M21274 – MATHFUN
Discrete Mathematics and Functional Programming
Worksheet 6: Higher-Order Functions

Introduction

This worksheet aims to give you practice in writing functions that take other functions as arguments. Use of higher-order functions `map`, `filter` and `foldr` is emphasized, as is the use of function composition.

Begin by downloading the `Week6.hs` file. The `Week6` module includes some functions from the lecture. Experiment with these definitions and study the worked examples before you attempt the exercises.

Worked example 1

Question: Write a function `alwaysEven` with the signature below:

```
alwaysEven :: (Int -> Int) -> [Int] -> Bool
```

The first parameter of `alwaysEven` is a function `f` (with signature `Int -> Int`) and the second parameter is a list `xs` of `Int`s. The function should apply `f` to every element of `xs` and return `True` if this operation only produces even numbers.

Test	Output
`alwaysEven double [1, 2, 3]`	`True`
`alwaysEven (multiply 3) [1, 2, 3]`	`False`
`alwaysEven (+3) [7, 9, 11]`	`True`

Solution 1: There are three higher-order functions that we can use:

- `map`: Apply a function to all elements
- `filter`: Keep only elements that pass a check
- `foldr`: Fold a list down to a single value

Clearly, we need to use `map` to apply function `f` to all elements of list `xs`:

```
map f xs
```

The above expression returns a list. For example, the expression:

```
map (+3) [7, 8, 9, 10]
```

evaluates to `[10,11,12,13]`.

Next, we want to keep all the even numbers from the list that the `map` produces. To do this we use `filter` as well as the Prelude's `even` function which takes an integer and returns true if it is even and false otherwise:

```
filter even (map f xs)
```

We first apply the function `f` to all elements using `map`, then filter (i.e. keep) the even ones. Test this by evaluating the following (it should return `[10,12]`).

```
filter even (map (+3) [7, 8, 9, 10])
```

We point out that you could also use lambda expressions instead of `even`; a way to filter the even numbers of a list using a lambda expression is:

```
filter (\y -> mod y 2 == 0)
```

After filtering the list, we can just check if the length of the resulting list matches the length of `xs`.

Here is our first definition for `alwaysEven`. Add it at the end of `Week6.hs`:

```
alwaysEven :: (Int -> Int) -> [Int] -> Bool
alwaysEven f xs = length (filter even (map f xs)) == length xs
```

Check it by evaluating the following expression (as well as the test cases from table 1):

```
alwaysEven (+3) [7, 8, 9, 10]
```

Solution 2 (better): Here's a new definition for the `andAll` function from last week that uses `foldr`. We fold the list `xs`, from right to left (hence `foldr`), using the `&&` operator. The parameter `True` specifies what `andAll` should return given an empty list:

```
andAll :: [Bool] -> Bool
andAll xs = foldr (&&) True xs
```

You could use the `andAll` function to write a shorter definition for `alwaysEven`:

```
alwaysEven f xs = andAll (map (even . f) xs)
```

In this solution we apply `f` composed with `even` to all elements of the list. This returns a list of Booleans. For example the following expression

```
map (even . (+3)) [7, 8, 9, 10]
```

evaluates to `[True,False,True,False]`.

Solution 3 (best): Our final definition is brief as it does not include the parameter `xs`:

```
alwaysEven f = andAll . map (even . f)
```

It is basically saying: "`alwaysEven f`" is the same as "`andAll . map (even . f)`".

Worked example 2

Question: Write a function with the signature below that takes a function and a list. It should only apply the function to the positive numbers in the list; non-positive numbers should remain unchanged.

```
updatePositivesOnly :: (Float -> Float) -> [Float] -> [Float]
```

Test	Output
`updatePositivesOnly sqrt [-1.0, 0.0, 1.0, 2.0]`	`[-1.0, 0.0, 1.0, 1.414]`

Solution 1: Let's start with a recursive definition. For its base case, `updatePositivesOnly` returns an empty list if it is given an empty list (it doesn't matter what function we have):

```
updatePositivesOnly :: (Float -> Float) -> [Float] -> [Float]
updatePositivesOnly _ [] = []
```

In the recursive case, we take an element `x` from the list, and see if it is positive. If it is, then we apply the function `f` to `x` and prepend the resulting value to the result of processing the tail of the list; otherwise we prepend `x` unchanged:

```
updatePositivesOnly _ [] = []
updatePositivesOnly f (x : xs)
  | x > 0 = f x : updatePositivesOnly f xs
  | otherwise = x : updatePositivesOnly f xs
```

Solution 2 (better): You could also use a lambda expression to define this function in 1 line:

```
updatePositivesOnly f = map (\x -> if x > 0 then f x else x)
```

Programming exercises

Use of map, filter and foldr

Use the higher-order functions `map`,`filter` and `foldr` to write the following functions. You should aim to make your solutions as compact as possible. Make use of function composition where appropriate.

1. Write a function `mult10` that multiplies every element of a list by 10.

```
mult10 :: [Int] -> [Int]
```

Test	Output
`mult10 [5, 7, 2, 4]`	`[50, 70, 20, 40]`

2. Write a function `onlyLowerCase` that uses the `isLower` function to remove any character from a string that is not a lowercase letter.

```
onlyLowerCase :: String -> String
```

Test	Output
`onlyLowerCase "Port 15"`	`"ort"`

3. Write a one-line definition for the `orAll` function from last week. See the definition for `andAll` in the first worked example.

Test	Output
`orAll [True,False,True]`	`True`
`orAll [False,False,False]`	`False`

4. Write a one-line definition for `sumSquares` (from worksheet 3).

Test	Output
`sumSquares [3, 2, 4]`	`29`

5. Write a function `zeroToTen` that takes a list of integers and only keeps those that are between 0 and 10 (inclusive).

Test	Output
`zeroToTen [7, -3, 0, 15, 10, 2]`	`[7, 0, 10, 2]`

6. Write a function `squareRoots` with the signature below. `squareRoots` should return a list of the square roots of all the non-negative values in the given list.

```
squareRoots :: [Float] -> [Float]
```

Test	Output
`squareRoots [4, -8, 10, 0]`	`[2.0, 3.16, 0.0]`

7. Write a function `countBetween` with the signature below. The function counts the number of items in a list that are between specified lower- and upper–bounds values (those equal to either bound should be counted).

```
countBetween :: Float -> Float -> [Float] -> Int
```

Test	Output
`countBetween 3 6 [5, 9, 2, 4, 6, 3, 1, 4]`	`5`

8. Write a function `alwaysPositive` that tests whether applying a given function to all the elements of a list results only in positive values.

```
alwaysPositive :: (Float -> Float) -> [Float] -> Bool
```

Test	Output
`alwaysPositive (+10) [-8, 2]`	`True`
`alwaysPositive (*2) [-1, -8, 2]`	`False`

   Write **three definitions of this function** similar to the first worked example.

9. Write a function `productSquareRoots` that returns the product of the square roots of all non-negative numbers in the given list:

```
productSquareRoots :: [Float] -> Float
```

Test	Output
`productSquareRoots [4, -8, 10]`	`6.3245554`

Other higher-order functions

Complete the following questions **without** using `map`, `filter` or `foldr`.

10. Write a recursive polymorphic function `removeFirst` that removes the first element of a list that has a certain property.

```
removeFirst :: (a -> Bool) -> [a] -> [a]
```

Test	Output
`removeFirst (<0) [3, -1, 4, -8, 2]`	`[3, 4, -8, 2]`
`removeFirst even [13, 14, 15, 16]`	`[13, 15, 16]`

11. Using your`removeFirst` function and and other function from the Prelude that you find useful, write a one-line `removeLast` function that removes the last element of a list that has a certain property.

Test	Output
`removeLast (<0) [3, -1, 4, -8, 2]`	`[3, -1, 4, 2]`
`removeLast even [13, 14, 15, 16]`	`[13, 14, 15]`

Using lambda expressions

Lambda expressions are nameless (i.e. anonymous) functions. These are similar to arrow functions in Dart and JavaScript. The term lambda expression comes from lambda calculus, which underlies all functional programming languages. Lambda expressions are often used as arguments to higher-order functions such as `map`, `filter` and folds.

To define a lambda expression, we use a backslash (intended to look like the letter lambda - λ), function arguments, an arrow and a return expression. For example:

Lambda expression (anonymous)	Standard function (named)
`\x -> 2 * x`	`mult2 x = x * 2`
`\x y -> x * y`	`multiply x y = x * y`
`\x -> x < 0 && even x`	`evenNegative x = x < 0 && even x`

We can apply a lambda expression like this (the following evaluates to 8):

```
(\x -> 2 * x) 4
```

12. Using `filter` and a lambda expression, write an alternative solution to exercise 5.

13. [harder] We can rewrite pretty much any of your functions using `foldr` (no need for a `map` or `filter`). Here is a definition of `mult10` using `foldr`:

    ```
    mult10 :: [Int] -> [Int]
    mult10 = foldr (\x xs -> x * 10 : xs) []
    ```

    Here is another definition for `onlyLowerCase`:

    ```
    onlyLowerCase :: String -> String
    onlyLowerCase = foldr (\x xs -> if isLower x then x : xs else xs) []
    ```

    Write similar definitions for the following functions:

    (a) `alwaysPositive`

    (b) `productSquareRoots`

    (c) `reverse` (a Prelude function that reverses the order of a list)