefined
```

(2) Define a transducer that behaves in 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
```

Running `transduce fstNasalAssimM "inmit"` should yield `"immit"`. However, English (unlike, say, Italian and Japanese), does not allow doubled, or *geminate*, consonants. Define a transducer for *degemination*, which turns any occurrences of `"mm"` anywhere in a string into `"m"` (again using the `'i'`, `'t'`, `'n'`, and `'m'` alphabet).

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

(3) 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 `fstNasalAssimM` and `fstElimMM` using `composeFST`. Test your answer by running `transduceBoth "inmit"`.

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

Your answer sequences the two transducers in one order. What happens if the transducers are sequenced in the other order? Why? Answer in a comment:

```
{- Your answer here -}
```

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

```
type FSA sym = ( [State]             -- Q
               , [sym]               -- Sigma
               , [State]             -- I
               , [State]             -- F
               , [Transition sym] )  -- Delta
type Transition sym = (State, sym, State) -- ***
```

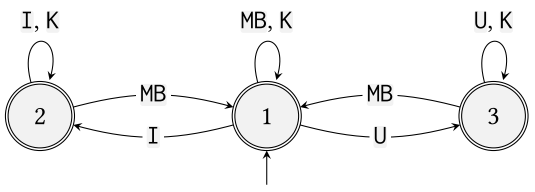
Define a function that turns a FSA into a FST which *traces the path through the machine* — in other words, which yields an output which lists, in order, the states traversed in the course of parsing.

```
fsaToFstTrace :: FSA a -> FST a [State]
fsaToFstTrace (_,_,i,f,delta) = undefined
```

For example, `transduce (fsaToFstTrace fsaHarmony) [I,K,I,K,MB,K,U,K,K,U]` should yield the output `[[1,2,2,2,2,1,1,3,3,3]]`. This reflects that this string of segments is parsed in just one way, in which `I` transitions us to state `2`, `MB` transitions us back to `1`, and `U` transitions us to `3`, where we conclude.

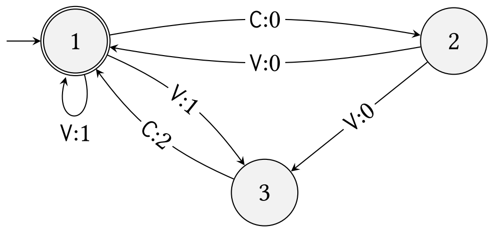
```
data SegmentKIU = K | I | U | MB deriving (Show,Eq)

fsaHarmony :: FSA SegmentKIU
fsaHarmony = (states, syms, i, f, delta)
  where
    states = [1,2,3]
    syms   = [K,I,U,MB]
    i      = [1]
    f      = [1,2,3]
    delta  = [ (1,K,1), (1,MB,1), (1,I,2), (1,U,3),
               (2,K,2), (2,I,2), (2,MB,1),
               (3,K,3), (3,U,3), (3,MB,1) ]
```



Hint: pay close attention to the type I gave to `fsaToFstTrace`. 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. The simplest approach is to use `map`, or list comprehensions (`map f xs` is equivalent to `[f x | x <- xs]`).

(5) Below is a FST which is very similar to one we discussed in class. Think of state #1 (the intial and accepting state) as representing a syllable boundary. This machine associates a **cost of 1** with each vowel that occurs in a syllable without an onset `C`, and a **cost of 2** for each `C` that occurs as a coda.



In the machine from class, weights along a path were accumulated by multiplication, which resulted in syllables without onsets/with codas *reducing* the overall “score” associated with a string. This led us to characterize `mappend` (aka `<>`) and `mempty` as follows for weights:

```haskell
instance Monoid Double where
  mappend = (*)
  mempty  = 1

instance Semigroup Double where -- boilerplate
  (<>) = mappend
```

Thus, the "gluing together of outputs" via `<>` that happens each time we take a `step` in a FST is implemented as multiplication when those outputs are `Double`'s (i.e., decimals); and the initial output we start off with in an initial state, `mempty`, is `1` (see the definitions of `step` and `transduce` in `W9.hs`).

A different notion of accumulation is required for costs, which we will treat as `Int`'s.[1] First, define a Haskell representation of the cost-based FST diagrammed above. Then, tell Haskell how to understand how `Int` costs are accumulated along a path in this FST, by defining appropriate values for `mappend` (aka `<>`) and `mempty`. Remember that `mappend`/`<>` must be **associative** (that is, `(a <> b) <> c == a <> (b <> c)`), and that `mempty` must be an **identity** for `<>` (that is, `m <> mempty == m == mempty <> m`).

```haskell
fstCVCosts :: FST CV Int
fstCVCosts = undefined

instance Monoid Int where
  mappend = undefined
  mempty  = undefined

instance Semigroup Int where -- boilerplate
  (<>) = mappend
```

Check/inspect your answer by running `transduce fstCVCosts [V,V,C]`.

(6) Consider the outputs that `transduce fstCVCosts` associates with `[V,C,V]` and `[C,V,C,V]`. Why do these tranductions have multiple outputs (recall our assumption that state #1 "represents" a syllable boundary)?

```haskell
{- Your answer here. -}
```

There are multiple ways, in principle, to summarize the outputs when a transducer delivers more than one. A natural strategy, for output types that can be ordered, is to report only the **best** of all the outputs. What counts as best depends on the context, and specifically, on what the outputs represent. Sometimes `minimum` is the right notion; other times, `maximum` is. Both `minimum` and `maximum` in Haskell have type `Ord a => [a] -> a`; given a list of orderable things, they return the lowest, or highest (naturally, both `Int`'s and `Double`'s can be ordered!)

Decide which notion of 'best' is appropriate for costs, and then define a function which `transduce`'s a `CV`-string according to `fstCVCosts`, and summarizes the possible outputs by reporting the 'best' one.

```haskell
transduceCVCostsBest :: [CV] -> Int
transduceCVCostsBest str = undefined
```

Test your answer on `[V,V,C]`, `[V,C,V]`, and `[C,V,C,V]`.

1 For example, consider a transduction of `[C,V,C]`. It should have a cost, since it has a coda `C`, but multiplying costs along the only possible parse-path yields 0. This tells us we need a different `Monoid` for costs.