Problem Sheet 1.2: Lists, Pattern Matching and Pairs

Lists

a) We give the function `halving` which takes as input an integer and outputs a list of integers. Before loading the Haskell file, try to work out what list `halving` will produce for a given input. Now test on a few different inputs to se if you were correct.

b) **Collatz sequences** are a well-studied mathematical phenomenon. They are generated by a simple rule:

- If the number is even, divide it by two.
- If the number is odd, triple it and add one.

The Collatz sequence for an integer `n` is the sequence that starts with `n`, and ends when `1` is first reached. It is an open problem whether there are any infinite Collatz sequences. Complete the definition of `collatz`, so it returns the Collatz sequnce corresponding to the input integer.

```
*Main> collatz 7
[7,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1]
```

c) Using your `collatz` function and the standard function `length`, give the type signature for and implement the function `colLength` which returns the length of the Collatz sequence for a given input.

```
*Main> colLength 9
20
```

Pattern Matching

A powerful and efficient way to write functions on lists is using **pattern matching**.

a) The function `maxList` should return the maximum value from a list of integers. We have filled in the first two cases: for the empty list and for singletons. Complete the function by giving the recursive case for when the list has at least two elements.

```
*Main> maxList [2,4,3,3,7,63,266,1]
266
```

b) The function `allDucks` should take a list of strings, returning `True` if every string is exactly “duck”, and `False` otherwise. Implement `allDucks`.

```
*Main> allDucks ["duck","duck","duck"]
True
*Main> allDucks ["duck","chicken","duck"]
False
```

c) The function `duckDuckGoose` should take a list of strings, returning `True` if every string except the last is “duck”, the final string is “goose”, and `False` otherwise. Implement `duckDuckGoose` . N.B. the list is allowed to contain no “duck”s, but must contain exactly one “goose”.

```
*Main duckDuckGoose ["duck","duck","duck"]
False
*Main duckDuckGoose ["duck","duck","duck",''goose'']
True
```

Pairs

Using lists of pairs is a useful way of storing structured information in Haskell. You should see the list `ducks` that stores the names and ages of various ducks.

a) Complete the function `noDDucks` , which inputs a list of pairs of the same format as `ducks` , returning a list containing only those elements where the name of the duck does not begin with th character ’D’. Hint: remember that the type `String` is equal to `[Char]` . Therefore, we can use standard functions on lists on strings.

```
*Main> noDDucks ducks
[("Huey",2),("Louie",2)]
```

b) In fact, for `noDDucks` , we don’t care about the age of ducks. Change the type signature and function so that it returns only the names, not the ages of the ducks. *Main> noDDucks ducks ["Huey","Louie"]

c) Add a type signature and complete the function `YoungOrShort` , which returns `True` if any ducks in the list are less than 3 years old, or if any ducks have a name of three or fewer letters.

```
*Main> youngOrShort ducks
True
```

d) Using the function `show` which converts an integer into a string, and the function `++` which concatenates strings, give the type signature and implement the function `describeDucks` which takes as input lists such as `Ducks` and outputs a string like the following. Feel free to personalise the details of your function.

```
ghci> describeDucks ducks
"Donald is a duck who is 6 years old. Daisy is a duck
who is 5 years old. Huey is a duck who is 2 years old.
Louie is a duck who is 2 years old. Dewey is a duck who
is 2 years old. "
```