Assignment #3

Due date: Thu. 1/28/2020, 12 noon

Download `FiniteState.hs` and `Assignment03_Stub.hs` from the CCLE site, save them in the same directory, and rename `Assignment03_Stub.hs` to `Assignment03.hs` (please be careful to use this name exactly). You will submit a modified version of `Assignment03.hs` on CCLE. You should not modify or submit `FiniteState.hs`.

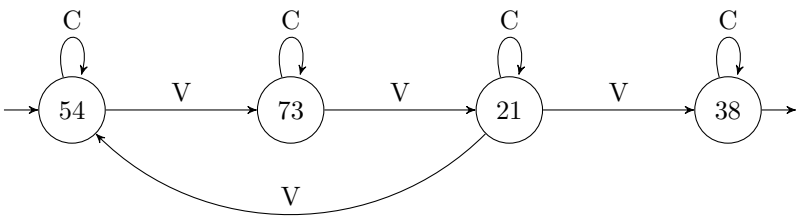
The `import` line near the top of the stub file imports all the definitions from `FiniteState.hs`, so you can use them exactly as you would if they were defined in the same file. Have a look at the code and the comments in `FiniteState.hs` to make sure you know what's going on.

A few things to note:

- I've switched to a slightly different (and probably more familiar-looking) syntax for defining functions, which puts the argument(s) on the left hand side of the `=` sign instead of using explicit lambda, i.e. instead of `generates = \m -> \w -> ...` I've written `generates m w = ...`. But it's important to remember that these are exactly the same; the different syntax does not change the fact that defining a function just *is* giving a name to a lambda expression. Always has been, always will be.
- If you're unsure about the `Eq` stuff, go back to the part of the Tuesday 1/19 lecture on polymorphism, or look at the section on polymorphism and type classes in the "Haskell tips and tricks" document on CCLE.
- If you're unsure about the use of `map` and `or` in the `backward` and `generates` functions, look at the section on `map` in the "Haskell tips and tricks" document on CCLE.
- The `let (states,syms,i,f,delta) = m in ...` part inside the bodies of the `backward` and `generates` functions is something slightly new, but unsurprising once you see what's going on. Remember that `m` is a five-tuple, i.e. something like `fsa_handout5` or `fsa_handout6`, so the `let` expression is giving names to the five components. Here's a simpler example that shows the same thing happening: the expression `let (x,y,z) = (2,3,4) in x + y - z` gives names to the three components of `(2,3,4)` at once, so it evaluates to `2 + 3 - 4`.
- (You may want to skip this when you're getting started and come back to it later, maybe when you start section 4.) A funny quirk of the functions we've seen so far is that none of them make reference to the specified collection of symbols, i.e. the second component of the five-tuple, corresponding to the alphabet Σ when we write an FSA on paper as $M = (Q, \Sigma, I, F, \Delta)$. For example, if we put the empty list there for all our FSAs, things would still "work", because all the information about symbols we need is specified in the transitions. But in principle, the alphabet specified by an FSA can matter: for example, imagine a function that finds all the strings over the relevant alphabet that are *not* generated by a given FSA, i.e. given an FSA $M = (Q, \Sigma, I, F, \Delta)$, find $\Sigma^* - \mathcal{L}(M)$. So, just to keep things neat and tidy, we'll require that the `fsaSanityCheck` function (defined in `Assignment03.hs`) returns `True` on all the FSAs you define below; this function checks that all the states and symbols mentioned in the transitions (i.e. Δ) do appear in the corresponding sets Q and Σ. (Looking at the code for this function to see how it works is a good exercise, too.)

1 Encoding finite-state automata formally

Your task here is to convert the following graphical representation of an FSA into our Haskell format.



Specifically: define `fsa_countVs` to be the appropriate thing of type `Automaton SegmentCV` to represent this FSA. Remember that `Automaton SegmentCV` just shorthand for the type:

```
([Int], [SegmentCV], [Int], [Int], [(Int,SegmentCV,Int)])
```

When you've done this you should check, using `generates`, that the automaton you've defined generates all strings using the alphabet $\{C, V\}$ in which the number of occurrences of 'V' is some non-zero multiple of three. Here are some examples of how it should behave.

```
*Assignment03> fsaSanityCheck fsa_countVs
True
*Assignment03> generates fsa_countVs [V, C, V]
False
*Assignment03> generates fsa_countVs [C, C]
False
*Assignment03> generates fsa_countVs [V, C, V, V, C]
True
*Assignment03> generates fsa_countVs [V, V, V, C, V, V]
False
*Assignment03> generates fsa_countVs [V, V, V, C, V, V, V]
True
```

2 "Snoc lists"

Notice that the built-in Haskell list type, whose definition looks approximately like this:

```
data [a] = [] | a : [a] deriving Show
```

arbitrarily makes the *leftmost* element of a list the *outermost* element (i.e. the one you get your hands on straight away when using a `case` statement).

Sometimes it's convenient to be able to work with lists the other way round, i.e. treating the *rightmost* element as the *outermost* element, the one that gets "peeled off" first when we analyze the list. For this we can define our own type like this (already in `Assignment03.hs`):

```
data SnocList a = ESL | (SnocList a) ::: a deriving Show
```

These are called "snoc lists", because — well, the first kind of lists are called "cons lists". A snoc list is either empty (i.e. `ESL` for "empty snoc list") or comprised of a snoc list and then an element; in the latter case the snoc operator `:::` is what holds them together.

So we might represent the word "hello" as a snoc list as follows:

```
sl :: SnocList Char
sl = ((((ESL ::: 'h') ::: 'e') ::: 'l') ::: 'l') ::: 'o'
```

and then if we were to deconstruct this using a `case` statement like this:

```
case sl of
ESL -> ...
rest:::x -> ...
```

the variable `x` on the last line would refer to the character `'o'`.[1]

A. Write a function `addToFront :: a -> SnocList a -> SnocList a` so that `addToFront x l` returns a snoc list that is like `l` but has an additional occurrence of `x` as its leftmost element.

```
*Assignment03> addToFront 2 (((ESL ::: 3) ::: 4) ::: 5)
(((ESL ::: 2) ::: 3) ::: 4) ::: 5
*Assignment03> addToFront 'x' ((ESL ::: 'y') ::: 'z')
((ESL ::: 'x') ::: 'y') ::: 'z'
*Assignment03> addToFront 'x' ESL
ESL ::: 'x'
*Assignment03> addToFront False (ESL ::: True)
(ESL ::: False) ::: True
```

B. Write a function `toSnoc :: [a] -> SnocList a` that produces the snoc list corresponding to the given "normal list" (i.e. cons list).

```
*Assignment03> toSnoc ['h','e','l','l','o']
((((ESL ::: 'h') ::: 'e') ::: 'l') ::: 'l') ::: 'o'
*Assignment03> toSnoc [3,4,5]
((ESL ::: 3) ::: 4) ::: 5
*Assignment03> toSnoc [True]
ESL ::: True
```

3 Forward values

C. Write a function `forward :: (Eq a) => Automaton a -> SnocList a -> State -> Bool` which computes forward values; `forward m w q` should evaluate to `True` iff there's a way to get from an initial state of the automaton `m` to the state `q` that produces the symbols of `w`. Follow the definition in (19) on the handout. Looking at the implementation of `backward` might also be helpful.

```
*Assignment03> forward fsa_handout4 ((ESL ::: C) ::: V) 43
False
*Assignment03> forward fsa_handout4 ((ESL ::: C) ::: V) 42
True
*Assignment03> forward fsa_handout4 (toSnoc [C,V]) 42
True
*Assignment03> map (forward fsa_handout4 (toSnoc [C,V,V])) [40,41,42,43]
[True,False,True,True]
*Assignment03> map (forward fsa_handout5 (toSnoc [C,V])) [1,2,3]
[True,True,False]
*Assignment03> map (forward fsa_handout5 (toSnoc [C,V,C])) [1,2,3]
[True,False,True]
*Assignment03> map (forward fsa_handout5 (toSnoc [C,V,C,C])) [1,2,3]
[True,False,False]
*Assignment03> map (forward fsa_handout5 (toSnoc [C,V,C,C,V])) [1,2,3]
```

[1] Don't mix up `'h'` and `"h"` — `'h'` is a character (type `Char`), but `"h"` is an abbreviation for `['h']`, i.e. a cons-list-of-characters (type `[Char]`) whose length happens to be one. Similarly, `"hello"` is an abbreviation for `['h','e','l','l','o']`, which of course is just an abbreviation for `'h':('e':('l':('l':('o':[]))))`. But `'hello'` is gibberish.

```
[True,True,False]
```

D. Write a function `generates2 :: (Eq a) => Automaton a -> [a] -> Bool` which checks whether the given automaton generates the given string of symbols. This should produce the same results as the existing function `generates`, but you should use your `forward` function to do it. (See (16) on the handout.)

```
*Assignment03> generates2 fsa_handout5 [C,V,C,C] == generates fsa_handout5 [C,V,C,C]
True
```

4 Designing finite-state automata

The `forward` and `backward` functions might be useful for "debugging" your automata in the following questions.

E. Define an FSA `fsa_twoCs :: Automaton SegmentCV` which has $\{C, V\}$ as its alphabet and generates all and only those strings that contain at least two 'C's. It should behave like this:

```
*Assignment03> fsaSanityCheck fsa_twoCs
True
*Assignment03> generates fsa_twoCs [C,C,C,C]
True
*Assignment03> generates fsa_twoCs [C,V,C,V]
True
*Assignment03> generates fsa_twoCs [V,V,C,V]
False
*Assignment03> generates fsa_twoCs []
False
*Assignment03> generates fsa_twoCs [C]
False
```

F. Define an FSA `fsa_thirdC :: Automaton SegmentCV` which has $\{C, V\}$ as its alphabet and generates all and only those strings that have 'C' as their third symbol. (All of these strings are necessarily of length three or more.) It should behave like this:

```
*Assignment03> fsaSanityCheck fsa_thirdC
True
*Assignment03> generates fsa_thirdC [V,C]
False
*Assignment03> generates fsa_thirdC [V,V,C]
True
*Assignment03> generates fsa_thirdC [V,V,C,C,V]
True
*Assignment03> generates fsa_thirdC [C,C,V,C,C]
False
*Assignment03> generates fsa_thirdC [C,C,C,C,C]
True
```

G. Define an FSA `fsa_thirdlastC :: Automaton SegmentCV` which has $\{C, V\}$ as its alphabet and generates all and only those strings that have 'C' as their third-to-last symbol. (All of these strings are necessarily of length three or more.) It should behave like this:

```
*Assignment03> fsaSanityCheck fsa_thirdlastC
True
```

```
*Assignment03> generates fsa_thirdlastC [C,C,C,C,C]
True
*Assignment03> generates fsa_thirdlastC [C,C,C,V,C]
True
*Assignment03> generates fsa_thirdlastC [C,C,V,V,C]
False
*Assignment03> generates fsa_thirdlastC [C,C,V]
True
*Assignment03> generates fsa_thirdlastC [C,V]
False
```

H. Define an FSA `fsa_oddEven :: Automaton SegmentCV` which has $\{\mathrm{C}, \mathrm{V}\}$ as its alphabet and generates all and only those strings that have an odd number of 'C's and an even number of 'V's (treating zero as an even number). It should behave like this:

```
*Assignment03> fsaSanityCheck fsa_oddEven
True
*Assignment03> generates fsa_oddEven [C]
True
*Assignment03> generates fsa_oddEven [C,V,V]
True
*Assignment03> generates fsa_oddEven [C,C,V]
False
*Assignment03> generates fsa_oddEven [C,C,V,C,V]
True
```

I. Define an FSA `fsa_harmony :: Automaton SegmentPKIU` which has $\{\mathrm{P}, \mathrm{K}, \mathrm{I}, \mathrm{U}, \mathrm{MB}\}$ as its alphabet and enforces a simple kind of *vowel harmony*[2]: if we interpret 'MB' as a morpheme-boundary, all the vowels within a morpheme must be identical to each other.[3] Any strings built out of this alphabet are allowed as long as they satisfy this requirement — this includes some strange ones such as those that contain two adjacent "morpheme boundaries", or have a "morpheme boundary" at the beginning or end, etc., but never mind, the goal is just to isolate out the vowel harmony requirement itself. It should behave like this:

```
*Assignment03> fsaSanityCheck fsa_harmony
True
*Assignment03> generates fsa_harmony [P,K,I,K,MB,U,P,U]
True
*Assignment03> generates fsa_harmony [P,K,I,K,U,P,U]
False
*Assignment03> generates fsa_harmony [K,I,P,I]
True
*Assignment03> generates fsa_harmony [K,P,P,P]
True
*Assignment03> generates fsa_harmony [K,I,P,U]
False
*Assignment03> generates fsa_harmony [K,I,MB,P,U]
True
*Assignment03> generates fsa_harmony [MB,MB,K,MB,P]
True
```

[2] `https://en.wikipedia.org/wiki/Vowel_harmony`

[3] The concept of a morpheme doesn't actually play any role here, it's just a boundary of some sort that splits the string into "chunks" that the vowel harmony constraint applies to.

J. Define an FSA `fsa_MBU :: Automaton SegmentPKIU` which has $\{P, K, I, U, MB\}$ as its alphabet, and generates all and only those strings satisfying the requirement that 'U' can only appear somewhere after (not necessarily immediately after) a 'MB'. (If we assume that 'MB' is used sensibly to mark morpheme boundaries, then this would amount to saying that 'U' cannot appear in the first morpheme of a word. But you should not assume that all input strings are sensible in this way.) It should behave like this:

```
*Assignment03> fsaSanityCheck fsa_MBU
True
*Assignment03> generates fsa_MBU [MB,U]
True
*Assignment03> generates fsa_MBU [U,MB]
False
*Assignment03> generates fsa_MBU [MB]
True
*Assignment03> generates fsa_MBU [MB,K,K,K,K,K,K,U]
True
*Assignment03> generates fsa_MBU [K,K,K,K,K,K,K,U]
False
*Assignment03> generates fsa_MBU [K,K,K,K,K,K,K,K]
True
*Assignment03> generates fsa_MBU [MB,I,I,I]
True
*Assignment03> generates fsa_MBU [MB,U,U,U,U,U]
True
```

K. Define an FSA `fsa_adjacentMBU :: Automaton SegmentPKIU` which has $\{P, K, I, U, MB\}$ as its alphabet, and generates all and only those strings satisfying the requirement that 'U' can only appear *immediately* after a 'MB'. (If we assume that 'MB' is used sensibly to mark morpheme boundaries, then this would amount to saying that 'U' can only appear at the beginning of a morpheme. But you should not assume that all input strings are sensible in this way.) It should behave like this:

```
*Assignment03> fsaSanityCheck fsa_adjacentMBU
True
*Assignment03> generates fsa_adjacentMBU [MB,U]
True
*Assignment03> generates fsa_adjacentMBU [MB,K,K,K,K,K,K,U]
False
*Assignment03> generates fsa_adjacentMBU [MB,U,MB,U]
True
*Assignment03> generates fsa_adjacentMBU [MB,U,MB,U,U]
False
*Assignment03> generates fsa_adjacentMBU [MB,U,U,U,U]
False
*Assignment03> generates fsa_adjacentMBU [MB,I,MB]
True
```