

# Higher-order functions

- In sequential programming there is a clear distinction between functions and ordinary values
- In functional programming this is not the case, and we can treat functions as any other expressions, in particular we can use them as *parameters* and *arguments*
- A function is called *higher-order* if it takes a function as an argument or returns a function as a result, or both.

# Higher-order functions

We already saw two examples of higher-order functions

- `takeWhile :: (a -> Bool) -> [a] -> [a]` selects elements from a list so long as its first argument (a function) is true
  - `takeWhile isAlpha "abc def"`  
`⇒ "abc"`
  - `isAlpha` is a function which checks if a character is a letter
- `dropWhile :: (a -> Bool) -> [a] -> [a]` removes elements from a list so long as its first argument (a function) is true
  - `dropWhile isSpace " abc def"`  
`⇒ "abc"`
  - `isSpace` is a function which checks if a character is a space

# Mapping

- We often want to transform each element of a list in some way
- For example: double every element of a list of integers  
`double :: [Int] -> [Int]`
- To do this, we can use either *recursion* or *list comprehension*
  - `double [] = []`  
`double (x:xs) = x*2 : double xs`
  - `double xs = [ x*2 | x <- xs]`
- Another example: add 1 to every element of a list of integers  
`increment :: [Int] -> [Int]`
  - `increment [] = []`  
`increment (x:xs) = x+1 : increment xs`
  - `increment xs = [ x+1 | x <- xs]`

- In both types of definition (recursive and list comprehension) a specific operation (for example multiplying by two for `double`) is applied to each element of the list (expression `x*2` for `double`)
  - `double [] = []`  
`double (x:xs) = x*2 : double xs`
  - `double xs = [ x*2 | x <- xs]`
- Moreover, if we consider the definitions of `double` and `increment` (for example by means of list comprehension), the form of the definition is exactly the same; the only difference is the way in which the elements are transformed (`x*2` vs. `x+1`)
  - `double xs = [ x*2 | x <- xs]`
  - `increment xs = [ x+1 | x <- xs]`
- So we can modify one definition replacing `(*2)` by `(+1)` to give the other

# Mapping

- Obviously any other function could be used in place of `(*2)` or `(+1)`
- For example: function `ord` that transforms a `Char` into an `Int` which is its ASCII code

- We have:

```
double xs = [ x*2 | x <- xs]
increment xs = [ x+1 | x <- xs]
convertChar xs = [ ord x | x <- xs]
```

- Taking this approach would mean that we would write a lot of definitions which differ only in the function used to make the transformation
- Why not make those functions into *arguments* of a (higher-order) function representing the form of the definition?

# Mapping

- We can write a *single* definition in which the transformation (function) becomes a parameter of the definition
- Such a *mapping* function will take two arguments:
  - a function to transform the elements
  - a list of elements to be transformed
- This is exactly the function `map :: (a -> b) -> [a] -> [b]` defined in the prelude:  
`map f xs = [ f x | x <- xs ]`
- Or equivalently:  
`map f [] = []`  
`map f (x:xs) = f x : map f xs`
- Notice that `map` is *polymorphic*

`map :: (a -> b) -> [a] -> [b]`

		input function			input list		output list	
<code>map</code>	<code>::</code>	<code>( a</code>	<code>-&gt;</code>	<code>b )</code>	<code>-&gt;</code>	<code>[a]</code>	<code>-&gt;</code>	<code>[b]</code>
<code>map</code>	<code>::</code>	<code>(...</code>	<code>-&gt;</code>	<code>...) -&gt;</code>	<code>[...]</code>	<code>-&gt;</code>	<code>[...]</code>	

- `a` and `b` are type variables, standing for arbitrary types
- The input list must have elements to which the function can be applied
- The output list is made up of elements from the output type of the function
- Instances of the type of `map` include:
  - `map :: (Int -> Int) -> [Int] -> [Int]`  
as in the definition of `double`
  - `map :: (Char -> Int) -> [Char] -> [Int]`  
as in the definition of `convertChar`

- The function to double every element of a list can now be given by applying `map` to two things: the transformation (function) `(*2)` and the list in question:

```
double xs = map (*2) xs
```

- `increment :: [Int] -> [Int]` and `convertChar :: [Char] -> [Int]` can be rewritten as:

```
increment xs = map (+1) xs
```

```
convertChar xs = map ord xs
```

# Example

- `increment :: [Int] -> [Int]`  
`increment xs = map (+1) xs`
- `map :: (a -> b) -> [a] -> [b]`  
`map f [] = []`  
`map f (x:xs) = f x : map f xs`
- `increment [1,2,3]`

# Example

```
increment [1,2,3]
⇒ map (+1) [1,2,3]
⇒ (1+1) : [2,3] (cf. map (+1) [1,2,3] = 1+1 : map (+1) [2,3])
⇒ (1+1) : map (+1) [2,3]
⇒ (1+1) : (2+1) : map (+1) [3]
⇒ (1+1) : (2+1) : (3+1) : map []
⇒ (1+1) : (2+1) : (3+1) : []
⇒ [2,3,4]
```

# What are the advantages of higher-order functions?

- The `map` function is a good example of *abstraction* and *generalisation*
- If you understand *what* `map` does, definitions that use it are easier to understand and modify
- `map` can be reused for all functions of this form

# Filtering

- We often want to produce a sub-list by selecting the elements of some other list that all have some property in common
- For example: `isAlpha :: Char -> Bool` decides whether a character is a letter or not giving a boolean result (`True` or `False`)
- This is the way a *property* over any type `t` is modelled:  
`t -> Bool`
- An element `x` has the property precisely when `f x` has the value `True`
- A filter function will thus take a property and a list as inputs and return a list containing those elements that have the property

- To filter out the letters in a string for example we can write:

```
getLetters :: String -> String
getLetters [] = []
getLetters (x:xs)
    | isAlpha x = x : getLetters xs
    | otherwise = getLetters xs
```

- If we want only the digits, the function looks the same, except that `isDigit` replaces `isAlpha`:

```
getDigits :: String -> String
getDigits [] = []
getDigits (x:xs)
    | isDigit x = x : getDigits xs
    | otherwise = getDigits xs
```

- If the guard condition `p x` is `True` than the element `x` is the first element of the result; the remainder of the result is given by selecting those elements in `xs` which have the property `p`

- This common form is known as filtering and there is a standard higher-order function `filter` that takes as input a predicate

`p :: a -> Bool` and a list:

```
filter :: (a -> Bool) -> [a] -> [a]
```

```
filter p [] = []
```

```
filter p (x:xs)
```

```
    | p x = x : filter p xs
```

```
    | otherwise = filter p xs
```

- Now we can redefine `getLetters` easily:

```
getLetters xs = filter isAlpha xs
```

```
filter :: (a -> Bool) -> [a] -> [a]
```

		input function			input list		output list	
filter	::	( a	->	b )	->	[a]	->	[a]
filter	::	(...	->	...)	->	[...]	->	[...]

- a and b are type variables, standing for arbitrary types
- The property is a function that returns a boolean
- The output list list is made up of elements from the input list; the property works over the same type too

# Folding

- We often want to combine all the elements of a list into a single value in uniform way
- For example: the standard function `sum` adds together all the elements of an integer list
- The total is given by *folding* the operator `+` into the list:  
 $\text{sum } [1,3,71] = 1 + 3 + 71$
- The `fold` higher-order functions implement the operation of *folding* an operator or function into a list of values
- There are different versions of the `fold` functions in the prelude: `foldr1`, `foldr`, etc.

- `foldr1 :: (a -> a -> a) -> [a] -> a` folds a function into a non-empty list
- `foldr1` takes two arguments:
  - the first is a binary function over the type `a`; for example, the function `+` over `Int`
  - the second is a list of elements of type `a` which are to be combined using the operator; for example, `[1,3,71]`
- The result is a single value of type `a`; in the current example we have:  
`foldr1 (+) [1,3,71] = 75`
- Other example which use `foldr1` include:  
`foldr1 (*) [1 .. 6] = 720`  
`foldr1 min [4,9,3,5] = 3`  
`foldr1 (++) [‘Hello’, ‘ ’, ‘world’] = ‘Hello world’`

- `foldr :: (a -> a -> a) -> a -> [a] -> a` works for non-empty and empty lists
- The extra argument `a` corresponds to the value to be returned on the empty list
- We can now define some of the standard functions of Haskell like `sum`, `product` and `concat` in terms of `foldr`:
  - `sum xs = foldr (+) 0 xs`
  - `product xs = foldr (*) 1 xs`
  - `concat xs = foldr (++) [] xs`

- How would you define a function `myLength` that computes the number of elements in a list using `map` and `sum`?
- Using `map`, `filter` and `sum` give the definitions of the following functions that take as input a list of integers `xs`:
  - `mySquare` returns the list consisting of the squares of the integers in `xs`
  - `mySquareEven` returns the list consisting of the squares of even integers in `xs`
  - `mySumSquareEven` returns the sum of squares of even integers in `xs`