COMP105 Lecture 8

List Recursion Examples

Take

```
take' 0 list   = []
take' n []     = []
take' n (x:xs) = x : take' (n-1) xs
```

```
ghci> take' 2 [1,2,3,4]
[1,2]
```

Here we have two base cases

- We either take as many as we want
- Or we run out of things to take

Drop

```haskell
drop' 0 list   = list
drop' n []     = [] -- could also be an error
drop' n (x:xs) = drop' (n-1) xs
```

The only differences?

- We don't prepend the head in the recursive rule
- The base case now returns the rest of the list

Implementing elem

Recall that `elem e list` checks whether `e` is an element of `list`

```
elem' e [] = False
elem' e (x:xs)
     | e == x    = True
     | otherwise = elem' e xs
```

The maximum element of a list

```
maximum' []     = error "Called with empty list"
maximum' [x]    = x
maximum' (x:xs) =
    let
        max_tail = maximum' xs
    in
        if (x > max_tail) then x else max_tail
```

Note that the function will break if you give it the empty list

Reversing a list

```
reverse' []     = []
reverse' (x:xs) = reverse' xs ++ [x]
```

```
reverse' [1,2,3]
→ reverse' [2,3] ++ [1]
→ reverse' [3] ++ [2] ++ [1]
→ reverse [] ++ [3] ++ [2] ++ [1]
→ [] ++ [3] ++ [2] ++ [1]
→ [3,2,1]
```

Consuming more than one element

Some recursive functions will use more than just the head of the list

```
add_adjacent []       = []
add_adjacent [x]      = error "Odd number of elements"
add_adjacent (x:y:xs) = x+y : add_adjacent xs
```

```
ghci> add_adjacent [1,2,3,4,5,6]
[3,7,11]
```

Consuming more than one element

You can use the next element of the list without consuming it

```
add_next []         = error "Not enough elements"
add_next [_]        = error "Not enough elements"
add_next [x,y]      = [x+y]
add_next (x:y:xs) =  x+y : add_next (y:xs)
```

```
ghci> add_next [1,2,3,4,5]
[3,5,7,9]
```

Consuming more than one element

You can also break down the list in more complex ways

```
group n []   = []
group n list =
    let
        first = take n list
        rest  = drop n list
    in
        first : group n rest
```

```
ghci> group 4 [1..12]
[[1,2,3,4],[5,6,7,8],[9,10,11,12]]
```

Exercises

1. Use recursion to implement a function `containsThree` that takes a list of integers and returns `True` if the list contains `3` and `False` otherwise

2. Write a function `sumEvenIdxs` that takes a list and returns the sum of the elements that even indexes, so `sumEvenIdxs [1,2,3,4,5] = 1 + 3 + 5` (remember that Haskell lists are zero-indexed)