Assignment #4

Due date: Thu. 2/4/2021, 12 noon

Download `FiniteStatePart2.hs` and `Assignment04_Stub.hs` from the CCLE site, save them in the same directory, and rename `Assignment04_Stub.hs` to `Assignment04.hs` (please be careful to use this name exactly). You will submit a modified version of `Assignment04.hs` on CCLE. You should not modify or submit `FiniteStatePart2.hs`. You will also submit a PDF or plain text file on CCLE with written answers to the last section.

The `import` line near the top of the stub file imports all the definitions from `FiniteStatePart2.hs`, so you can use them exactly as you would if they were defined in the same file.

Some things to note before getting started:

- The type for representing ϵ-FSAs (i.e. FSAs with the option to include epsilon transitions) is `EpsAutomaton`, defined in `FiniteStatePart2.hs`. The only difference from the `Automaton` type we've seen before is that the transition labels have type `Maybe sy`, instead of simply `sy`. The definition of `Maybe` types, under the hood, looks like this:

  ```
  data Maybe a = Nothing | Just a
  ```

  So the type `Maybe Char`, for example, has members like `Just 'a'` and `Just 'b'` and `Nothing`. You can sort of think of `Maybe a` as being a bit like the type for lists of `a`s, but with a maximum length of one; or, like a box that is either empty or contains exactly one `a`.[1] We use `Nothing` as the label for epsilon transitions. To see how this works, compare `efsa_handout21` and `efsa_handout22` (also defined in `FiniteStatePart2.hs`) with the corresponding diagrams on the class handout.

- Have a look at the functions `intersect` and `removeEpsilons` in `FiniteStatePart2.hs`, and try to understand how they relate to the corresponding "recipes" for manipulating FSAs given on the class handout. To remind yourself of the example we saw in class, try `intersect fsa_oddCs fsa_evenVs`. It might also be useful to try `removeEpsilons efsa_handout22` and then draw the resulting FSA on paper.

- Notice that the first argument to our `generates` function has type `Automaton st sy`, not `EpsAutomaton st sy`. So if we want to check whether `efsa_handout21` generates a particular `[Char]`, for example, we need to use `removeEpsilons`.[2] Note that `"abb"` is just shorthand for `['a','b','b']`.

  ```
  *Assignment04> generates (removeEpsilons efsa_handout21) "abb"
  True
  *Assignment04> generates (removeEpsilons efsa_handout21) "abbbbb"
  False
  *Assignment04> generates (removeEpsilons efsa_handout21) "abbbbbcc"
  False
  *Assignment04> generates (removeEpsilons efsa_handout22) "abbbbbcc"
  True
  ```

[1] OCaml's `option` types are analogous.

[2] If we're a bit sneaky though, we might notice that `efsa_handout21` also has type `Automaton Int (Maybe Char)`, which means we can also do things like `generates efsa_handout21 [Just 'a', Nothing]`.

- Have a look at the `RegExp` type defined in `FiniteStatePart2.hs`. It should remind you of the `Formula` type from a couple of weeks ago. Convince yourself that the `denotation` function for regular expressions follows the "pencil and paper" definition we saw in class.

1 Strictly-local grammars

A strictly-local grammar (SLG) is an alternative to an FSA: like an FSA, an SLG generates a set of strings over some alphabet. Formally speaking, an SLG is a four-tuple (Σ, I, F, T), where

- Σ is the alphabet of symbols,
- I is a subset of Σ, specifying the allowable starting symbols,
- F is a subset of Σ, specifying the allowable final symbols, and
- T is a subset of $\Sigma \times \Sigma$, i.e. a set of pairs of symbols, specifying the allowable two-symbol sequences (or allowable "bigrams").

1.1 Recognizing strings generated by an SLG

An SLG (Σ, I, F, T) generates a string of n symbols $x_1 x_2 \dots x_n$ iff:

- $x_1 \in I$, and
- for all $i \in \{2, \dots, n\}$, $(x_{i-1}, x_i) \in T$, and
- $x_n \in F$.

Notice that by this definition, there is no way for an SLG to generate the empty string.[3]

For any type `sy` that our chosen symbols belong to, we can straightforwardly represent an SLG in Haskell as a tuple with the type `([sy], [sy], [sy], [(sy,sy)])`, where the four components specify the alphabet, the starting symbols, the final symbols and the allowable bigrams, respectively.

```
type SLG sy = ([sy], [sy], [sy], [(sy,sy)])
```

For example, the SLG in (1) (with `SegmentCV` as its symbol type) generates all strings consisting of one or more `C`s followed by one or more `V`s; and the SLG in (2) (with `Int` as its symbol type) generates all strings built out of the symbols `1`, `2` and `3` which do not have adjacent occurrences of `2` and `3` (in either order). These two SLGs are defined for you with the names `slg1` and `slg2`.

(1) `([C,V], [C], [V], [(C,C),(C,V),(V,V)])`

(2) `([1,2,3], [1,2,3], [1,2,3], [(1,1),(2,2),(3,3),(1,2),(2,1),(1,3),(3,1)])`

Your task here is to write a function

```
generatesSLG :: (Eq sy) => SLG sy -> [sy] -> Bool
```

which checks whether the given string of symbols is generated by the given SLG.

(There are a few different ways to do this, but one way is to write a recursive helper function analogous to `backward` for FSAs (called, say, `backwardSLG`), which then allows a non-recursive implementation of `generatesSLG` that's analogous to `generates` for FSAs.)

Here are some examples of how it should behave:

[3] This is slightly non-standard. The usual definitions of strictly-local grammars in the literature include special start-of-string and end-of-string markers as components of bigrams, which makes it possible to generate the empty string. Also, to be more precise, what I'm describing here are 2-strictly-local grammars, which are a special case of the general idea of a k-strictly-local grammar which specifies allowable substrings of length k.

```
*Assignment04> generatesSLG slg1 [C,C,V]
True
*Assignment04> generatesSLG slg1 [C,V]
True
*Assignment04> generatesSLG slg1 [V]
False
*Assignment04> generatesSLG slg1 [V,C]
False
*Assignment04> generatesSLG slg2 [1]
True
*Assignment04> generatesSLG slg2 [1,2,1,2,1,3]
True
*Assignment04> generatesSLG slg2 [1,2,1,2,1,3,2]
False
*Assignment04> generatesSLG slg2 []
False
```

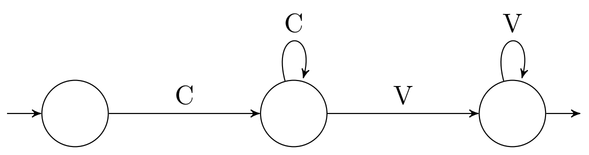
Of course, it should work for all SLGs, not just the two defined above:

```
*Assignment04> generatesSLG (["mwa","ha"],["mwa"],["ha"],[("mwa","ha"),("ha","ha")]) ["mwa","ha"]
True
*Assignment04> generatesSLG (["mwa","ha"],["mwa"],["ha"],[("mwa","ha"),("ha","ha")]) ["mwa"]
False
*Assignment04> generatesSLG (["mwa","ha"],["mwa"],["ha"],[("mwa","ha"),("ha","ha")]) ["mwa","mwa"]
False
*Assignment04> generatesSLG (["mwa","ha"],["mwa"],["ha"],[("mwa","ha"),("ha","ha")]) ["mwa","ha","ha"]
True
```

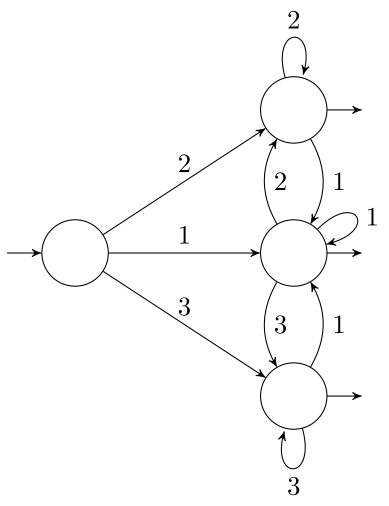
1.2 Conversion to FSAs

It turns out that *any stringset that can be generated by an SLG can also be generated by some FSA*. We know this because, for any SLG there is a mechanical recipe for constructing an FSA that generates exactly the same stringset as that SLG. (But the reverse is not true: some FSAs generate stringsets for which no SLG exists. Try to think of one.) I won't tell you exactly what this recipe is, but given the following examples you should be able to figure it out: applying the recipe to the SLG in (1) produces the FSA in (3), and applying the recipe to the SLG in (2) produces the FSA in (4).

(3)



(4)



Your task here is to write a function **slgToFSA** that takes as input an SLG in the format of (1) and (2), and produces an equivalent FSA. Take a few minutes to make sure you properly understand what the recipe is that has produced (3) and (4), before going on to think about how to actually get this done in Haskell. (Hints: Recall the relationship between the "bucketings" of strings and the states of an FSA. Notice that when L is the stringset generated by an SLG, $u \equiv_L v$ iff the strings u and v end with the same symbol.)

What will the type of this function `slgToFSA` be? Its input can be an SLG with any type at all as its symbol type (i.e. it might have type `SLG SegmentCV`, or `SLG Int`, or `SLG Bool`, etc.). So let's just call this `SLG sy`. Remember that our `Automaton` type has two "type parameters": a type for its symbols and a type for its states. Working out the *symbol type* of the output FSA is easy: this will be the same as the symbol type for the SLG, i.e. the type `sy`, whatever that is. For the *state type* of the output FSA — an SLG has no states, remember! — you should use the type `ConstructedState sy` (where, still, `sy` is whatever type the given SLG's symbols have) which is defined as follows:

(5) `data ConstructedState sy = ExtraState | StateForSymbol sy`

So the type `ConstructedState sy` has one value in it for every value of the type `sy`, plus the additional special value `ExtraState`.[4] (Another hint: the SLG in (2) has *three* symbols, and the equivalent FSA in (4) has *four* states.) The end result then is that `slgToFSA` will have this type:

(6) `slgToFSA :: SLG sy -> Automaton (ConstructedState sy) sy`

When this is working properly, the result of evaluating `slgToFSA slg1` should correspond to the FSA in (3), and the result of evaluating `slgToFSA slg2` should correspond to the FSA in (4). And these results can be used with the existing `generates` function for FSAs: `generates (slgToFSA g) x` should give the same result as `generatesSLG g x`, for any SLG `g` and any string `x`.

```
*Assignment04> generates (slgToFSA slg1) [C,C,V]
True
*Assignment04> generates (slgToFSA slg1) [V,C]
False
*Assignment04> generates (slgToFSA slg2) [1,2,1,2,1,3]
True
*Assignment04> generates (slgToFSA slg2) []
False
*Assignment04> generates (slgToFSA (["mwa","ha"],["mwa"],["ha"],[("mwa","ha"),("ha","ha")])) ["mwa","mwa"]
False
*Assignment04> generates (slgToFSA (["mwa","ha"],["mwa"],["ha"],[("mwa","ha"),("ha","ha")])) ["mwa","ha","ha"]
True
```

[4]This type is exactly equivalent ("isomorphic", in the jargon) to `Maybe sy`, actually, and we could have used that instead. But since we're using that for transitions in ϵ-FSAs it might avoid some confusion to use a different type here.

2 Converting regular expressions into ϵ-FSAs

The eventual goal here will be to write a function

```
reToFSA :: (Eq sy) => RegExp sy -> EpsAutomaton Int sy
```

which converts a regular expression into an equivalent FSA, following the procedure described in class. We'll build up to it in a few steps.

There's a little trick that arises here that arises when you want to "include an FSA in a larger FSA" (in the sense indicated by the diagrams illustrating the RE-to-FSA conversion): you can't assume that all the state names can be left unchanged, because this might create "collisions". For example, notice that:

- `(0,[1],[(0,Just 'a',1)])` is a fine FSA for the regular expression $\underline{a}$, and
- `(0,[1],[(0,Just 'b',1)])` is a fine FSA for the regular expression $\underline{b}$, but
- `(0,[1],[(0,Just 'a',1),(0,Just 'b',1),(1,Nothing,0)])` is a *not* fine FSA for the regular expression $(\underline{a} \cdot \underline{b})$!

The `Either a b` type will be useful here. It's defined for you like this:

```
data Either a b = First a | Second b deriving (Show,Eq)
```

So the type `Either Int Char`, for example, has elements like `First 2`, `First 5`, `Second 'a'` and `Second 'b'`. More interestingly the type `Either Int Int` has "two copies of `Int`" inside it, i.e. this type includes elements such as `First 0`, `First 1` and `First 2`, but also the distinct elements `Second 0`, `Second 1` and `Second 2`.[5]

A. Write a function `unionFSAs` with type

```
(Eq sy) => EpsAutomaton st1 sy -> EpsAutomaton st2 sy -> EpsAutomaton (Either st1 st2) sy
```

which, given an automaton that generates the stringset L and an automaton that generates the stringset L', produces a new automaton that generates the stringset $L \cup L'$.

```
*Assignment04> generates (removeEpsilons (unionFSAs efsa_handout21 efsa_xyz)) "abbb"
True
*Assignment04> generates (removeEpsilons (unionFSAs efsa_handout21 efsa_xyz)) "abbbbb"
False
*Assignment04> generates (removeEpsilons (unionFSAs efsa_handout21 efsa_xyz)) "xxxyz"
True
*Assignment04> generates (removeEpsilons (unionFSAs efsa_handout21 efsa_xyz)) "abbbxxxyz"
False
```

B. Write a function `concatFSAs` with type

```
(Eq sy) => EpsAutomaton st1 sy -> EpsAutomaton st2 sy -> EpsAutomaton (Either st1 st2) sy
```

which, given an automaton that generates the stringset L and an automaton that generates the stringset L', produces a new automaton that generates the stringset $\{u +\!\!+ v \mid u \in L, v \in L'\}$.

```
*Assignment04> generates (removeEpsilons (concatFSAs efsa_handout21 efsa_xyz)) "abbbxxxyz"
True
*Assignment04> generates (removeEpsilons (concatFSAs efsa_handout21 efsa_xyz)) "xxxyzabbb"
False
```

C. Write a function `starFSA` with type

```
EpsAutomaton st sy -> EpsAutomaton (Either Int st) sy
```

[5] Formally what we're doing here is analogous to taking the *disjoint union* of two sets, which is sometimes defined as $X \uplus Y = (\{0\} \times X) \cup (\{1\} \times Y)$. In type theory it's known as a *sum type* — and pair types like `(a,b)` are known as *product types*. (See why?) In Haskell, `Either` is usually defined with the constructors `Left` and `Right` instead of `First` and `Second`, but I think the latter is slightly less confusing.

which, given an automaton that generates the stringset L, produces a new automaton that generates all strings producible but concatenating together zero or more strings from L. (This corresponds to the "star operation" from regular expressions.) (There's a choice for a state number in here which you can make arbitrarily.)

```
*Assignment04> generates (removeEpsilons (starFSA efsa_handout21)) ""
True
*Assignment04> generates (removeEpsilons (starFSA efsa_handout21)) "bbbbb"
True
*Assignment04> generates (removeEpsilons (starFSA efsa_handout21)) "bbabbb"
True
*Assignment04> generates (removeEpsilons (starFSA efsa_handout21)) "aaab"
False
```

OK, that's the hard work relating to FSAs done. In order to put things together in the `reToFSA` function though, you'll need a couple more small helpers ...

D. Write a function `flatten :: Either Int Int -> Int` which squashes all values of type `Either Int Int` back into the type `Int` without ever "collapsing distinctions", i.e. without creating any "collisions". This function can do whatever you want as long as `flatten x /= flatten y` whenever `x /= y`, i.e. no two distinct possible inputs to the function get mapped to the same output. (`x /= y` is Haskell's notation for the negation of `x == y`, i.e. `not (x == y)`.)

(Hint: There are any number of ways to do this, but one strategy involves mapping some inputs to the even numbers, and mapping other inputs to the odd numbers; another strategy involvers mapping some inputs to the positive numbers, and mapping other inputs to the negative numbers ...)

E. Write a function `mapStates` with type

```
(a -> b) -> EpsAutomaton a sy -> EpsAutomaton b sy
```

which produces a new version of the given FSA, with the state labels "updated" according to the given function.

F. Write a function `reToFSA` with type

```
(Eq sy) => RegExp sy -> EpsAutomaton Int sy
```

which produces an automaton that generates the stringset that is the denotation of the given regular expression.

```
*Assignment04> generates (removeEpsilons (reToFSA re2)) "acacbcac"
True
*Assignment04> generates (removeEpsilons (reToFSA re2)) "acacbca"
False
*Assignment04> generates (removeEpsilons (reToFSA re3)) []
True
*Assignment04> generates (removeEpsilons (reToFSA re3)) [3]
False
*Assignment04> generates (removeEpsilons (reToFSA re4)) [0,2,2,2]
True
*Assignment04> generates (removeEpsilons (reToFSA re4)) [1,2,2]
True
*Assignment04> generates (removeEpsilons (reToFSA re4)) [0,1,2,2,2,2,2]
False
*Assignment04> generates (removeEpsilons (reToFSA (Star re4))) [0,1,2,2,2,2,2]
True
```

3 Some nice patterns

Now let's briefly look a bit more at the relationship between regular expressions and propositional formulas, as we defined them a couple of weeks ago.

G. Notice that for any formula ϕ that we pick, its denotation $[\![\phi]\!]$ is going to be equal to $[\![\neg\neg\phi]\!]$. Similarly, $[\![(\phi \wedge \neg\phi)]\!]$ is going to be equal to $[\![\mathbf{F}]\!]$, for any formula ϕ that we choose. When denotations coincide like this we can say that ϕ is equivalent to $\neg\neg\phi$, and that $(\phi \wedge \neg\phi)$ is equivalent to $\mathbf{F}$. For each of the following, say whether it is equivalent to ϕ, equivalent to $\mathbf{T}$, equivalent to $\mathbf{F}$, or not equivalent to any of these.

(a) $(\phi \vee \mathbf{F})$

(b) $(\phi \wedge \mathbf{F})$

(c) $(\phi \wedge \mathbf{T})$

(d) $(\mathbf{T} \vee \mathbf{F})$

(e) $(\mathbf{T} \wedge \mathbf{F})$

H. We can do something analogous for regular expressions. For example, for any regular expression r, its denotation $[\![r]\!]$ is going to be equal to $[\![(r \mid r)]\!]$. (Because $X = X \cup X$, for any set X.) So we can say that r is equivalent to $(r \mid r)$. For each of the following, say whether it is equivalent to r, equivalent to $\mathbf{1}$, equivalent to $\mathbf{0}$, or not equivalent to any of these.

(a) $(r \mid \mathbf{0})$

(b) $(r \cdot \mathbf{0})$

(c) $(r \cdot \mathbf{1})$

(d) $(\mathbf{1} \mid \mathbf{0})$

(e) $(\mathbf{1} \cdot \mathbf{0})$

(Just for fun: (i) Think about $(\phi \wedge (\psi \vee \chi))$. (ii) Can you find some cases where the pattern breaks down?)