Declarative Programming – Lab 2

Comprehension and Recursion

In these problems you'll be asked to define several functions in two ways: first with a list comprehension and second with recursion. These two definitions should *not* depend on one another. The recursive version should not mention the list-comprehension version, and vice-versa.

To allow the two solutions to these problems to co-exist in one file, you need to give them different names. For this tutorial, use the given name for the list-comprehension version, and append `Rec` to the name for the recursive version. For example, `halveEvens` should be a function using a list comprehension and `halveEvensRec` should be a recursive function that behaves the same.

In addition, to verify that both functions work in the same way, you will write and run an appropriate QuickCheck test. You will find the skeletons of the functions in the file `lab2.hs`, which came packaged with this document.

Note: for these exercises you may not use any library functions other than the ones stated. If you have an additional solution using other library functions, you're welcome to discuss it during the tutorial.

Exercises

1. a. Write a function `halveEvens :: [Int] -> [Int]` that returns half of each even number in the list. For example,

    ```
    halveEvens [0,2,1,7,8,56,17,18] == [0,1,4,28,9]
    ```

    Your definition should use a *list comprehension*, not recursion. You may use the functions

    ```
    div, mod :: Int -> Int -> Int.
    ```

    b. Write an equivalent function `halveEvensRec`, this time using *recursion*, not a list comprehension. You may use `div` and `mod` again.
    c. To confirm the two functions are equivalent, write a test function `prop_halveEvens` and run the appropriate QuickCheck test.

2. a. Write a function `inRange :: Int -> Int -> [Int] -> [Int]` to return all numbers in the input list within the range given by the first two arguments (inclusive). For example,

    ```
    inRange 5 10 [1..15] == [5,6,7,8,9,10]
    ```

    Your definition should use a *list comprehension*, not recursion.
    b. Write an equivalent function `inRangeRec`, using *recursion*.
    c. To confirm the two functions are equivalent, write a test function `prop_inRange` and run the appropriate QuickCheck test.

3. a. Write a function `countPositives` to count the positive numbers in a list (the ones strictly greater than 0). For example,

    ```
    countPositives [0,1,-3,-2,8,-1,6] == 3
    ```

    Your definition should use a *list comprehension*. You may not use recursion, but you will need a specific library function (there is an overview of the most common list functions in the tables prelude-fn1.png and prelude-fn2.png).
    b. Write an equivalent function `countPositivesRec`, using *recursion* and without using any library functions.
    c. To confirm the two functions are equivalent, write a test function `prop_countPositives` and

run the appropriate QuickCheck test.

   d. Why do you think it's not possible to write countPositives using only list comprehension, without library functions?

4.   a. Professor Pennypincher will not buy anything if he has to pay more than £199.00. But, as a member of the Generous Teachers Society, he gets a 10% discount on anything he buys. Write a function `pennypincher` that takes a list of prices and returns the total amount that Professor Pennypincher would have to pay, if he bought everything that was cheap enough for him.

Prices should be represented in Pence, not Pounds, by integers. To deduct 10% off them, you will need to convert them into floats first, using the function `fromIntegral`. To convert back to ints, you can use the function `round`, which rounds to the nearest integer. You can write a helper function `discount :: Int -> Int` to do this. For example,

```
pennypincher [4500, 19900, 22000, 39900] == 41760
```

Your solution should use a *list comprehension*, and you may use a library function to do the additions for you.

   b. Write an equivalent function `pennypincherRec`, using *recursion* and without using library functions. You may use your function `discount`.

   c. To confirm the two functions are equivalent, write a test function `prop_pennypincher` and run the appropriate QuickCheck test.

5.   a. Write a function `multDigits :: String -> Int` that returns the product of all the digits in the input string. If there are no digits, your function should return 1. For example,

```
multDigits "The time is 4:25" == 40

multDigits "No digits here!" == 1
```

Your definition should use a *list comprehension*. You'll need a library function to determine if a character is a digit, one to convert a digit to an integer, and one to do the multiplication.

   b. Write an equivalent function `multDigitsRec`, using *recursion*. You may use library functions that act on single characters or integers, but you may not use library functions that act on a list.

   c. To confirm the two functions are equivalent, write a test function `prop_multDigits` and run the appropriate QuickCheck test.

6.   a. Write a function `capitalise :: String -> String` which, given a word, capitalises it. That means that the first character should be made uppercase and any other letters should be made lowercase. For example,

```
capitalise "edINBurgH" == "Edinburgh"
```

Your definition should use a *list comprehension* and library functions `toUpper` and `toLower` that change the case of a character.

   b. Write a recursive function `capitaliseRec`. You may need to write a helper function; of the helper function and the main function only one needs to be recursive.

   c. To confirm the two functions are equivalent, write a test function `prop_capitalise` and run the appropriate QuickCheck test.

7.   a. Using the function `capitalise` from the previous problem, write a function

```
title :: [String] -> [String]
```

which, given a list of words, capitalises them as a title should be capitalised. The proper capitalisation of a title (for our purposes) is as follows: The first word should be capitalised. Any other word should be capitalised if it is at least four letters long. For example,

```
title ["tHe", "sOunD", "ANd", "thE", "FuRY"]

== ["The", "Sound", "and", "the", "Fury"]
```

Your function should use a *list comprehension*, and not recursion. Besides the `capitalise` function, you will probably need some other auxiliary functions. You may use library functions that change the case of a character and the function `length`.

b. Write a *recursive* function `titleRec`. You may use `capitaliseRec` and any of its auxiliary functions.

c. To confirm the two functions are equivalent, write a test function prop_title and run the appropriate QuickCheck test.

```
title ["tHe", "sOunD", "ANd", "thE", "FuRY"]

== ["The", "Sound", "and", "the", "Fury"]
```