COMP105 Lecture 15

Filter

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]
```

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

Use `filter` to implement the following functions

1. Write a function `onlyDiv3` that takes a list of numbers and returns a list containing all of the numbers in that list that are divisible by three

2. Write a function `onlyLower` that takes a string and returns a string containing all of the lower case letters in the string

3. Write a function that takes a list of lists of integers, and returns the same list, but with all even numbers removed (hint: you will need to use `map` as well)