The general recipe for writing recursive functions

We have seen in the previous lesson that one may follow these guidelines in order to write a recursive function in Haskell:

1. Identify the end goal(s) of the problem.
2. Determine what happens when a goal is reached.
3. List all alternate possibilities.
4. Determine your "rinse and repeat" process.
5. Ensure that each alternative brings you closer to the goal.

We carry on our exploration and explain how to use pattern-matching in order to resolve problems recursively in an elegant and easy way.
Using the `map` function

The `map` function plays a central role in functional programming.

The `map` function is a higher-order function which takes two arguments:

1. a function
2. a list

and applies the function to every element of the list:

```ghci
ghci> map reverse ["dog","cat","moose"]
["god","tac","esoom"]
ghci> map head ["dog","cat","moose"]
"dcm"
ghci> map (take 4) ["pumpkin","pie","peanut butter"]
["pump","pi","pean"]
```
The **myMap** function

The **myMap** function is implemented by pattern matching in this way:

\[
\begin{align*}
\text{myMap } f \ [\ ] &= [\ ] \\
\text{myMap } f \ (x:xs) &= (f \ x) : \text{myMap } f \ xs
\end{align*}
\]

**Exercise:** Explain how the **myMap** function works.
Filtering a list – using the `filter` function

The higher-order `filter` function takes

(1) a test function which returns `True` or `False`  
(2) a list

and keeps only elements of the list which pass the test:

```
ghci> filter even [1,2,3,4]  
[2,4]  
ghci> filter (\(x:xs) \rightarrow x == 'a') ['apple','banana','avocado']  
['apple','avocado']
```
The myFilter function

The filter function is implemented by pattern matching in this way:

```
myFilter test [] = []
myFilter test (x:xs) = if test x
                          then x:myFilter test xs
                          else myFilter test xs
```

**Exercise:** Explain how the myFilter function works.

**Exercise:** Implement the remove function which takes a test function and a list and removes the elements which pass the test.
Folding a list on the left – using the foldl function

The higher-order function `foldl` takes three arguments:
(1) a binary function (2) an initial value (3) a list
and reduces the list to a single value using the binary function and the initial value.

The `foldl` function is typically used to sum a list:

```
ghci> foldl (+) 0 [1,2,3,4]
10
```

in the following way:
Visualising how the \texttt{foldl} function works

\begin{itemize}
  \item \texttt{foldl (+) 0 [1,2,3,4]}
  \item \quad 0 + 1 = 1
  \item \texttt{foldl (+) 1 [2,3,4]}
  \item \quad 1 + 2 = 3
  \item \texttt{foldl (+) 3 [3,4]}
  \item \quad 3 + 3 = 6
  \item \texttt{foldl (+) 6 [4]}
  \item \quad 6 + 4 = 10
  \item \texttt{foldl (+) 10 [] = 10}
\end{itemize}
Folding a list from the left – using the **foldl** function

The **foldl** function can also be used to reverse a list:

```hs
rcons x y = y:x
myReverse xs = foldl rcons [] xs
```

Note that the **myReverse** function takes a list as argument, and returns a list:

```
ghci> myReverse [1,2,3]
[3,2,1]
```

Can you explain how the **foldl** function works in this case?
Visualising how the `foldl` function works

```
foldl rcons [] [1,2,3]
```

```
foldl rcons [1] [2,3]
```

```
foldl rcons [2,1] [3]
```

```
foldl rcons [3,2,1] [] = [3,2,1]
```
The `myFoldl` function

The `myFoldl` function is implemented by pattern matching in this way:

\[
\begin{align*}
\text{myFoldl } f \text{ init } [] &= \text{ init} \\
\text{myFoldl } f \text{ init } (x:xs) &= \text{myFoldl } f \text{ newInit } xs \\
&\text{ where newInit } = f \text{ init } x
\end{align*}
\]

**Exercise:** Explain how the `myFoldl` function works.
Folding a list on the right – using the `foldr` function

The higher-order function `foldr` takes three arguments:

1. a binary function
2. an initial value
3. a list

and reduces the list to a single value using the binary function and the initial value.

The `foldr` function may be used for instance to sum a list:

```
ghci> foldr (+) 0 [1,2,3,4]
10
```

in the following way:

```
1 2 3 4
```

- the elements of the list `[1,2,3,4]` taken as argument by the function `foldr`
- the binary function `(+)` taken as argument by the function `foldr`
- the initial value `0` taken as argument by the function `foldr`
The **myFoldr** function

The **myFoldr** function is implemented by pattern-matching in this way:

\[
\text{myFoldr } f \text{ init } [] = \text{init} \\
\text{myFoldr } f \text{ init } (x:xs) = f \ x \ \text{rightResult} \\
\text{where rightResult } = \text{myFoldr } f \text{ init } \ xs
\]

**Exercise:** Explain how the **myFoldr** function works.
Folding a list on the left and on the right

Note that the `foldl` and `foldr` functions generally compute different values:

```
ghci> foldl (-) 0 [1,2,3,4]
-10
ghci> foldr (-) 0 [1,2,3,4]
-2
```

= -10

= -2
Thank you for your attention!