Some standard Haskell functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>not</td>
<td>Bool -&gt; Bool</td>
</tr>
<tr>
<td>(&amp;&amp;), (</td>
<td></td>
</tr>
<tr>
<td>(+++)</td>
<td>[a] -&gt; [a] -&gt; [a]</td>
</tr>
<tr>
<td>(:)</td>
<td>a -&gt; [a] -&gt; [a]</td>
</tr>
<tr>
<td>concat</td>
<td>[[a]] -&gt; [a]</td>
</tr>
<tr>
<td>concatMap</td>
<td>(a -&gt; [b]) -&gt; [a] -&gt; [b]</td>
</tr>
<tr>
<td>delete</td>
<td>(Eq a) =&gt; a -&gt; [a] -&gt; [a]</td>
</tr>
<tr>
<td>drop, take</td>
<td>Int -&gt; [a] -&gt; [a]</td>
</tr>
<tr>
<td>elem</td>
<td>(Eq a) =&gt; a -&gt; [a] -&gt; Bool</td>
</tr>
<tr>
<td>filter</td>
<td>(a -&gt; Bool) -&gt; [a] -&gt; [a]</td>
</tr>
<tr>
<td>foldl</td>
<td>(a -&gt; b -&gt; a) -&gt; a -&gt; [b] -&gt; a</td>
</tr>
<tr>
<td>foldr</td>
<td>(a -&gt; b -&gt; b) -&gt; b -&gt; [a] -&gt; b</td>
</tr>
<tr>
<td>fst</td>
<td>(a,b) -&gt; a</td>
</tr>
<tr>
<td>snd</td>
<td>(a,b) -&gt; b</td>
</tr>
<tr>
<td>head, last</td>
<td>[a] -&gt; a</td>
</tr>
</tbody>
</table>
| init, tail, reverse | [a] -> [a] |}
| isLower, isUpper | Char -> Bool |
| length      | [a] -> Int                                |
| map         | (a -> b) -> [a] -> [b]                    |
| null        | a -> [a] -> Bool                          |
| product, sum| (Num a) => [a] -> a                       |
| zip         | [a] -> [b] -> [(a,b)]                     |
| zipWith     | (a -> b -> c) -> [a] -> [b] -> [c]       |

**Question 1 (4 points).** Write a function

\[
\text{countCaps :: String} \rightarrow \text{Int}
\]

that given a String, returns the number of upper case letters in the string. (Obvious hint: isUpper may be useful.)

*Examples:*

\[
(\text{countCaps "FoOBaR"}) \sim 4
\]

\[
(\text{countCaps "e.e.cummings"}) \sim 0.
\]

**Question 2 (6 points).** Write a function

\[
\text{partition :: (a} \rightarrow \text{Bool} \rightarrow [a] \rightarrow ([a],[a])}
\]
such that if \(\text{partition test xs} \sim (\text{passes},\text{fails})\), then passes is a list of all those elements of xs on which test returns True and fails is a list of all the elements of xs on which test returns False. Note that \(\text{partition test []}\) should return \(([],[]))\).

**Question 3 (6 points).** This problem involves the following data-type for multiway trees.

\[
data \text{Multi} = \text{Fork Int [Multi]} \text{ deriving (Eq,Show)}
\]

For multiway trees, a leaf is a Fork-node of the form: \((\text{Fork k []})\).

Define a function

\[
\text{listLeaves :: Multi} \rightarrow \text{[Int]}
\]
such that \((\text{listLeaves t})\) returns a list of all (and only) the numbers in leaf nodes of t. (Hint: \text{map} or \text{list comprehension} may be useful here along with \text{concat, concatMap, etc.})

*Example:*

\[
(\text{listLeaves (Fork 2 [Fork 10 [], Fork 3 []])}) \sim [10,3]
\]

**Question 4 (4 points).** This problem involves the following data-type for binary trees.

\[
data \text{BinTree} = \text{Empty} \mid \text{Branch Int BinTree BinTree}
\]

\text{deriving (Eq,Show)}

*Definitions:*

(i) The weight of a BinTree t is the sum of all the Int-values in t. Empty has weight 0.

(ii) A BinTree t is weight-balanced exactly when its subtrees are weight-balanced and the weight of its left subtree equals the weight of its right subtree. Empty is balanced.

Write a function

\[
\text{bal :: BinTree} \rightarrow \text{(Int,Bool)}
\]
such that \((\text{bal t}) \sim (w,b)\) where w is the weight of t and, if t is weight-balanced, b is True; otherwise b is False.
Possible answers for 1

```haskell
countCaps cs = length [c | c <- cs, isUpper c]
countCaps' "" = 0
countCaps' (c:cs)
  | isUpper c = 1 + countCaps' cs
  | otherwise = countCaps' cs
```

Possible answers for 2

```haskell
partition tst [] = ([], [])
partition tst (x:xs)
  | tst x = (x:ys, zs)
  | otherwise = (ys, x:zs)
  where (ys, zs) = partition tst xs

partition' tst xs = ([x | x<-[xs], tst x], [x | x<-xs, not(tst x)])
```

Possible answers for 3

```haskell
listLeaves (Fork k []) = [k]
listLeaves (Fork _ ts) = concatMap listLeaves ts

listLeaves' (Fork k []) = [k]
listLeaves' (Fork _ ts) = getLeaves ts
  where getLeaves [] = []
       getLeaves (t:ts) = listLeaves' t ++ getLeaves ts
```

A possible answer for 4

```haskell
bal Empty = (0, True)
bal (Branch n tl tr) = (n + nl + nr, ball && balr && (nl == nr))
  where
    (nl, ball) = bal tl
    (nr, balr) = bal tr
```