# SET07106 - Exercise sheet 3
# Recursion I and Sets

**NOTE:** This practical introduces some new Haskell syntax, but in a very focussed way. It should be read in conjunction with one of the Haskell textbooks found on Moodle, or with this online version of Learn You a Haskell, in particular the chapter *Hello Recursion!*.

The type signatures of all of the functions you will need to define already exist inside the file `exercise3.hs`. You just need to fill in the gaps then uncomment the function. The gaps for some functions are larger than for others.

## Recursion

Remember that recursion is a way of defining functions when the structure of your inputs allows it. By that, we mean there are a number of base cases, and a number of inductive cases, for how elements of that structure look. For example, the natural numbers ($\mathbb{N}$) are defined as:

- $0 \in \mathbb{N}$ (i.e. 0 is a natural number), and
- $n \in \mathbb{N} \rightarrow (n + 1) \in \mathbb{N}$ (i.e. if $n$ is a natural number, then $n + 1$ is a natural number).

Any function that we define over natural numbers (for us, `Int`, since we will generally not care about the distinction) could usually be defined by induction. We have to cover the two cases: what happens to 0, and what happens to $n$ *in terms of* what happens to $n - 1$.

The structure we will most often use is the list. Lists are similarly recursively defined. Lists are either empty, or an element added to the front of a list:

- `[]` is a list, and
- if `x` is an element of type `a`, and `xs` is a list of elements of type `a`, then `x:xs` is a list of elements of type `a`.

As an example, consider adding up all the whole numbers up to a certain number. So `sumUpTo 3 = 0 + 1 + 2 + 3`, for example. We could define this function using induction:

```
sumUpTo ::  Int -> Int
sumUpTo 0 = 0
sumUpTo n = n + sumUpTo (n-1)
```

Note we have given two separate cases: one for the base case, and one for the inductive case. How do we know what the inductive case's definition is? Consider the following:

$$0 + 1 + \ldots + (n-1) + n = n + 0 + 1 + \ldots + (n-1) = n + (0 + 1 + \ldots + (n-1))$$

The left-hand side is `sumUpTo n`, and the right-hand side is `n + sumUpTo (n-1)`, which is what our inductive case is. In a similar vein, the factorial of $n$ (we will see this in later weeks and denote it as $n!$) is the product of all the numbers up to $n$, with the convention that $0! = 1$. Inductively, it could be defined as:

```
fac ::  Int -> Int
fac 0 = 1
fac n = n * fac (n-1)
```

We can also define `map` and `filter` inductively on lists:

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

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

What does each say? The inductive case of `map` says that we apply `f` to `x`, then stick it on the front of the list obtained by applying `map` to the smaller list `xs`. The inductive case of `filter` is slightly more complex, having a set of guards. The first says that, if `f x` evaluates to `True`, then keep `x` and stick it on the front of the list obtained from filtering `xs`, and the other case says throw `x` away.

## Paper-based exercises

In order to get to grips with recursion, it will be useful for you to assume the role of the Haskell interpreter. To do this, you should calculate the following things on paper, by writing out each line individually.

**Exercise 1** *`map (*2) [3,5,0,1]`*

**Exercise 2** *`fac 3`*

**Exercise 3** *`filter (<5) [6,2,5,1,6,3,8]`*

Whenever you come across a new recursive function, it is useful to write out by hand what the function is doing, and this will help you clarify how the function works. Similarly, when trying to debug a recursive function you have written, write out by hand some small example and see if you can see what has gone wrong.

### Recursion on lists

Attempt the following exercises about lists using recursion. You can leave out the type signature if you want. Note it is also possible to do these exercises using `map` and `filter`, since, as we have seen above, those functions just hide the recursion away. The base case will always be `[]`, and the inductive case will always be `x:xs`.

**Exercise 4** *Write a function called* `myLength` *which works out how many elements are in a list. The empty list has no elements.*

**Exercise 5** *Write a function called* `mySum` *which works out the sum of a list of integers. (Note:* `mySum [1..n] == sumUpTo n` *if you have done it correctly.)*

**Exercise 6** *Write a function called* `myProduct` *which works out the product of a list of integers. (Remember, product means multiplication. If it works correctly, you should find that* `myProduct [1..n] = fac n`*.)*

**Exercise 7** *Write an function called* `flipSign`*. This function should take a list of integers, and flip the sign of each of them (i.e. positive numbers become negative, and* vice versa*, whilst* 0 *stays as* 0*).*

**Exercise 8** *Write a function called* `addAtEnd`*. This function takes an element of type* `a`*, a list of type* `a`*, and returns a list of type* `a`*. It takes the element given and adds it at the end of the given list. For example,* `addAtEnd 0 [1..5] = [1,2,3,4,5,0]`*. You might find it helpful to know that the list containing one object* `x` *is written as* `[x]`*.*

**Exercise 9** *(Harder) Write a function called* `append` *which takes two lists, and adds the first list to the end of the second. You might like to use your function* `addAtEnd`*. For example,* `append [1..5] [6..10] = [6,7,8,9,10,1,2,3,4,5]`*.*

**Exercise 10** *(Harder) Write a function called* `myReverse` *which takes a list and reverses it. You might like to use your function* `addAtEnd`*.*

## Sets

We know from the lectures that sets contain no duplicates, and are unordered. So, we have:

$$\{1,2,3,2,1\} = \{1,2,3\} = \{3,2,1\} = \{2,1,3\} = \{3,3,3,3,3,1,1,1,1,2,2,2,2,3\}$$

Haskell's lists, on the other hand, can have duplicates and are ordered. So:

```
ex3> [1,2] == [2,1]
False
```

Removing duplication is straightforward. There is a function called `elem` which checks whether or not a given value is in a given list:

```
elem ::  (Eq a) => a -> [a] -> Bool
elem _ [] = False
elem x (y:ys)
  | x == y = True
  | otherwise = elem x ys
```

Do not worry about the underscore in `elem _ []`. It is called a wildcard and we will investigate them later in the semester. Note the syntax: there are *guards* that tell us what to do depending on a Boolean. So, if `x==y`, we do one thing, and `otherwise` we do something else. The reserved word `otherwise` is used to catch all remaining cases. When there are only two cases, like this, we could use `if .. then .. else` as well.

**Exercise 11** *Write a function which takes a list and returns the same list where all elements are unique. In other words, at each stage the function should check whether the first element is in the remaining list (use the function* `elem`*) and only keep it if it does* ***not*** *appear. The type signature you should use is:*

```
uniqueList ::  (Eq a) => [a] -> [a]
```

Of course, the values returned by `uniqueList` will still be ordered:

```
ex3> uniqueList [1,2,3,4,2,1,3,2,3,2,1]
[1,2,3,4]
ex3> uniqueList [4,3,2,1,2,1,3,2,3,2,1]
[4,3,2,1]
```

So, we cannot check for set equality simply by equating two duplicate-free lists. There are a few ways to determine set equality, of which we will focus on using the `subset` function, matching the definition given in the lectures:

$$A = B \text{ iff } A \subseteq B \wedge B \subseteq A$$

Clearly, we just need to define the subset function, that takes two lists (they can have duplicate entries) and works out if all the elements from the first appear somewhere in the second.

**Exercise 12** *Write a function* `subset` *which takes two lists and determines whether the elements of the first appear in the second. Remember that the empty set, $\emptyset$, is a subset of anything. The type signature you should use is:*

```
subset ::  (Eq a) => [a] -> [a] -> Bool
```

You should now be able to define set equality as follows:

```
setEquality ::  (Eq a) => [a] -> [a] -> Bool
setEquality xs ys = subset xs ys && subset ys xs
```

## Set operations

We saw three set operations in the lecture: intersection $\cap$, union $\cup$, and complement $\cdot^C$. Union is very simple to implement, since all we do is add two lists together and then remove any duplication. The first of those is an inbuilt function in `Haskell`:

```
union ::  (Eq a) => [a] -> [a] -> [a]
union xs ys = uniqueList (xs ++ ys)
```

The double-plus `++` is the inbuilt version of `append` that you created earlier.

Intersection is a bit more complicated. Recall that the intersection of anything with the empty set is the empty set. Next, if a set is non-empty, we take the elements common to both. We can either use list comprehension or induction. The latter is an exercise for you. Here is the former:

```
intersection ::  (Eq a) => [a] -> [a] -> [a]
intersection xs ys = uniqueList [z | z <- xs, elem z ys]
```

**Exercise 13** *Write a function for set intersection, called* `myIntersection'`*. Do not worry about removing duplication: that is handled in the function* `myIntersection`*, already included in your Haskell file.*

Complement is more complicated again. That will form part of your coursework.