Higher-Order Functions

- 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:

```haskell
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.

  **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:

    ```haskell
    fun :: t_1 -> t_2 -> \cdots -> t_n -> t
    fun a_1 a_2 \cdots a_n = \cdots
    ```

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

      ```haskell
      fun_1 :: t_2 -> \cdots -> t_n -> t
      fun_1 a_2 \cdots a_n = \cdots
      
      fun_2 :: t_3 -> \cdots -> t_n -> t
      fun_2 a_3 \cdots a_n = \cdots
      ```
Currying Revisited...

Duh, how about an example?

Certainly. Let's define a recursive function `get_nth n xs` which returns the `n`th element from the list `xs`:

- `get_nth 1 (x:_:xs) = x`
- `get_nth n (x:_:xs) = get_nth (n-1) xs`

Now, let's use `get_nth` to define functions `get_second`, `get_third`, `get_fourth`, and `get_fifth`, without using explicit recursion:

- `get_second = get_nth 2`
- `get_third = get_nth 3`
- `get_fourth = get_nth 4`
- `get_fifth = get_nth 5`

get_third 10 "Bartholomew" ⇒ 'h'

Remember the Rule of Cancellation?

The type of `get_nth` is `Int -> [a] -> 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: `Int -> [a] -> 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 `map` Function

- `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.

- `map`'s first argument is a function of type `a -> b`. The second argument is a list of type `[a]`. The result is a list of type `[b]`.

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

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

\texttt{map \textit{f} [ \textit{} \textit{}] = [ \textit{} \textit{}]}

\texttt{map \textit{f} (\textit{x:xs}) = \textit{f} \textit{x} : \texttt{map} \textit{f} \textit{xs}}

\begin{itemize}
  \item \texttt{inc \textit{x} = \textit{x} + 1}
  \item \texttt{map inc [1,2,3,4]} \Rightarrow \texttt{[2,3,4,5]}
\end{itemize}

\texttt{Simulation:}

\begin{itemize}
  \item \texttt{map square [5,6] \Rightarrow}
  \item \texttt{\textit{square 5} : \texttt{map square [6]} \Rightarrow}
  \item \texttt{25 : \texttt{map square [6]} \Rightarrow}
  \item \texttt{25 : (\textit{square 6} : \texttt{map square [ ]}) \Rightarrow}
  \item \texttt{25 : (36 : \texttt{map square [ ]}) \Rightarrow}
  \item \texttt{25 : [36] \Rightarrow}
  \item \texttt{[25,36]}
\end{itemize}

The \textbf{filter} Function

\begin{itemize}
  \item \texttt{Filter takes a predicate \textit{p} and a list \textit{L} as arguments. It returns a list \textit{L'} consisting of those elements from \textit{L} that satisfy \textit{p}.}
  \item \texttt{The predicate \textit{p} should have the type \textit{a} \rightarrow \texttt{Bool}, where \textit{a} is the type of the list elements.}
\end{itemize}

\texttt{Examples:}

\begin{itemize}
  \item \texttt{filter even [1..10] \Rightarrow [2,4,6,8,10]}
  \item \texttt{filter even (map square [2..5]) \Rightarrow}
  \item \texttt{filter even [4,9,16,25] \Rightarrow [4,16]}
  \item \texttt{filter gt10 [2,5,9,11,23,114]}
  \item \texttt{where gt10 \textit{x} = \textit{x} > 10 \Rightarrow [11,23,114]}
\end{itemize}
The filter Function...

We can define \( \text{filter} \) using either recursion or list comprehension.

**Using recursion:**
\[
\text{filter} :: (a \to \text{Bool}) \to [a] \to [a] \\
\text{filter } \_ \ [\_] \ = \ [\_] \\
\text{filter } p \ (x:xs) \ = \ \begin{cases} 
  x : \text{filter } p \ xs & \text{if } p \ x \\
  \text{filter } p \ xs & \text{otherwise}
\end{cases}
\]

**Using list comprehension:**
\[
\text{filter} :: (a \to \text{Bool}) \to [a] \to [a] \\
\text{filter } p \ xs = [x | x \leftarrow xs, p \ x]
\]

The filter Function...

doublePos doubles the positive integers in a list.

getEven :: [Int] \to [Int]  
getEven \( \, xs = \text{filter } \text{even } xs \)

doublePos :: [Int] \to [Int]  
doublePos \( \, xs = \text{map } \text{dbl} (\text{filter } \text{pos } xs) \)  
where \( \text{dbl } x = 2 * x \)  
\( \text{pos } x = x > 0 \)

Simulation:
getEven [1,2,3] \( \Rightarrow \) [2]

doublePos [1,2,3,4] \( \Rightarrow \) 
\( \text{map } \text{dbl} (\text{filter } \text{pos } [1,2,3,4]) \Rightarrow \)  
\( \text{map } \text{dbl} [2,4] \Rightarrow \) [4,8]

fold Functions

A common operation is to combine the elements of a list into one element. Such operations are called reductions or accumulations.

**Examples:**
sum [1,2,3,4,5] \( \equiv \) 
\( (1 + (2 + (3 + (4 + (5 + 0)))))) \Rightarrow 15 \)

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 `foldr` ("fold right") which captures this pattern of computation.

`foldr` takes three arguments: a function, a seed value, and a list.

**Examples:**

```
foldr (+) 0 [1,2,3,4,5] ⇒ 15
foldr (++) "" ["H","i","!"] ⇒ "Hi!"
```

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

Note how the fold process is started by combining the last element $x_n$ with $z$. Hence the name seed.

```
foldr(+)z[x_1⋯x_n] = (x_1 ⊕ (x_2 ⊕ (⋯(x_n ⊕ z))))
```

Several functions in the standard prelude are defined using `foldr`:

```
and, or :: [Bool] -> Bool
and xs = foldr (&&) True xs
or xs = foldr (||) False xs
```

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

Remember that `foldr` binds from the right:

```
foldr (+) 0 [1,2,3] ⇒ (1+(2+(3+0)))
```

There is another function `foldl` that binds from the left:

```
foldl (+) 0 [1,2,3] ⇒ ((0+1)+2)+3
```

In general:

```
foldl(+)z[x_1⋯x_n] = foldr(+)z[x_1⋯x_n]
```

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

\[
\begin{align*}
\text{foldr } \oplus z \ [x_1 \ldots x_n] \\
\text{foldl } \oplus z \ [x_1 \ldots x_n]
\end{align*}
\]

Operator Sections

- We've already seen that it is possible to use operators to construct new functions:

  \((\times 2)\) – function that doubles its argument
  \((>2)\) – function that returns True for numbers > 2.

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

\[
\begin{align*}
\text{(op a) b} &= b \text{ op a} \\
\text{(*2) } 4 &= 4 \times 2 = 8 \\
\text{(>2) } 4 &= 4 > 2 = \text{True} \\
\text{(+"\n") } \text{"Bart"} &= \text{"Bart" ++ "\n"}
\end{align*}
\]

Operator Sections...

- \((a \text{ op}) b = a \text{ op } b\)

\[
\begin{align*}
\text{(3:) } [1,2] &= 3 : [1,2] = [3,1,2] \\
\text{(0<) } 5 &= 0 < 5 = \text{True} \\
\text{(1/) } &= 1/5
\end{align*}
\]

Examples:

- \((+1)\) – The successor function.
- \((/2)\) – The halving function.
- \((:[])\) – The function that turns an element into a singleton list.

More Examples:

- \(? \text{ filter (0<) (map (+1) [-2,-1,0,1])} \Rightarrow [-1]\)

takeWhile & dropWhile

- We've looked at the list-breaking functions \(\text{drop} \) & \(\text{take}:\)

  \(\text{take 2 ['a','b','c']} \Rightarrow ['a','b']\)
  \(\text{drop 2 ['a','b','c']} \Rightarrow ['c']\)

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

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

**takeWhile & dropWhile...**

```haskell
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
```

**Remove initial/final blanks from a string:**

```haskell
dropWhile ((==) ' ') " Hi! " => "Hi!
```

```haskell
takeWhile ((/=) ' ') " Hi! "
```

**The standard prelude contains many useful higher-order functions:**

- `map f xs` creates a new list by applying the function `f` to every element of a list `xs`.
- `filter p xs` creates a new list by selecting only those elements from `xs` that satisfy the predicate `p` (i.e. `(p x)` should return `True`).
- `foldr f z xs` reduces a list `xs` down to one element, by applying the binary function `f` to successive elements, starting from the right.
- `scanl/scanr f z xs` perform the same functions as `foldr/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 \( \text{lengthall} \) \( xss \) which takes a list of strings \( xss \) as argument and returns a list of their lengths as result.

Examples:
? \( \text{lengthall} \) ["Ay", "Caramba!"]
[2, 8]

Homework...

- 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] ⇒ [5,7,9]
\( \text{zipp} \) (==) [1,2,3] [4,2,2] ⇒ [False,True,True]
\( \text{zipp} \) (==) [1,2,3] [4,2] ⇒ ERROR

Homework

1. Give a accumulative recursive definition of \( \text{foldl} \).
2. Define the \( \text{minimum} \) \( xs \) function using \( \text{foldr} \).
3. Define a function \( \text{sumsq} \ n \) that returns the sum of the squares of the numbers \([1 \ldots n] \). Use map and \( \text{foldr} \).
4. What does the function mystery below do?

\[
mystery \ xss = \text{foldr} \ (++) \ [\] \ (\text{map} \ \text{sing} \ xss) \\
n \text{sing} \ x = [x]
\]

Examples:
\( \text{minimum} \) [3,4,1,5,6,3] ⇒ 1

Homework

- Define a function \( \text{filterFirst} \) \( p \) \( xs \) that removes the first element of \( xs \) that does not have the property \( p \).

Example:
\( \text{filterFirst} \) even [2,4,6,5,6,8,7] ⇒ [2,4,6,6,8,7]

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

Example:
\( \text{filterLast} \) even [2,4,6,5,6,8,7] ⇒ [2,4,6,5,6,8]