1 `filter`-ing and `map`-ping over `String`s

(1) Write a function that decodes some (very lightly) encrypted text by removing all upper-case letters. Test it on `"cALKJoJmIpKKlJKiKnQgQQU"`. Hint: use `filter`.

```
decryptCaps :: String -> String
decryptCaps xs = undefined
```

(2) Another way to encrypt text is to shift each character one letter forward in the alphabet. For example, "linguistics" becomes "mjohvjtujdt". Let's define a function that does this for us. Haskell makes it easy: `Char` is an ordered (technically, enumerable) type, which means we can use a library function called `succ` to get the alphabetically next character: `succ 'a' == 'b'`. Hint: use `map`.

```
encryptShift :: String -> String
encryptShift xs = undefined
```

Haskell also defines a `pred` function on ordered (technically, enumerable) types going in the other direction: `pred 'b' == 'a'`. Use it to define a function that decrypts a message encrypted by `encryptShift`.

```
decryptShift :: String -> String
decryptShift xs = undefined
```

What does `\xs -> decryptShift (encryptShift xs)` do to a `String`?

```
{- Put your answer between these braces -}
```

(3) What if we had a text that was encrypted like in (1), *and* like in (2), with both distracting upper-case letters and shifted lower-case ones. Show how to decode it by **composing** your two decryption functions. Check that `doubleDecrypt "mJjQoDDhZIvIjAtujMMdtWET"` evaluates to `"linguistics"` in `ghci`.

```
doubleDecrypt :: String -> String
doubleDecrypt xs = undefined
```

Does the order you composed the decryption functions in make any difference? Why or why not?

```
{- Put your answer between these braces -}
```

(4) Write a function that reports the longest word in a list of words, together with its length. (Big) hint: your answer should `map withLength` (defined below) over the starting list, `sort` the resulting list, and (finally) use `last` to get the longest.

```
longestWord :: [String] -> (Int, String)
longestWord xs = undefined

withLength :: String -> (Int, String)
withLength word = (length word, word)
```

2 Working with a real corpus

(5) Adapt the code from this week's slides (in `W3.hs`) to clean and tokenize the Brown corpus (in `aux/brown.txt`), group word tokens together into types, pair each type with its count in a sorted list, and find the most common one, together with its raw count.

This file loads `W3.hs`, so you have access to everything defined there! Be careful with indents. Make sure that each line of the `do`-block that you add is indented the same as the lines I've already put there.

```haskell
main :: IO ()
main = do
  -- *** load it
  -- *** clean it

  -- *** tokenize it
  let howManyTokens = undefined -- *** count em

  -- *** group by types
  let howManyTypes = undefined  -- *** count em

  -- *** add counts and sort
  let mostCommon = undefined    -- *** get it

  -- some code that prints the number of tokens, number of types, and the
  -- most common word, when you load this file into ghci and call `main`:
  putStrLn ("Total tokens: " ++ show (howManyTokens :: Int))
  putStrLn ("Total types: " ++ show (howManyTypes :: Int))
  putStrLn ("Most common word is: " ++ show (mostCommon :: (Int, String)))
```

The `show` functions I've sprinkled in are there to convert your answers into `String`s as demanded by `putStrLn`: `putStrLn 2` is a type error, but `putStrLn (show 2)` works.