A function is **Higher-Order** if it takes a function as an argument or returns one as its result.

Higher-order function aren’t weird; the differentiation operation from high-school calculus is higher-order:

```
deriv :: (Float->Float)->Float->Float
deriv f x = (f(x+dx) - f x)/0.0001```

Many recursive functions share a similar structure. We can capture such “recursive patterns” in a higher-order function.

We can often avoid the use of explicit recursion by using higher-order functions. This leads to functions that are shorter, and easier to read and maintain.
Currying Revisited

We have already seen a number of higher-order functions. In fact, any curried function is higher-order. Why? Well, when a curried function is applied to one of its arguments it returns a new function as the result.

 Uh, what was this currying thing?

A curried function does not have to be applied to all its arguments at once. We can supply some of the arguments, thereby creating a new specialized function. This function can, for example, be passed as argument to a higher-order function.
Currying Revisited...

How is a curried function defined?

A curried function of \( n \) arguments (of types \( t_1, t_2, \ldots, t_n \)) that returns a value of type \( t \) is defined like this:

\[
\text{fun} :: t_1 \to t_2 \to \cdots \to t_n \to t
\]

This is sort of like defining \( n \) different functions (one for each \( \to \)). In fact, we could define these functions explicitly, but that would be tedious:

\[
\begin{align*}
\text{fun}_1 & :: t_2 \to \cdots \to t_n \to t \\
\text{fun}_1 \ a_2 \cdots a_n & = \cdots \\
\text{fun}_2 & :: t_3 \to \cdots \to t_n \to t \\
\text{fun}_2 \ a_3 \cdots a_n & = \cdots
\end{align*}
\]
Duh, how about an example?

Certainly. Let's define a recursive function \( \text{get\_nth} \ n \ x\!s \) which returns the \( n \):th element from the list \( x\!s \):

\[
\text{get\_nth} \ 1 \ (x:\_\_) = x \\
\text{get\_nth} \ n \ (\_\!:x\!s) = \text{get\_nth} \ (n-1) \ x\!s \\
\]

\( \text{get\_nth} \ 10 \ "\text{Bartholomew}" \Rightarrow 'e' \)

Now, let's use \( \text{get\_nth} \) to define functions \( \text{get\_second} \), \( \text{get\_third} \), \( \text{get\_fourth} \), and \( \text{get\_fifth} \), without using explicit recursion:

\[
\text{get\_second} = \text{get\_nth} \ 2 \\
\text{get\_third} = \text{get\_nth} \ 3 \\
\text{get\_fourth} = \text{get\_nth} \ 4 \\
\text{get\_fifth} = \text{get\_nth} \ 5 \\
\]

[5]
get_fifth "Bartholomew" ⇒ 'h'

map (get_nth 3)
  ["mob","sea","tar","bat"] ⇒ "bart"

So, what’s the type of get_second?

Remember the Rule of Cancellation?

The type of get_nth is \( \text{Int} \rightarrow [\text{a}] \rightarrow \text{a} \).

get_second applies get_nth to one argument. So, to get the type of get_second we need to cancel get_nth’s first type: \( \text{Int} \rightarrow [\text{a}] \rightarrow \text{a} \equiv [\text{a}] \rightarrow \text{a} \).
Patterns of Computation

Mappings

Apply a function $f$ to the elements of a list $L$ to make a new list $L'$. Example: Double the elements of an integer list.

Selections

Extract those elements from a list $L$ that satisfy a predicate $p$ into a new list $L'$. Example: Extract the even elements from an integer list.

Folds

Combine the elements of a list $L$ into a single element using a binary function $f$. Example: Sum up the elements in an integer list.
The \texttt{map} Function

\textbf{map takes two arguments, a function and a list. map creates a new list by applying the function to each element of the input list.}

\textbf{map’s first argument is a function of type} \( a \rightarrow b \). The second argument is a list of type \([a]\). The result is a list of type \([b]\).

\texttt{map :: (a -> b) -> [a] -> [b]}
\texttt{map f [ ] = [ ]}
\texttt{map f (x:xs) = f x : map f xs}

\textbf{We can check the type of an object using the :type command. Example: :type map.}
The map Function...

\[
\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]

\[
\text{map } f\ [\ ]\ =\ [\ ]
\]

\[
\text{map } f\ (x:xs)\ =\ f\ x:\ \text{map } f\ xs
\]

\[
\text{inc } x = x + 1
\]

\[
\text{map inc } [1,2,3,4] \Rightarrow [2,3,4,5]
\]
The \textit{map} Function...

\begin{align*}
\text{map} :: & \quad (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
\text{map } f \ [ \ ] & \quad = \ [ \ ] \\
\text{map } f \ (x:x:s) & \quad = \ f \ x \ : \ \text{map } f \ x:s
\end{align*}

\textbf{map } f \ [ \ ] = [ \ ] \text{ means: “The result of applying the function } f \text{ to the elements of an empty list is the empty list.”}

\textbf{map } f \ (x:x:s) = f \ x \ : \ \text{map } f \ x:s \text{ means: “applying } f \text{ to the list } (x:x:s) \text{ is the same as applying } f \text{ to } x \text{ (the first element of the list), then applying } f \text{ to the list } x:s, \text{ and then combining the results.”}
The \texttt{map} Function...

Simulation:

\begin{verbatim}
map square [5,6] ⇒
square 5 : map square [6] ⇒
25 : map square [6] ⇒
  25 : (square 6 : map square [ ]) ⇒
  25 : (36 : map square [ ]) ⇒
    25 : (36 : [ ]) ⇒
    25 : [36] ⇒
[25,36]
\end{verbatim}
The filter Function

Filter takes a predicate \( p \) and a list \( L \) as arguments. It returns a list \( L' \) consisting of those elements from \( L \) that satisfy \( p \).

The predicate \( p \) should have the type \( a \rightarrow \text{Bool} \), where \( a \) is the type of the list elements.

Examples:

\[
\text{filter even [1..10]} \Rightarrow [2,4,6,8,10] \\
\text{filter even (map square [2..5])} \Rightarrow \\
\quad \text{filter even [4,9,16,25]} \Rightarrow [4,16] \\
\text{filter gt10 [2,5,9,11,23,114]} \\
\quad \text{where gt10 x = x > 10} \Rightarrow [11,23,114]
\]
The \textbf{filter} Function...

We can define \textbf{filter} using either recursion or list comprehension.

\underline{Using recursion:}

\begin{align*}
\text{filter} :: (a -> \text{Bool}) -> [a] -> [a] \\
\text{filter} \_ [] &= [] \\
\text{filter} \ p \ (x:xs) &= \begin{cases} 
    p \ x & = x : \text{filter} \ p \ xs \\
    \text{otherwise} & = \text{filter} \ p \ xs
\end{cases}
\end{align*}

\underline{Using list comprehension:}

\begin{align*}
\text{filter} :: (a -> \text{Bool}) -> [a] -> [a] \\
\text{filter} \ p \ xs &= [x \mid x \leftarrow xs, \ p \ x]
\end{align*}
The `filter` Function...

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

filter even [1,2,3,4] ⇒ [2,4]
```
The `filter` Function...

- `doublePos` doubles the positive integers in a list.

```haskell
getEven :: [Int] -> [Int]
geven xs = filter even xs

doublePos :: [Int] -> [Int]
doublePos xs = map dbl (filter pos xs)
  where dbl x = 2 * x
    pos x = x > 0
```

**Simulations:**

```
getEven [1,2,3] ⇒ [2]

doublePos [1,2,3,4] ⇒
  map dbl (filter pos [1,2,3,4]) ⇒
  map dbl [2,4] ⇒ [4,8]
```
A common operation is to combine the elements of a list into one element. Such operations are called reductions or accumulations.

**Examples:**

\[
\text{sum } [1,2,3,4,5] \equiv \\
(1 + (2 + (3 + (4 + (5 + 0)))))) \Rightarrow 15
\]

\[
\text{concat } ["H","i","!"] \equiv \\
("H" ++ ("i" ++ ("!" ++ ""))) \Rightarrow "Hi!"
\]

Notice how similar these operations are. They both combine the elements in a list using some binary operator (+, ++), starting out with a “seed” value (0, " ").
Haskell provides a function \texttt{foldr} ("fold right") which captures this pattern of computation.

\texttt{foldr} takes three arguments: a function, a seed value, and a list.

**Examples:**

\[
\text{foldr} \ (+) \ 0 \ [1,2,3,4,5] \Rightarrow 15 \\
\text{foldr} \ (++) \ "" \ ["H","i","!"] \Rightarrow "Hi!"
\]

**foldr:**

\[
\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\
\text{foldr} \ f \ z \ [ ] \quad = \quad z \\
\text{foldr} \ f \ z \ (x:xs) \quad = \quad f \ x \ (\text{foldr} \ f \ z \ xs)
\]
Note how the fold process is started by combining the last element $x_n$ with $z$. Hence the name seed.

$$\text{foldr}(\oplus) z[x_1 \cdots x_n] = (x_1 \oplus (x_2 \oplus (\cdots (x_n \oplus z)))))$$

Several functions in the standard prelude are defined using \texttt{foldr}:

\begin{verbatim}
and,or :: [Bool] -> Bool
and xs = foldr (&&) True xs
or xs = foldr (||) False xs
?  or [True,False,False] ⇒
    foldr (||) False [True,False,False] ⇒
    True || (False || (False || False)) ⇒ True
\end{verbatim}
fold Functions...

- Remember that `foldr` binds from the right:
  \[
  \text{foldr} \ (+) \ 0 \ [1,2,3] \Rightarrow (1+(2+(3+0)))
  \]

- There is another function `foldl` that binds from the left:
  \[
  \text{foldl} \ (+) \ 0 \ [1,2,3] \Rightarrow (((0+1)+2)+3)
  \]

- In general:
  \[
  \text{foldl}(\oplus)z[x_1 \cdots x_n] = (((z \oplus x_1) \oplus x_2) \oplus \cdots \oplus x_n)
  \]
In the case of (+) and many other functions

\[
\text{foldl}(\oplus)z[x_1 \cdots x_n] = \text{foldr}(\oplus)z[x_1 \cdots x_n]
\]

However, one version may be more efficient than the other.
fold Functions...

\[
\text{foldr} \odot z [x_1 \cdots x_n] \quad \text{foldl} \odot z [x_1 \cdots x_n]
\]
We’ve already seen that it is possible to use operators to construct new functions:

\((*2)\) – function that doubles its argument
\((>2)\) – function that returns \texttt{True} for numbers \(> 2\).

Such partially applied operators are known as operator sections. There are two kinds:

\[
\text{(op a) b = b op a}
\]

\((*2) 4\) = \(4 * 2 = 8\)
\((>2) 4\) = \(4 > 2 = \text{True}\)
\((++ "\n") \text{"Bart"} = \text{"Bart" ++ "\n"}\)
Operator Sections...

\[(a \ op) b = a \ op b\]

\[(3:) \ [1,2] = 3 : [1,2] = [3,1,2]\]

\[(0<) \ 5 = 0 < 5 = True\]

\[(1/) = 1/5\]

Examples:

(+1) – The successor function.

(/2) – The halving function.

(:[]) – The function that turns an element into a singleton list.

More Examples:

? filter (0<) (map (+1) [-2,-1,0,1])

[-1]
We’ve looked at the list-breaking functions \texttt{drop} & \texttt{take}:

\begin{align*}
\text{take 2 } ["a","b","c"] & \Rightarrow ["a","b"] \\
\text{drop 2 } ["a","b","c"] & \Rightarrow ["c"]
\end{align*}

\texttt{takeWhile} and \texttt{dropWhile} are higher-order list-breaking functions. They take/drop elements from a list while a predicate is true.

\begin{align*}
\text{takeWhile even } [2,4,6,5,7,4,1] & \Rightarrow [2,4,6] \\
\text{dropWhile even } [2,4,6,5,7,4,1] & \Rightarrow [5,7,4,1]
\end{align*}
takeWhile & dropWhile...

takeWhile :: (a->Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x:xs)
  | p x      = x : takeWhile p xs
  | otherwise = []

dropWhile :: (a->Bool) -> [a] -> [a]
dropWhile p [] = []
dropWhile p (x:xs)
  | p x      = dropWhile p xs
  | otherwise = x:xs
takeWhile \& dropWhile...

Remove initial/final blanks from a string:

dropWhile \((==) \ ' \_\_\_'\) "\_\_\_Hi!\" \Rightarrow "Hi!"

takeWhile \((/=) \ ' \_\_\_'\) "Hi!\_\_\_\_\_\_\_\_\" \Rightarrow "Hi!"
Summary

- Higher-order functions take functions as arguments, or return a function as the result.
- We can form a new function by applying a curried function to some (but not all) of its arguments. This is called partial application.
- Operator sections are partially applied infix operators.
The standard prelude contains many useful higher-order functions:

**map** \( f \ \text{xs} \) creates a new list by applying the function \( f \) to every element of a list \( \text{xs} \).

**filter** \( p \ \text{xs} \) creates a new list by selecting only those elements from \( \text{xs} \) that satisfy the predicate \( p \) (i.e. \((p x)\) should return True).

**foldr** \( f \ z \ \text{xs} \) reduces a list \( \text{xs} \) down to one element, by applying the binary function \( f \) to successive elements, starting from the right.

**scanl*/**\( \text{scanr} \) \( f \ z \ \text{xs} \) perform the same functions as **foldr*/**\( \text{foldl} \), but instead of returning only the ultimate value they return a list of all intermediate results.
Homework

Homework (a):
Define the map function using a list comprehension.

Template:
map f x = [ ... | ... ]

Homework (b):
Use map to define a function lengthall xss which takes a list of strings xss as argument and returns a list of their lengths as result.

Examples:
? lengthall ["Ay", "Caramba!"]
[2,8]
Homework

1. Give a accumulative recursive definition of \texttt{foldl}.

2. Define the \texttt{minimum} \texttt{xs} function using \texttt{foldr}.

3. Define a function \texttt{sumsq} \texttt{n} that returns the sum of the squares of the numbers \texttt{[1 \cdots n]}. Use \texttt{map} and \texttt{foldr}.

4. What does the function \texttt{mystery} below do?

\begin{verbatim}
mystery xs =
    foldr (++) [] (map sing xs)
sing x = [x]
\end{verbatim}

\underline{Examples:}

\texttt{minimum [3,4,1,5,6,3] \Rightarrow 1}
Define a function \( \text{zipp } f \ xs \ ys \) that takes a function \( f \) and two lists \( xs = [x_1, \ldots, x_n] \) and \( ys = [y_1, \ldots, y_n] \) as argument, and returns the list \( [f \ x_1 \ y_1, \ldots, f \ x_n \ y_n] \) as result.

If the lists are of unequal length, an error should be returned.

**Examples:**

\[
\text{zipp } (+) \ [1,2,3] \ [4,5,6] \Rightarrow [5,7,9]
\]

\[
\text{zipp } (==) \ [1,2,3] \ [4,2,2] \Rightarrow [\text{False}, \text{True}, \text{True}]
\]

\[
\text{zipp } (==) \ [1,2,3] \ [4,2] \Rightarrow \text{ERROR}
\]
Define a function \( \text{filterFirst} \ p \ \text{xs} \) that removes the first element of \( \text{xs} \) that does not have the property \( p \).

**Example:**

\[
\text{filterFirst even \ [2,4,6,5,6,8,7] } \Rightarrow \ [2,4,6,6,8,7]
\]

Use \( \text{filterFirst} \) to define a function \( \text{filterLast} \ p \ \text{xs} \) that removes the last occurrence of an element of \( \text{xs} \) without the property \( p \).

**Example:**

\[
\text{filterLast even \ [2,4,6,5,6,8,7] } \Rightarrow \ [2,4,6,5,6,8]
\]