COMP105 Lecture 14

Map and Filter

Outline

Today

- Higher order programming
- Map
- Filter

Relevant book chapters

- Programming In Haskell Chapter 7
- Learn You a Haskell Chapter 6

Recap: transforming lists

```
double_list [] = []
double_list (x:xs) = 2 * x : double_list xs

ghci> double_list [1..5]
[2,4,6,8,10]
```

```
square_list [] = []
square_list (x:xs) = x * x : square_list xs

ghci> square_list [1..5]
[1,4,9,16,25]
```

Map

Map applies a function `f` to every element in a list

```
map' :: (a -> b) -> [a] -> [b]
map' _ []     = []
map' f (x:xs) = f x : map' f xs
```

```
ghci> map even [1..5]
[False,True,False,True,False]
```

Map examples

```
square x = x * x
```

```
ghci> map square [1..5]
[1,4,9,16,25]
```

```
ghci> map reverse ["the", "quick", "brown", "fox"]
["eht","kciuq","nworb","xof"]
```

```
ghci> map fst [(1,2),(3,5),(6,3),(2,6),(2,5)]
[1,3,6,2,2]
```

Currying and map

It is common to use **curried functions** with map

```
ghci> map (*2) [1..5]
[2,4,6,8,10]
```

```
ghci> map (2^) [1..5]
[2,4,8,16,32]
```

```
ghci> map (drop 2) ["the", "quick", "brown"]
["e","ick","own"]
```

Anonymous functions and map

It is common to use an **anonymous function** with map

```
ghci> map (\x -> x*x) [1..5]
[1,4,9,16,25]
```

```
ghci> map (\(x, y) -> x + y) [(1,1), (2,2), (3,3)]
[2,4,6]
```

```
ghci> map (\(_:y:_) -> y) ["the", "quick", "brown"]
"hur"
```

Nested maps

When working with nested lists, it is common to use **nested maps**

```
ghci> map (map (*2)) [[1,2,3], [4,5,6], [7,8]]
[[2,4,6],[8,10,12],[14,16]]
```

```
import Data.Char

ghci> map (map toUpper) ["the", "quick", "brown"]
["THE","QUICK","BROWN"]
```

Note the use of currying for the inner map

Exercise

What do these functions do?

```
mystery list = map (`mod` 2) list
```

```
mystery2 list = sum (map (\_ -> 1) list)
```

```
mystery3 list = map (\x -> ['a'..'z'] !! x) list
```

Recap: dropping elements

We know how to use recursion to **drop some** elements of a list

```
drop_odds [] = []
drop_odds (x:xs)
    | even x    = x : rest
    | otherwise = rest
    where rest = drop_odds xs
```

```
ghci> drop_odds [1..10]
[2,4,6,8,10]
```

Filter

Filter keeps only the elements for which `f` returns `True`

```
filter' :: (a -> Bool) -> [a] -> [a]
filter' _ [] = []
filter' f (x:xs)
    | f x       = x : rest
    | otherwise = rest
    where rest = filter' f xs
```

```
ghci> filter' even [1..10]
[2,4,6,8,10]
```

Filter examples

```
ghci> filter (>=10) [1..12]
[10,11,12]
```

```
ghci> filter (\x -> length x <= 2) ["aaa", "bb", "c"]
["bb","c"]
```

```
ghci> filter (\x -> x `elem` "aeiou") "the quick brown"
"euio"
```

Combining map and filter

```
square_even :: [Int] -> [Int]
square_even list = map (^2) (filter even list)
```

```
ghci> square_even [1..10]
[4,16,36,64,100]
```

```
squares_gt100 :: [Int] -> [Int]
squares_gt100 list = filter (>100) (map (^2) list)
```

```
ghci> squares_gt100 [1..15]
[121,144,169,196,225]
```

Exercise

What do these functions do?

```
mystery4 list = filter even list ++ filter odd list
```

```
mystery5 list = map (\(x, y) -> x * y) (zip list [0..])
```

```
mystery6 list1 list2 =
    let
        filtered = filter (< length list2) list1
    in
        map (\x -> list2 !! x) filtered
```

Higher order programming

`map` and `filter` are examples of **higher order** programming

This style

- de-emphasises recursion
- focuses on applying functions to lists
- is available in imperative languages (python, C++)

There is a whole **family** of higher order programming functions available in Haskell

Exercises

1. Use `map` to write a function `cube_list` that cubes every element of a list.

2. Use `map` to write a function `div2_list` that takes a list of floats, and divides each element of the input list by 2

3. Use `filter` to write a function `only_div3` that takes a list of integers, and returns only the integers divisible by 3

4. Use `filter` to write a function `only_lower_case` that takes a string, and returns only the lower case letters.

Exercises

5. Use `filter` to write a function `no_letters` that takes a string, and returns the characters that are *not* upper or lower case letters.

6. Use `map` and `filter` to write a function `cubes_lt1000` that takes a list, and returns the cubes of the elements in the list that are less than 1000.

7. Write a function `sum_gt100` that takes an input of type `[[Int]]` and returns all sub-lists whose sum is greater than 100.

Summary

- Higher order programming
- Map
- Filter

Next time: `fold`