Coverage & Logistics

This homework covers material through the seventh chapter of *Haskell: The Craft of Functional Programming* (HCFP). This homework is officially due in class on **Thursday, March 2**. However, it comes with an automatic extension: anything submitted to the CIS 252 bin near CST 4-226 by **noon on Friday, March 3** will be accepted as on time.

You may work singly or in pairs on this assignment.

What to turn in

The same general grading criteria from Assignment 2 applies for this assignment as well. You should submit hard copies of (1) your source code and (2) a clean transcript demonstrating convincingly that your code is correct. As always, include a completed disclosure cover sheet.

Exercises

Some of these functions are easier to write recursively, and others may be easier to write using list comprehensions or built-in functions. Unless otherwise specified, you may use whichever strategy you prefer. However, you should be striving for simple, easy-to-read code: if your code is unduly complicated, you may lose some points.

1. Write a Haskell function `duplicates::String -> Bool` that takes a string and determines whether any characters appear more than once in that string. For example:

```
*Main> duplicates "Two or more"
True
*Main> duplicates "Exactly four"
False
```

2. Use a **list comprehension** to write a **one-line** Haskell function `zap::Char -> String -> String` such that `zap ch cs` returns the string obtained from `cs` by removing all occurrences of `ch`. For example:

```
*Main> zap 'a' "Abracadabra"
"Abrcdbr"
```

3. Write a Haskell function `unique::String -> String` such that `unique cs` returns a string that contains those characters that occur exactly once in `cs`. For example:

```
*Main> unique "Abracadabra"
"Acd"
```

Hint: *`zap` is useful here.*

4. For the purposes of this question, let's introduce the following vocabulary related to strings (as a caveat, some other sources may define these terms differently, but these definitions apply for this assignment):

   - A string `xs` is a **prefix** of string `ys` provided that `ys` can be obtained by adding zero or more elements to the end of `xs`.

     For example, `"comp"` is a prefix of `"computer science"`, but `"comp sci"` is not a prefix of `"computer science"`.

   - A string `xs` is a **subsequence** of string `ys` provided that `xs` can be obtained by removing zero or more elements from `ys`.

     For example, both `"comp sci"` and `"music"` are subsequences of `"computer science"` (i.e., `"computer science"`), but neither is a subsequence of `"mustard cider"`.

   - A string `xs` is a **substring** of string `ys` provided that `ys` can be obtained by adding zero or more elements to the front of `xs` and adding zero or more elements to the end of `xs`. That is, the string `xs` appears as a sequence of consecutive elements in the string `ys`.

     For example, `"sci"` is a substring of `"computer science"`, but `"music"` is not a substring of `"computer science"`.

   Note that the empty string `""` is a prefix, subsequence, and substring of every string, and every string a prefix, subsequence, and substring of itself.

   (a) Write a Haskell function `prefix::String -> String -> Bool` such that `prefix xs ys` determines whether `xs` is a prefix of `ys`. For example:

```
*Main> prefix "music" "musician"
True
*Main> prefix "music" "musingcian"
False
```

(b) Write a Haskell function `subseq::String -> String -> Bool` such that `subseq xs ys` determines whether `xs` is a subsequence of `ys`. For example:

```
*Main> subseq "music" "computer science"
True
*Main> subseq "music" "mustard cider"
False
```

(c) Write a Haskell function `substring::String -> String -> Bool` such that `substring xs ys` determines whether `xs` is a substring of `ys`. For example:

```
*Main> substring "music" "computer science"
False
*Main> substring "mile" "smiled"
True
```

Hint: *`prefix` is useful here.*

(d) Write a Haskell function `subsequences::String -> [String]` such that `subsequences xs` returns a list containing all of the subsequences of `xs`. For example:

```
*Main> subsequences "abcb"
["abcb","bcb","acb","cb","abb","bb","ab","b","abc",
 "bc","ac","c","ab","b","a",""]
```

Hint: *Every subsequence of a string `cs` gives rise to* two *subsequences of `c:cs`: itself, plus the result of placing `c` at its front. (For a more explicit hint, ask in class!)*

None of these functions requires more than five lines of code (not counting the type declaration), and most of them can be written using only two or three lines. If your answers are significantly longer than that, then your approach is too complicated: please ask for assistance.