Assignment 5

Ethan Poole
LING 185A: Comp. Ling. I
Due: 7 May 2019

Instructions: Download `FiniteState.hs` and `Assignment05.hs` from the course website into the same directory on your computer. The **`import`** line near the top of `Assignment05.hs` imports all of the definitions from `FiniteState.hs`, which you may then use in your assignment. Please submit your assignment as one file: a modified version of `Assignment05.hs`.

1 *10 points*

A strictly-local grammar (SLG) is an alternative to an FSA. Like an FSA, an SLG generates a set of strings over some alphabet:

(1) A strictly-local grammar is a four-tuple $\langle \Sigma, S, F, T \rangle$ where:

   a. Σ, the alphabet, is a finite set of symbols;

   b. $S \subseteq \Sigma$ is the set of allowable starting symbols;

   c. $F \subseteq \Sigma$ is the set of allowable final symbols; and

   d. $T \subseteq \Sigma \times \Sigma$ is the set of allowable two-symbol sequences (called "bigrams").

For an SLG to generate a string of n symbols $x_1 x_2 \ldots x_n$, the following must hold: (i) $x_1 \in S$, (ii) $x_n \in F$, and (iii) for all $i \in \{2, \ldots, n\}$, $(x_{i-1}, x_i) \in T$.

For any type `sy` which our chosen symbols belong to, an SLG can be straightforwardly represented in Haskell as a tuple with the following type (technically, a type alias):

(2)
```
type SLG sy = ([sy], [sy], [(sy,sy)])
```

Two examples using this type are given below in (3). The SLG in (3a) has `SegmentCV` as its symbol type and generates all strings consisting of one or more Cs followed by one or more Vs. The SLG in (3b) has **`Int`** as its symbol type and 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 in `FiniteState.hs` with the names `slg1` and `slg2` respectively.

(3) a. `([C], [V], [(C,C),(C,V),(V,V)])`

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

Task: Please write a function `recognizeSLG` that checks whether a given string of symbols is generated by a given SLG:

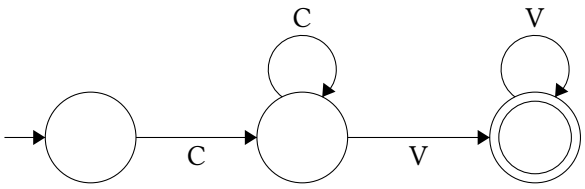
(4) a. **Type signature:**
`recognizeSLG :: (Ord sy) => SLG sy -> [sy] -> Bool`

b. **Example usage:**
`recognizeSLG slg1 [C,C,V]` $\Longrightarrow^*$ **`True`**
`recognizeSLG slg1 [V,C]` $\Longrightarrow^*$ **`False`**
`recognizeSLG slg2 [1,2,1,2,1,3]` $\Longrightarrow^*$ **`True`**
`recognizeSLG slg2 [1,2,1,2,1,3,2]` $\Longrightarrow^*$ **`False`**
`recognizeSLG slg2 []` $\Longrightarrow^*$ **`False`**

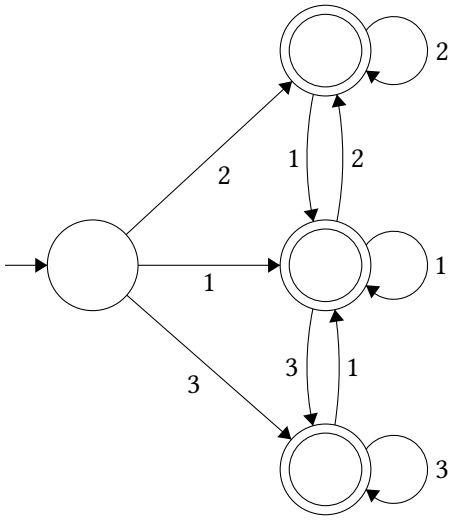
2 *10 points*

Any string set 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 string set (though not vice versa). Part of your task is to figure out what this recipe is—which you should be able to do from (5) and (6): applying the recipe to the SLG in (3a) produces the FSA in (5), and applying the recipe to the SLG in (3b) produces the FSA in (6).

(5)



(6)



Task: Please write a function `slgToFSA` that takes as input an SLG and produces an FSA. Note that the `Automaton` type in `FiniteState.hs` has been simplified so that it does not use 'record' syntax; it is just an ordinary tuple now.

What is the type of `slgToFSA`? Its input can be an SLG with any type as its symbol type. Remember that our FSA 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. For the state type of the output FSA—an SLG has no states after all—, you should use the type `ConstructedState sy` (where `sy` is whatever type the given SLG's symbols are):

(7)
```
data ConstructedState sy = ExtraState | StateForSymbol sy
```

Putting all of this together, the type of `slgToFSA` is thus:

(8)
```
slgToFSA :: SLG sy -> Automaton (ConstructedState sy) sy
```

You may test your function using `recognize`: `recognize (slgToFSA g) x` should give the same result as `recognizeSLG g x`, for any SLG `g` and any string `x`.

3 *10 points*

Please write a function `reToFSA` that converts regular expressions to FSAs. You should break this task up into two parts.

First, you should write the functions `unionFSAs`, `concatFSAs`, and `starFSA`, with the following types:

(9)
```
unionFSAs :: (Ord sy) => EpsAutomaton Int sy
    -> EpsAutomaton Int sy -> EpsAutomaton Int sy

concatFSAs :: (Ord sy) => EpsAutomaton Int sy
    -> EpsAutomaton Int sy -> EpsAutomaton Int sy

starFSA :: (Ord sy) => EpsAutomaton Int sy
    -> EpsAutomaton Int sy
```

These three functions should implement the recipes for constructing new ϵ-FSAs out of existing ϵ-FSAs, which we need in order to implement the overall recipe for constructing an ϵ-FSA out of a regular expression, as described in the class handout on ϵ-FSAs.

Second, those three functions should be used as part of the definition of `reToFSA`:

(10)
```
reToFSA :: (Ord sy) => RegExp sy -> EpsAutomaton Int sy
reToFSA (Lit x) = ...
reToFSA (Alt r1 r2) =
    unionFSAs (reToFSA r1) (reToFSA r2)
reToFSA (Concat r1 r2) =
    concatFSAs (reToFSA r1) (reToFSA r2)
reToFSA (Star r) = starFSA (reToFSA r)
reToFSA ZeroRE = ...
reToFSA OneRE = ...
```

Putting all of this together should have the following behavior:

(11)
```
recognize (removeEpsilons (reToFSA re2)) "acacbcac" ⟹* True
recognize (removeEpsilons (reToFSA re2)) "acacbca" ⟹* False
recognize (removeEpsilons (reToFSA re3)) [] ⟹* True
recognize (removeEpsilons (reToFSA re3)) [3] ⟹* False
recognize (removeEpsilons (reToFSA re4)) [0,2,2,2] ⟹* True
recognize (removeEpsilons (reToFSA re4)) [1,2,2] ⟹* True
recognize (removeEpsilons (reToFSA re4)) [0,1,2,2,2,2,2]
                                                    ⟹* False
recognize (removeEpsilons (reToFSA (Star re4))) [0,1,2,2,2,2,2]
                                                    ⟹* True
```

To help you out, I have provided the helper function `ensureUnused`. Given a list of "reserved" integers and an FSA that uses integers for its states, this function produces a new equivalent FSA that is guaranteed not to use any of the reserved integers for its states. While it is not important exactly *how* this function works, it will be useful on this part of the assignment because if you want to "include" an FSA α in a larger FSA β, you need to make sure that the states of α are kept distinct from all the other states in β.