Computer Science 320 (Fall, 2016)
Concepts of Programming Languages

Homework Assignment 4: Modularity and Higher-order List Functions

Out: Thursday, October 6, 2016
Due: Saturday, October 22, 2016, by 11:59 pm

This assignment should be entirely written as a Haskell script. There is no Python script to be turned in.

The shortest superstring problem is encountered in both DNA sequencing and data compression. The problem is defined in the following way: given a list of strings $S = \{s_1, \ldots, s_n\}$, what is the shortest possible string s^* such that every string $s_i \in S$ is a substring of s^*? In this assignment, you will use higher-order functions to implement several algorithms for solving this problem. All your solutions should be defined within a module named `Superstring`, and you should submit the file `Superstring.hs` (or `Superstring.lhs`). **File names are case sensitive.** Verbal responses to non-programming questions should be in the form of comments in your code.

Note: All function definitions at top level must have explicit type annotations. Unless otherwise specified, your solutions may utilize functions from the standard prelude as well as material from lecture notes.

(25 pts) **Problem 1.** All the algorithms you implement will be polymorphic. “Strings” will be represented as lists of elements of some type `a`. In your module, add the following type synonym declaration:

```
type Str a = [a]
```

Henceforth, whenever we talk about strings, we are referring to values of type `Str a`.

(a) Define a function `overlap :: Eq a => (Str a, Str a) -> Int` which takes two strings s and s' and returns the number of elements at the end of s which overlap with the beginning of s'. For example:

```
 overlap ("fire", "red")  =  2
overlap ("water", "blue")  =  0
```

You may want to use the function `isPrefixOf`, found in the prelude. To use it, you will need to add `import List` immediately after your module declaration.

(b) Define a function `contains :: Eq a => Str a -> Str a -> Bool` which returns `True` only if the second string is a substring of the first string. Otherwise, it returns `False`.

(c) Define an infix operator `o :: Eq a => Str a -> Str a -> Str a` which concatenates two strings s and s' by merging the overlapping elements at the end of s and the beginning of s'. For example:

```
 "fire" `o` "red"  =  "fired"
"water" `o` "blue"  =  "waterblue"
```

(d) Using `foldr` and `(o)`, define a superstring algorithm `naive :: Eq a => [Str a] -> Str a` that takes a list of strings, and constructs a superstring containing every string in that list.

(10 pts) **Problem 2.**

(a) Define a function `maximize :: Eq a => (a -> Int) -> a -> a -> a` which takes a function f of type `a -> Int`, and two arguments of type `a`. It returns the argument which maximizes the function f.

(b) Define a function `minimize :: Eq a => (a -> Int) -> a -> a -> a` which takes a function f of type `a -> Int` along with two arguments, and returns the argument which minimizes the function f.

(30 pts) **Problem 3.** You will now define the optimal superstring algorithm.

(a) Using `filter`, define a function `update :: Eq a => [Str a] -> (Str a,Str a) -> [Str a]` which takes a list of strings ℓ, and a pair of strings (s, s'). The function should combine the two strings s and s' (taking advantage of any overlapping) to obtain a new string s''. It should remove from the list of strings ℓ any strings contained in s'', and should then return a list which contains all of the remaining strings, as well as one copy of s''. For example:

```
update ["fire", "red", "blue"] ("fire", "red")  =  ["fired","blue"]
```

(b) Define a function `allPairs :: Eq a => [Str a] -> [(Str a,Str a)]` which takes a list of strings, and returns a list of all pairs of strings which can be made out of this list, *excluding* any pairs of equal strings.

(c) Define a recursive algorithm `superstring :: Eq a => ([Str a] -> [(Str a,Str a)]) -> [Str a] -> Str a` that takes as arguments a function *next* for generating pairs of strings out of a list, and a list of strings ℓ.

It should take the list of strings and generate a list of pairs using the supplied function. For *every* pair, it should generate a new list in which *only* that pair is combined, and should call itself recursively on this new, smaller list of strings (this can be done with either `map` or list comprehensions; remember that the recursive call is also expecting f as an argument). Once all the recursive calls have returned a result, it should choose the shortest result, and return it (you can use `foldr`, `length`, and `minimize` for this; for the base case of `foldr`, you can use `naive`). (Note: This can all be done in a single line, so plan carefully and use list functions wherever possible.)

Given an empty list, `superstring` should return an empty list. Given a list with only one string, it should return that string.

(d) Define a superstring algorithm `optimal :: Eq a => [Str a] -> Str a` which takes a list of strings, and tries all possible ways of combining the strings, returning the best result. (Hint: All you need are your solutions from parts (b) and (c) above.)

Because it tries all possibilities, this algorithm always returns the shortest possible superstring, but runs in exponential time. Nevertheless, your implementation should be able to handle the following test cases:

```
test1  =  ["ctagcgacat", "aagatagtta", "gctactaaga", "gacatattgt", "tagttactag"]
test2  =  ["101001","010100010100", "100101", "001010", "11010", "100", "11010"]
test3  =  [x++y | x<-["aab","dcc","aaa"], y<-["dcc", "aab"]]
```

If you get a stack overflow on any of these examples, make sure that in `superstring`, recursive calls are made on progressively smaller lists, and that you are filtering out pairs of strings which are equal to each other.

(30 pts) **Problem 4.** You will now define three greedy superstring algorithms. For all three of the functions below, you may assume that the list supplied as an argument will always have at least two elements.

(a) Define a function `firstPair :: Eq a => [Str a] -> [(Str a,Str a)]` that takes a list of strings, and returns a list containing only one element: the first two strings as a tuple.

(b) Define a function `bestWithFirst :: Eq a => [Str a] -> [(Str a,Str a)]` that takes a list of strings, and returns a list containing a single pair of strings. The first component in the pair should be the first string in the list. The second component in the pair should be the string in the list which has the largest overlap with the first string.

(c) Define a function `bestPair :: Eq a => [Str a] -> [(Str a,Str a)]` that takes a list of strings, and returns a list containing the one pair which has the largest overlap.

By applying `superstring` to one of the the above functions, it is possible to produce one of three distinct greedy algorithms. Because each function produces a list with only a single element, these algorithms will not take exponential time.

(d) Define a superstring algorithm `greedy :: Eq a => [Str a] -> Str a` which produces at least as short a superstring as any of the above three algorithms.

(5 pts) **Problem 5.** One advantage of higher-order functions is that it is easy to write generic testing scripts.

(a) Define a function `compare :: Eq a => ([Str a] -> Str a) -> ([Str a] -> Str a) -> [Str a] -> Double` which takes two superstring algorithms and a single list of strings, and returns the ratio between the length of the superstring generated by the first algorithm, and the length of the one generated by the second algorithm. You will need to use `fromIntegral` to convert to a `Double`.

(b) For the three test cases presented in **Problem #3**, how do the results of the four greedy algorithms compare to the optimal superstring?