PLC: Homework 1 [100 points]

Due date: Wednesday, February 6th
4 extra-credit points if you turn it in by Tuesday, February 5th

About This Homework

For this homework, you will practice writing functions by recursion and pattern-matching in Haskell.

You are free to write helper functions as needed for any problems.

How to Turn In Your Solution

You should create a directory called hw1 (exactly this!) in your personal repository, and add your Haskell files to this directory. Then push the directory (and all your Haskell files) to your github.uiowa.edu repop. You should copy the files from the hw1 directory of the course repo to your own hw1 directory, before you start modifying them. We may release changes to the original homework files if a bug is reported, for example, so you do not want to modify the original files, just your copies in the hw1 directory of your personal repo.

As for hw0, you can check that you have submitted correctly by going to github.uiowa.edu in a web browser and checking that you can see your submitted files in your repo there.

Partners Allowed

You may work by yourself or with one partner (no more). Only one partner should submit the solution by adding the Haskell files to his/her personal repository. Both partners should submit, in their hw1 directories, files called partner.txt. Each file contains the hawkid of the other partner. So if jan and ben are partners, jan should include a partner.txt file that contains just the word ben. Similarly, ben's partner.txt file should contain just the word jan. This protocol is to ensure that both partners agree that they are submitting the solution together. You are free to divide up the problems and tackle them separately, or to work on problems together.

How To Get Help

You can post questions in the hw1 section on Piazza.

You are also welcome to come to our office hours. See the course's Google Calendar, linked from the README.md file of the github.uiowa.edu page for the class, for the locations and times for office hours.

1 Reading

Read Chapters 3, 4, 5, and 6 of the required book, *Programming in Haskell*, by Graham Hutton.

2 Basic Problems [42 points]

In the file `TableTags.hs` is the start of some code dealing with the HTML tags for tables (you can search online for "html table tags" to find basic background information – but the problems below do not require this). There are apparently a bunch more tags than the ones we will include below, but these are the basics.

1. Fill in the definition of the `TableTag` datatype to give a single constructor for each of the table tags `table`, `tr`, `th`, and `td`. The constructor's name should be the same as the tag except starting with a capital letter: `Table`, `Tr`, `Th`, `Td` (because Haskell requires constructor names to start with a capital letter). [6 points]
2. Fill in the definition of `showTableTag` to turn `TableTag`s into strings for the tag name. `Table` should turn into `"Table"`, etc. You will notice that the code I am providing you already makes `TableTag` an instance of the `Show` class, so `TableTag`s can be printed by ghci, once you define `showTableTage`. [6 points]
3. Fill in the definition of `equalTableTags` to return `True` when the two input `TableTag`s are exactly the same (like `Tr` and `Tr`), and `False` otherwise (like `Table` and `Td`). [6 points]
4. Fill in the definition of `directElt` to say which tag is for an entity that can be a direct component of which other tag [6 points]:
   - a `tr` (table row) can be a direct element of a `table`
   - a `td` (table data) can be a direct element of a `tr`
   - a `th` (table header) can be a direct element of a `tr`
   - a `table` can be a direct element of a `td` or `th` (this is actually allowed, it seems)
   - that is it: your code should return `False` for all other pairs
5. Now skip ahead in `TableTags.hs` just a little for the following, which concern the recursive datatype `TableHtml`, intended to represent the HTML for a table:
   - Implement functions `getSubelts`, `hasTag`, and `rowLength`, according to the descriptions in the comments above those functions. [6 points each]

3 Intermediate Problems for TableHtml [25 points]

1. Fill in the definition of `showTable` to turn a `TableHtml` object into a `String`. Do not add any extra spaces or newlines to your output (this is actually easier to code, though it makes the output pretty hard to read). So for the `testTable` in `Tests.hs`, the output should be

```
<table><tr><td>hi<td/><td>bye<td/><tr/><tr><td>open<td/><td>shut<td/><tr/><table/>
```

This is worth 10 points.

2. Fill in the definition of `tableOk` to check that the given `TableHtml` is really a valid table. The things you should check are:

   - The input has `Table` as its table tag
   - As you descend into the structure of the table, the pairs of tags you see are allowed by the `directElt` function you wrote above.
   - The rows of the table all have the same length

   I found I had to write a couple helper functions for this. The problem is worth 15 points. You can find some test tables in `Tests.hs`.

4 Intermediate Problems for SnocLists [21 points]

In `SnocLists.hs`, you will find a definition of a recursive datatype `SList`, representing lists where the head is the second argument and the tail is the first to the `Scons` constructor (corresponding to the usual `(:)` constructor for Haskell's built-in lists).

1. Fill in the definitions of `sappend`, `slength`, and `smap`, corresponding to `(++)` (append), `length`, and `map` on Haskell's built-in lists. Each function is worth 7 points.

5 Challenge Problems [12 points]

1. Fill in the definitions of `sfilter`, `sintersperse`, and `sconcat` in `SnocLists.hs`, so that they behave in corresponding manner to the Haskell list operations `filter`, `intersperse`, and `concat` (see the documentation for these functions on Hoogle). These are worth 2 points each.

2. In a file called `More.hs` (which you must create), define a function called `descendings` which, given a list of elements of type `a` where `a` is in the `Ord` type class, return a list of lists of `a` elements, consisting of the maximal descending subsequences of the input list. For example, given `[100,1,4,2,3,2,1]`, your function should return `[[100,1],[4,2],[3,2,1]]`. Getting the correct type for this (which you should write as part of your code) is worth 2 points, and the correct code is then 4 more points.