Problem Sheet 7.1: Tail Recursion

Tail Recursion

In this exercise, we will look to implement an efficient implementation of a function that generates the **Fibonacci** numbers using tail recursion. We have given you a function `badFib :: Int -> Integer` which given an input n, will output the nth Fibonacci number. However, if you try using this `badFib` on values beyond just 35, you should notice ghci producing answers slower than even some speedy humans would!. The commented simulation of comptation below `badFib` shows why this is: the computation increases exponentially with n. We can do better than this.

a) Our function `goodFib` will use a tail recursive auxiliary function `fibAcc` with two accumulators to generate the Fibonacci numbers efficiently. Complete the definition of both functions so that it returns the correct values for n. Unlike `badFib`, `goodFib` should have no problem calculating the 100th Fibonacci number - even the 1000th and 10000th should be manageable!

```
*Main> goodFib 100
354224848179261915075
```

b) To understand better why `goodFib` is so much more efficient than `badFib` complete the commented calculation of `goodFib 8`.

c) We can implement a function that generates the entire list of Fibonacci numbers using similar tail recursive principles. Using the functions `zipWith` and `tail`, complete the definition of `fibList`. Again, it should have no problem finding the first 1000 Fibonacci numbers!

```
*Main> take 10 fibList
[1,1,2,3,5,8,13,21,34,55]
```

d) To understand why this function is also efficient, fill in the commented calculation until it produces 8, the 6th Fibonacci number.

Difficult Recursion Exercises

Following on from last week, here are some more tricky fuctions to try and implement in three ways:

(i) Using explicit recursion (i.e., defined on an empty list and a list split into head and tail).

(ii) Using list comprehension.

(iii) Using the inbuilt recursive funtions: `map`, `filter`, `zip`, `foldl`, `foldr` etc.

a) The function `wordScore` that, given a string, returns the sum "score" of its letters, where the letter `'a'` or `'A'` scores one, `'b'` or `'B'` scores two, etc. until `'z'` or `'Z'` scores 26. Any other symbol gets score zero. You can use (but there are other ways!) the following helpful functions (partly from `Data.Char`):

- `toUpper` and `toLower` to change the case of a letter,
- `ord` to get the ASCII integer index of a letter,
- `subtract` to get from `ord 'A'` to 1; the minus symbol `(-)` doesn't make sections easily, since e.g. `(-3)` is interpreted as the integer "negative three".

Use `where` clauses where needed.

b) The function `concatCheapWords` which, given a list of words, selects the ones with a `wordScore` of 42 or less, adds a space at the front of each, and then concatenates them.