Higher-Order Functions

- A function is **Higher-Order** if it takes a function as an argument or returns one as its result.
- Higher-order functions aren’t weird; the differentiation operation from high-school calculus is higher-order:

  \[ \text{deriv} :: (\text{Float} \to \text{Float}) \to \text{Float} \to \text{Float} \]

  \[ \text{deriv} \ f \ x = (f(x+dx) - f \ x) / 0.0001 \]

- 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.

  \[ \text{fun} :: t_{1} \to t_{2} \to \cdots \to t_{n} \to t \]

  \[ \text{fun}_{1} \ a_{2} \cdots a_{n} = \cdots \]

  \[ \text{fun}_{2} :: t_{3} \to \cdots \to t_{n} \to t \]

  \[ \text{fun}_{2} \ a_{3} \cdots a_{n} = \cdots \]

**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...

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

`get_nth 10 "Bartholomew" ⇒ 'e'`

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_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 `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:

```
```

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]`.

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

We can check the type of an object using the `:type` command. Example: `:type map`. 
The map Function

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

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

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

\[
\begin{align*}
\text{map inc } [1,2,3,4] &= [2,3,4,5] \\
\end{align*}
\]

Simulation:

\[
\begin{align*}
\text{map square } [5,6] &\Rightarrow \\
\text{square } 5 : \text{map square } [6] &\Rightarrow \\
25 : \text{map square } [6] &\Rightarrow \\
25 : (\text{square } 6 : \text{map square } [ ] ) &\Rightarrow \\
25 : (36 : \text{map square } [ ] ) &\Rightarrow \\
25 : [36] &\Rightarrow \\
[25,36] &\Rightarrow
\end{align*}
\]

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 \(a \to \text{Bool}\), where \(a\) is the type of the list elements.

Examples:

\[
\begin{align*}
\text{filter even } [1..10] &\Rightarrow [2,4,6,8,10] \\
\text{filter even } (\text{map square } [2..5]) &\Rightarrow \\
\text{filter even } [4,9,16,25] &\Rightarrow [4,16] \\
\text{filter gt10 } [2,5,9,11,23,114] &\Rightarrow [11,23,114]
\end{align*}
\]

where \(\text{gt10 } x = x > 10\)
The filter Function...

- We can define filter using either recursion or list comprehension.

__________________________ Using recursion: ________________________

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

__________________________ Using list comprehension: ________________________

filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]

The filter Function...

- We can define filter using either recursion or list comprehension.

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.

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

getEven [1,2,3] ⇒ [2]
doublePos [1,2,3,4] ⇒
  map dbl (filter pos [1,2,3,4]) ⇒
  map dbl [2,4] ⇒ [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] ⇒
  (1 + (2 + (3 + (4 + (5 + 0))))) ⇒ 15
concat ["H","i","!" ] ⇒
  ("H" ++ ("i" ++ ("!" ++ " "))) ⇒ "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:

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

In general:

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

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

\[
\text{and, or :: } [\text{Bool}] \to \text{Bool}
\]

\[
\text{and xs = foldr } (\&\&) \ \text{True} \ \text{xs} \\
\text{or xs = foldr } (||) \ \text{False} \ \text{xs}
\]

- 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

\[ \text{foldr} \oplus z [x_1 \cdots x_n] \]

\[ \text{foldl} \oplus z [x_1 \cdots x_n] \]

Operator Sections

- We’ve already seen that it is possible to use operators to construct new functions:
  
  \((\ast 2)\) – 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:

\[ (\text{op \hspace{1mm} a}) \hspace{1mm} b = b \text{ \hspace{1mm} op \hspace{1mm} a} \]

\((\ast 2)\) 4 = 4 * 2 = 8

\((> 2)\) 4 = 4 > 2 = True

\((++ \text{" \backslash n"})\) "Bart" = "Bart" ++ "\n"

Operator Sections...

\[ (\text{a \hspace{1mm} op}) \hspace{1mm} b = a \hspace{1mm} \text{op} \hspace{1mm} b \]

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

(0<) 5 = 0 < 5 = True

(1/) 5 = 1/5

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])} \]

\[ [1,2] \]

takeWhile & dropWhile

- We’ve looked at the list-breaking functions drop & take:

\take 2 ['a','b','c'] \Rightarrow ['a','b']

\drop 2 ['a','b','c'] \Rightarrow ['c']

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

\takeWhile even [2,4,6,5,7,4,1] \Rightarrow [2,4,6]

\dropWhile even [2,4,6,5,7,4,1] \Rightarrow [5,7,4,1]
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!"
takeWhile ((/=) '␣') "Hi!␣␣␣" ⇒ "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 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 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...  

定义一个函数 zipp f xs ys，它接受一个函数 f 和两个列表 xs=[x_1, ⋯ , x_n] 和 ys=[y_1, ⋯ , y_n] 作为参数，并返回列表 [f x_1 y_1, ⋯ , f x_n y_n] 作为结果。

如果两个列表的长度不相等，应返回错误。

Examples:

zipp (+) [1,2,3] [4,5,6] ⇒ [5,7,9]
zipp (==) [1,2,3] [4,2,2] ⇒ [False,True,True]
zipp (==) [1,2,3] [4,2] ⇒ ERROR

Homework

1. 给出一个累积递归的 fold1 定义。
2. 定义最小的 xs 函数使用 foldr。
3. 定义一个函数 sumsq n，它返回数字 [1 ⋯ n] 的平方和。使用 map 和 foldr。
4. 这个函数 mystery 有什么作用？

mystery xs = foldr (+) [] (map sing xs)
sing x = [x]

Examples:

minimum [3,4,1,5,6,3] ⇒ 1

Homework

定义一个函数 filterFirst p xs，它移除 xs 中第一个不具有属性 p 的元素。

Example:

filterFirst even [2,4,6,5,6,8,7] ⇒ [2,4,6,6,8,7]

Use filterFirst to define a function filterLast p xs that removes the last occurrence of an element of xs without the property p.

Example:

filterLast even [2,4,6,5,6,8,7] ⇒ [2,4,6,5,6,8]