Assignment #4

Due date: Thu. 2/6/2020, 10:00am

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`.

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_handout6` and `efsa_handout7` (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_handout7` 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_handout6` 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_handout6) "abb"
  True
  *Assignment04> generates (removeEpsilons efsa_handout6) "abbbbb"
  False
  *Assignment04> generates (removeEpsilons efsa_handout6) "abbbbbcc"
  False
  *Assignment04> generates (removeEpsilons efsa_handout7) "abbbbbcc"
  True
  ```

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

[2] If we're a bit sneaky though, we might notice that `efsa_handout6` also has type `Automaton Int (Maybe Char)`, which means we can also do things like `generates efsa_handout6 [Just 'a', Nothing]`.

- Have a look at the `RegExp` type defined in `Assignment04.hs`. This is almost the same as what we saw a couple of weeks ago, except that I have pulled the symbol type out as a type parameter (`sy`), in the same way that we've been doing for FSAs.

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
```

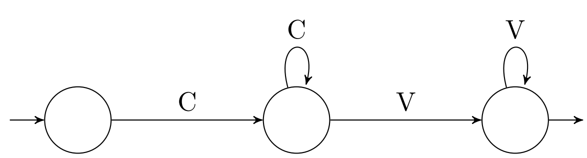
Of course, it should work for all SLGs, not just the two defined above:

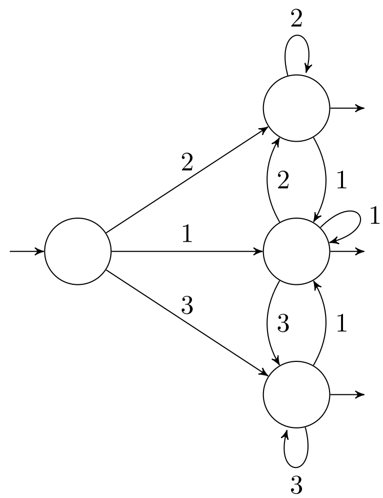
```
*Assignment04> generatesSLG (["mwa","ha"],["mwa"],["ha"],[("mwa","ha"),("ha","ha")]) ["mwa","ha"]
True
*Assignment04> generatesSLG (["mwa","ha"],["mwa"],["ha"],[("mwa","ha"),("ha","ha")]) ["mwa"]
False
*Assignment04> generatesSLG (["mwa","ha"],["mwa"],["ha"],[("mwa","ha"),("ha","ha")]) ["mwa","mwa"]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

The `DisjointUnion a b` type will be useful here. But notice that the applicability of the mathematical notion of "disjoint union" here has nothing to do with the fact that, in one of the exercises below, you need to write a function that computes the "union" of two FSAs; the `DisjointUnion` types will play the same role in computing the "concatenation" of two FSAs, or applying the "star" operation to an FSA.

A. 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.

B. Write a function `flatten :: DisjointUnion Int Int -> Int` which squashes all values of type `DisjointUnion 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. (Functions that have this property are said to be *injective*, or *invertible*.) (`x /= y` is Haskell's notation for the negation of `x == y`, i.e. `not (x == y)`.)

C. Write a function `unionFSAs` with type

```
(Eq sy) => EpsAutomaton st1 sy -> EpsAutomaton st2 sy -> EpsAutomaton (DisjointUnion 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_handout6 efsa_xyz)) "abbb"
True
*Assignment04> generates (removeEpsilons (unionFSAs efsa_handout6 efsa_xyz)) "abbbbb"
False
*Assignment04> generates (removeEpsilons (unionFSAs efsa_handout6 efsa_xyz)) "xxxyz"
True
*Assignment04> generates (removeEpsilons (unionFSAs efsa_handout6 efsa_xyz)) "abbbxxxyz"
False
```

D. Write a function `concatFSAs` with type

```
(Eq sy) => EpsAutomaton st1 sy -> EpsAutomaton st2 sy -> EpsAutomaton (DisjointUnion 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_handout6 efsa_xyz)) "abbbxxxyz"
True
*Assignment04> generates (removeEpsilons (concatFSAs efsa_handout6 efsa_xyz)) "xxxyzabbb"
False
```

E. Write a function `starFSA` with type

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

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_handout6)) ""
True
*Assignment04> generates (removeEpsilons (starFSA efsa_handout6)) "bbbbb"
True
*Assignment04> generates (removeEpsilons (starFSA efsa_handout6)) "bbabbb"
True
*Assignment04> generates (removeEpsilons (starFSA efsa_handout6)) "aaab"
False
```

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
```

Appendix: Hints for `flatten`

I can't give you a sample of illustrative test cases for the `flatten` function, because there are many different functions that satisfy the stated criteria. Here are some hints that might help you get started thinking about it though.

First, an example that would be *incorrect*: if your `flatten` function works like this:

(7)
```
flatten (This 1)  ⟹  3
flatten (That 1)  ⟹  12
flatten (This 2)  ⟹  7
flatten (That 2)  ⟹  3
```

then that's a problem, because it produces the same result for `This 1` as it does for `That 2`.

If it works like this, however:

(8)
```
flatten (This 1)  ⟹  3
flatten (That 1)  ⟹  12
flatten (This 2)  ⟹  7
flatten (That 2)  ⟹  5
```

then it might be correct, because no two distinct inputs here produce the same output value. Similarly, it might be correct if it works like this:

(9)
```
flatten (This 1)  ⟹  41
flatten (That 1)  ⟹  -52
flatten (This 2)  ⟹  171
flatten (That 2)  ⟹  0
```

I say that the function *might* be correct in these last two cases, because it's not only those four inputs that we need to check! If we take the function being tested in (9) and try out some other inputs, we might find this:

(10)
```
flatten (This 1)      ⟹  41
flatten (That 1)      ⟹  -52
flatten (This 2)      ⟹  171
flatten (That 2)      ⟹  0
flatten (This 27)     ⟹  65
flatten (That (-31))  ⟹  65
```

This also means that the function is incorrect, because two different inputs (`This 27` and `That (-31)`) produced the same output. (And it's also incorrect if two different `This` inputs produce the same output, or two different `That` inputs.)

A good way to approach this is to try to think of two sets of integers that are non-overlapping, and then write your function so that all the `This` inputs get mapped to integers in the first set and all the `That` inputs get mapped to integers in the second set. In other words: think of two sets A and B such that $A \subseteq \mathbb{Z}$ and $B \subseteq \mathbb{Z}$ and $A \cap B = \varnothing$, and then make sure that all `This` inputs get mapped to elements of A and all `That` inputs get mapped to elements of B. (It will probably be easier to convince yourself that your function is correct if you think about it abstractly like this than if you try to convince yourself by running lots of tests.)