Advanced Programming (CSE 399)
Homework Assignment 3

Due Tuesday, February 1, at noon

This assignment involves processing and transforming XML documents. You will write Haskell functions to collect some simple statistics from an XML document and to transform an XML document to an HTML representation.

To keep things simple, we will not deal with the full generality of XML. We will represent XML documents as instances of the following simplified type:

```
data SimpleXML =
        PCDATA  String
      | Element ElementName [SimpleXML]
    deriving Show

type ElementName = String
```

That is, a `SimpleXML` value is either a `PCDATA` ("parsed character data") node containing a string or else an `Element` node containing a tag and a list of sub-nodes.

A collection of files that form the starting point of this assignment is available on the Schedule page under the main course web page. Begin by grabbing these files, unpacking them, and making sure that you can successfully run the main program (in `Main.hs`). We've provided a `Makefile`, which you can use if you like. You should see this:

```
THESEUS gets 48 speeches in this play!
```

Next, have a look at the provided files.

- `Main.hs` is the main program. This is where you will put your own code, and it's the (only!) file you will turn in.
- `XMLTypes.hs` contains the type definitions for our simplified XML trees.
- `Play.hs` contains a sample XML value. To avoid getting into details of parsing actual XML concrete syntax, we'll work with just this one value for purposes of this assignment.
- `sample.html` contains an (very basic) HTML rendition of the same information as `Play.hs`. You may want to have a look at it in your favorite browser.
- `samplestats.txt` contains some interesting statistical information about this play. Have a look at it too.
- `RedBlack.hs` contains the code for the implementation of red-black trees presented in class. It isn't used by the code we've provided, but you'll need it for the first part of the assignment.

Your tasks are as follows:

1. The red-black trees we saw in class can easily be extended so that, instead of just maintaining a set, they maintain a set of key/value pairs. Finite maps have operations similar to sets, except that when we insert a key into a finite map, we include an associated value, and when we look up a key, instead of just a boolean we get back the corresponding value (or `Nothing` if the key is not present).

   ```
   emptyMap :: FiniteMap a b
   lookupMap :: Ord a => a -> FiniteMap a b -> Maybe b
   insertMap :: Ord a => a -> b -> FiniteMap a b -> FiniteMap a b
   ```

   Implement these functions. (Headers are already provided in `Main.hs`. You'll want to copy and change the code from `RedBlack.hs`.)

2. The XML value in `Play.hs` has the following structure (in standard XML syntax):

   ```
   <PLAY>
      <TITLE>TITLE OF THE PLAY</TITLE>

      <PERSONAE>
        <PERSONA> PERSON1 </PERSONA>
        <PERSONA> PERSON2 </PERSONA>
        ... -- MORE PERSONAE
      </PERSONAE>

      <ACT>
         <TITLE>TITLE OF FIRST ACT</TITLE>
         <SCENE>
             <TITLE>TITLE OF FIRST SCENE</TITLE>
             <SPEECH>
                 <SPEAKER> PERSON1 </SPEAKER>
                 <LINE>LINE1</LINE>
                 <LINE>LINE2</LINE>
                 ... -- MORE LINES
             </SPEECH>
             ... -- MORE SPEECHES
         </SCENE>
         ... -- MORE SCENES
      </ACT>

      ... -- MORE ACTS
   </PLAY>
   ```

   Fill in the definition of a function `countAllSpeeches` that walks over an XML structure of this form and builds a finite map that counts the number of speeches given by each `SPEAKER`. You may want to use our function `countSpeeches` as a starting point.

   The `main` action that we've provided will use your function to generate a summary file `speeches.txt`. The contents of this file after your program runs must be *character for character identical* to what is in `samplespeeches.txt`. The `main` action that we've written will check this for you and report where they differ.

3. *Important:* This is the interesting part of the assignment—the part where you get to do some design. You will be graded not only on correctness (producing the required output), but also on the elegance of your solution and the clarity and readability of your code and documentation. I.e., style counts. It

is strongly recommended that you rewrite this part of the assignment a couple of times: get something working, then step back and see if there is anything you can abstract out or generalize and rewrite it, then leave it for a few hours and rewrite it again. Try to use the higher-order programming techniques we've been discussing in class.

The HTML in `sample.html` has the following structure (with whitespace added for readability).

```
<html>
  <body>
    <h1>TITLE OF THE PLAY</h1>
    <h2>Dramatis Personae</h2>
    PERSON1<br>
    PERSON2<br>
    ...
    <h2>TITLE OF THE FIRST ACT</h2>
    <h3>TITLE OF THE FIRST SCENE</h2>
    <b>PERSON1</b><br>
    LINE1<br>
    LINE2<br>
    ...
    <b>PERSON2</b><br>
    LINE1<br>
    LINE2<br>
    ...
    <h3>TITLE OF THE SECOND SCENE</h2>
    <b>PERSON3</b><br>
    LINE1<br>
    LINE2<br>
    ...
  </body>
</html>
```

Fill in the definition of a function `playToHtml` that walks over an XML structure and returns a string containing HTML of this form (but with no whitespace except what's in the textual data in the original XML).

The `main` action that we've provided will use your function to generate a file `dream.html`. The contents of this file after your program runs must be *character for character identical* to `sample.html`. Our `main` action will test this.

Submission instructions:

- Email (just) the file `Main.hs` to `bcpierce@cis.upenn.edu`.
- Make sure this file is accepted by Hugs (or GHC) without errors and follows the other rules in the Style Guide on the course web page. In particular, please put your name in a comment at the top of the file.
- Remember that style counts!