Comprehensions
Informatics 1 – Introduction to Computation
Functional Programming Tutorial 2

Week 2 (23-27th Sep.)

Attendance at tutorials is obligatory; please send email to lambrose@ed.ac.uk if you cannot join your assigned tutorial.

Assessment is formative, meaning that marks from coursework do not contribute to the final mark. But coursework is not optional. If you do not do the coursework you are unlikely to pass the exams.

Each tutorial involves a mixture of individual homework as a preparation step, followed by work and discussion done by groups of five to six students during the tutorial meeting. You are being encouraged to work together and help each other throughout the whole tutorial. A number of tutors will also be available to give help or advice. If at any point your group needs assistance please raise your hand to draw our attention.

Good Scholarly Practice: The tutorial exercises in this course are not for credit and so the University requirements on academic misconduct do not apply. You are encouraged to collaborate and to seek help from others when you get stuck. But please do not publish solutions to these exercises on the internet or elsewhere, to prevent other students from just googling your solutions without learning. Please note that when, in other courses, work is for credit, you must abide by the regulations. These permit general discussions but they must stop well short of an actual solution.

1 Homework: List Comprehension

The present tutorial is all about understanding *list comprehensions*, to do so you will be asked to use them to define several functions. You will also write and run some QuickCheck tests to test basic properties of your code. The skeletons of the functions can be found in the file `tutorial2.hs`, which came packaged with this document.

Note: For the following exercises you are allowed to use some common library functions, which can be found on the course's Learn page. You would refer to this list many times throughout the tutorial, so you may find it useful to print it as well. If you have an additional solution using other library functions or your own functions, you are welcome to discuss it during the tutorial.

Exercise 1

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

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

Your definition should use a *list comprehension*. Consider to use the library's functions for `div` and `mod`.

(b) Write a test function `prop_halveEvens` to check that your `halveEvens` function gives the same results as the reference implementation `halveEvensReference`.

The `halveEvensReference` function is predefined in the exercise file. Do *not* try to understand how it works, it is written in a deliberately obtuse style.

(c) Fill out the *Answer Sheet* in Section 2 of the tutorial with the body of the aforementioned functions `[1a, 1b]`.

Exercise 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*.

(b) Fill out the *Answer Sheet* in Section 2 with the body of the aforementioned function `[2a]`.

Exercise 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 might want to use a specific library function for a part of the implementation.

(b) Fill out the *Answer Sheet* in Section 2 with the body of the aforementioned function `[3a]`.

(c) Why do you think it's not possible to write `countPositives` using only list comprehension, without library functions?

Exercise 4

(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 a function `countDigits :: String -> Int` that returns the number of digits in the input string.

(c) Because 9 is the largest digit, the number returned by `multDigits` on any given input should be less than or equal to 9^x where x is the number of digits as returned by `countDigits`. Write and execute a QuickCheck property `prop_multDigits` to confirm. The exponentiation operator is `(^)`, e.g. `9 ^ 3` $= 9^3 = 729$.

(d) Fill out the *Answer Sheet* in Section 2 with the body of the aforementioned functions `[4a, 4b, 4c]`.

Exercise 5

(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 the library functions `toUpper` and `toLower` that change the case of a character.

Hint: Use pattern matching to decompose the input string into the first character and the rest.

(b) Fill out the *Answer Sheet* in Section 2 with the body of the aforementioned function `[5a]`.

Exercise 6

(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*. 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) Fill out the *Answer Sheet* in Section 2 with the body of the aforementioned functions `[6a, 6b]`.

Exercise 7

(a) Write a function `sign :: Int -> Char` that takes an integer and returns the character

- `'+'` if the integer is between 1 and 9 (inclusive),
- `'0'` if the integer is 0,
- `'-'` if the integer is between -1 and -9 (inclusive),

and indicates an error otherwise (using the `error` function).

(b) Write a function `signs :: [Int] -> String` that takes a list of integers and returns the sign of each integer between -9 and 9 (inclusive), ignoring any number out of that range. For example, `signs [5, 10, -5, 0]` should return `"++-0"`. You might want to use the `sign` function defined earlier.

(c) Fill out the *Answer Sheet* in Section 2 with the body of the aforementioned functions `[7a, 7b]`.

Exercise 8

(a) Write a function `score :: Char -> Int` that converts a character to its score. Each letter starts with a score of one; one is added to the score of a character if it is a vowel (a, e, i, o, u) and one is added to the score of a character if it is upper case; a character that is not a letter scores zero. For example,

```
score 'A' = 3
score 'a' = 2
score 'B' = 2
score 'b' = 1
score '.'  = 0
```

(b) Write a function `totalScore :: String -> Int` that given a string returns the *product* of the score of every letter in the string, ignoring any character that is not a letter. For example, `totalScore "aBc4E"` should return 12. The `product` function might come in handy.

(c) Write a test function `prop_totalScore_pos` that checks that `totalScore` always returns a number greater than or equal to 1.

(d) Fill out the *Answer Sheet* in Section 2 with the body of the aforementioned functions `8a, 8b, 8c`.

2 Answer Sheet

Please fill in your name as another member of your group will check your answers.

Name:

Exercise	
1a: `halveEvens xs`	
1b: `prop_halveEvens xs`	
2a: `inRange lo hi xs`	
3a: `countPositives list`	
4a: `multDigits str`	
4b: `countDigits str`	
4c: `prop_multDigits xs`	

5a: `capitalise s`	
6a: `lowercase xs`	
6b: `title _`	
7a: `sign i`	
7b: `signs xs`	
8a: `score x`	
8b: `totalScore xs`	
8c: `prop_totalScore_pos xs`	

3 Tutorial Activities

Exercise 9

Compare each of your answers in the *Answer Sheet* with a buddy sitting next to you. Try to convince each other to agree on an answer and if possible use a computer to check your implementations.

Exercise 10

(a) Switch pairs and work with the person sitting next to you to fill out the table below with the body of the following functions `[10b, 10c]`.

10b (helper function): `discount price`	
10b: `pennypincher prices`	
10c: `prop_pennypincher xs`	

(b) 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. For example,

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

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

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

(c) To confirm that Professor Pennypincher actually saves money, write a test function `prop_pennypincher`. This should check that the result of `pennypincher` is less or equal to the sum of the positive numbers in the input.

4 Optional Material

In this final part of the tutorial we will use *list comprehensions* to write some more involved functions. Please note that you may **NOT** have time to complete this part during the tutorial; you can do it at home afterwards if you want.

Exercise 11

(a) Dame Curious is a crossword enthusiast. She has a long list of words that might appear in a crossword puzzle, but she has trouble finding the ones that fit a slot. Write a function

```
crosswordFind :: Char -> Int -> Int -> [String] -> [String]
```

to help her. The expression

```
crosswordFind letter inPosition len words
```

should return all the items from `words` which (a) are of the given length and (b) have `letter` in the position `inPosition`. For example, if Curious is looking for seven-letter words that have `'k'` in position 1, she can evaluate the expression:

```
crosswordFind 'k' 1 7 ["funky", "fabulous", "kite", "icky", "ukelele"]
```

which returns `["ukelele"]`. (*Remember that we start counting with 0, so position 1 is the second position of a string.*)

Your definition should use a *list comprehension*. You may also use a library function which returns the nth element of a list, for argument n, and the function `length`.

Exercise 12

(a) Write a function `search :: String -> Char -> [Int]` that returns the positions of all occurrences of the second argument in the first. For example

```
search "Bookshop" 'o'  ==  [1,2,6]
search "senselessness's" 's'  ==  [0,3,7,8,11,12,14]
```

Your definition should use a *list comprehension*. You may use the function `zip :: [a] -> [b] -> [(a,b)]`, the function `length :: [a] -> Int`, and the term forms `[m..n]` and `[m..]`.

(b) Try to come up with a property of `search` that should always hold. Write a QuickCheck test to confirm it does.

Exercise 13

(a) Write a function `contains` that takes two strings and returns `True` if the first contains the second as a substring. You can use the library function `isPrefixOf`, which returns `True` if the second string begins with the first string, and any list function on page 127 of Thompson; see the course webpage for a copy if you are using Lipovaca. For example,

```
contains "United Kingdom" "King"  ==  True
contains "Appleton" "peon"  ==  False
contains "" ""  ==  True
```

Your definition should use a *list comprehension*. A hint: you can use the library function `drop` to create a list of all possible suffixes ("last parts") of a string.

(b) Write a QuickCheck property `prop_contains` to test your function.

You could test positive or negative results (or both) with specifically crafted strings where you know that one does contain the other, or not. Or you could test that longer strings are never contained in shorter strings. Or anything else interesting you can come up with.