Assignment 3

Ethan Poole
LING 185A: Comp. Ling. I
Due: 21 April 2020

Instructions: Download `RegEx.hs`, `SLG.hs`, and `Assignment03.hs` from the course website into the same directory on your computer. The **`import`** line near the top of `Assignment03.hs` imports all of the definitions from `RegEx.hs` and `SLG.hs`, which you may then use in your assignment. Please submit your assignment as a modified version of `Assignment03.hs`.

Note: To make the instructions more readable, I do not include the type-class declarations here (namely for **`Eq`**), but they are given in `Assignment03.hs`.

1 *4 points*

As warm-up, please code up the following utility functions for working with SLGs.

(1) a. `bigrams :: [a] -> [(a, a)]` which takes a list of expressions of type `a` and returns the bigrams over that list.

Example usage:
`bigrams [1,2,3]` $\Longrightarrow^*$ `[(1,2),(2,3)]`
`bigrams "cat"` $\Longrightarrow^*$ `[('c','a'),('a','t')]`
`bigrams []` $\Longrightarrow^*$ `[]`

b. `pretty :: [(a, a)] -> [a]` which takes a list of "chained" bigrams and returns them as a simple list. For the sake of simplicity, let `pretty` return the empty list if the bigrams are not well-chained. The function `isChained` is provided in `SLG.hs` to check this for you!

Example usage:
`pretty [(1,2),(2,3)]` $\Longrightarrow^*$ `[1,2,3]`
`pretty [('c','a'),('a','t')]` $\Longrightarrow^*$ `"cat"`
`pretty [(1,2),(3,4)]` $\Longrightarrow^*$ `[]`
`pretty []` $\Longrightarrow^*$ `[]`

2 *10 points*

One of the most important aspects of a formal grammar is what string set it generates. This exercise is designed for you to explore the string sets generated by SLGs. I have broken down the exercise into incremental steps, which you should complete in order!

First, it is helpful to consider what are called "forward" and "backward" strings. Consider $x \in \Sigma$ for an SLG G $\langle \Sigma, S, F, T \rangle$. The FORWARD STRINGS are the sub-strings that can be generated *starting* from x, according to the transitions in T. (These are sub-strings because they may or may not actually be generated by G depending on whether the first and last characters of each sub-string are in S and F respectively.) In the same vein, the BACKWARD STRINGS are the sub-strings that can be generated according to T, *ending* with x.

Task 1: Please code up the following functions that calculate what could come after x or could come before x according to an SLG G.

(2) a. `follows :: SLG sy -> sy -> [sy]` such that `follows g x` returns the list of characters that can follow `x` according to the SLG `g`.

Example usage:

`follows g1 V` $\Longrightarrow^*$ `[C,V]`

`follows g2 "the"` $\Longrightarrow^*$ `["cat","very","fat"]`

`follows g2 "cat"` $\Longrightarrow^*$ `[]`

b. `precedes :: SLG sy -> sy -> [sy]` such that `precedes g x` returns the list of characters that can precede `x` according to the SLG `g`.

Example usage:

`precedes g1 V` $\Longrightarrow^*$ `[C,V]`

`precedes g2 "the"` $\Longrightarrow^*$ `[]`

`precedes g2 "cat"` $\Longrightarrow^*$ `["the","fat"]`

Tips: Do not forget about **`map`** and **`filter`**!

Task 2: Please code up the following functions that calculate forward and backward strings. These functions should make use of `follows` and `precedes` from above.

(3) a. `forward :: SLG sy -> Int -> sy -> [[sy]]` such that `forward g n x` returns the list of possible sub-strings that start with `x` and take between 0 and `n` transitions, according to the SLG `g`.

Example usage:

`forward g1 1 C` $\Longrightarrow^*$ `[[C],[C,V]]`
`forward g1 1 V` $\Longrightarrow^*$ `[[V],[V,C],[V,V]]`
`forward g1 2 C` $\Longrightarrow^*$ `[[C],[C,V,C],[C,V,V],[C,V]]`
`forward g1 2 V` $\Longrightarrow^*$ `[[V],[V,C,V],[V,V,C],[V,V,V],[V,C],[V,V]]`

(See Assignment03.hs for more examples.)

b. `backward :: SLG sy -> Int -> sy -> [[sy]]` such that `backward g n x` returns the list of possible sub-strings that take between 0 and `n` transitions and end with `x`, according to the SLG `g`.

Example usage:

`backward g1 1 C` $\Longrightarrow^*$ `[[C],[V,C]]`
`backward g1 1 V` $\Longrightarrow^*$ `[[V],[C,V],[V,V]]`
`backward g1 2 C` $\Longrightarrow^*$ `[[C],[C,V,C],[V,V,C],[V,C]]`
`backward g1 2 V` $\Longrightarrow^*$ `[[V],[V,C,V],[C,V,V],[V,V,V],[C,V],[V,V]]`

(See Assignment03.hs for more examples.)

Tips: Make use of **`nub`** and **`concat`**! **`nub`** takes a list and removes all of the duplicate elements. **`concat`** takes a list of lists and concatenates them. `forward` and `backward` will need to be recursive. It may be helpful to use a "worker wrapper", a simple helper function that sets up the initial input to a recursive function. In this case, it will be much easier to write the recursive parts of `forward` and `backward` if the second argument of each function were of type `[[sy]]` rather than of type `sy`. I will leave it to you to figure out why …

Task 3: You are now in a position to calculate string sets that an SLG actually generates, as opposed to just sub-strings. Please code up the following function that calculates the set of strings that an SLG generates by taking up to some arbitrary number of transitions. Note that this function will not be recursive, and it should make use of `forward` (or `backward`) from above.

(4) `generates :: SLG sy -> Int -> [[sy]]` such that `generate g n` returns the list of strings that the SLG g generates by taking between 1 and `n` transitions.

Example usage:

`generates g1 1` $\Longrightarrow^*$ `[[C,V]]`
`generates g1 2` $\Longrightarrow^*$ `[[C,V,V],[C,V]]`
`generates g2 1` $\Longrightarrow^*$ `[["the","cat"]]`
`generates g2 2` $\Longrightarrow^*$ `[["the","fat","cat"],["the","cat"]]`

Tips: Do not forget about **`map`**, **`filter`**, and **`concat`**!

3 *4 points*

To begin familiarizing yourself with the `RegEx` type, please code up the following functions that extend the utility of our regular-expression implementation in Haskell. Note that I cannot provide example usage for these functions without giving away the answers!

(5) a. `occurrences :: Int -> (RegEx a) -> (RegEx a)` such that `occurrences n r` returns a `RegEx` that would match exactly *n* occurrences of *r*.

b. `optional :: (RegEx a) -> (RegEx a)` such that `optional r` returns a `RegEx` that would match either zero occurrences or one occurrence of *r*.