Utrecht University

Lecture 4. Higher-order functions

Functional Programming

Utrecht University

Why learn (typed) functional programming?

Why Haskell?

Goal of typed purely functional programming

Keep programs easy to reason about by

- data-flow only through function arguments and return values
  - no hidden data-flow through mutable variables/state

Goal of typed purely functional programming

Keep programs easy to reason about by

- data-flow only through function arguments and return values
  - no hidden data-flow through mutable variables/state
- function call and return as only control-flow primitive
  - no loops, `break`, `continue`, `goto`

Goal of typed purely functional programming

Keep programs easy to reason about by

- data-flow only through function arguments and return values
  - no hidden data-flow through mutable variables/state
- function call and return as only control-flow primitive
  - no loops, `break`, `continue`, `goto`
- (almost) unique types
  - no inheritance hell

Goal of typed purely functional programming

Keep programs easy to reason about by

- data-flow only through function arguments and return values
    - no hidden data-flow through mutable variables/state
- function call and return as only control-flow primitive
    - no loops, `break`, `continue`, `goto`
- (almost) unique types
    - no inheritance hell
- high-level declarative data-structures
    - no explicit reference-based data structures

Goal of typed purely functional programming

Keep programs easy to reason about by

- function call and return as only control-flow primitive
  - no loops, `break`, `continue`, `goto`
  - instead: **higher-order functions** (functions which use other functions)
  - extra pay-off: huge abstraction power -> more code reuse!

The remaining two: this Thursday!

Goals of today

- Define and use higher-order functions
  - Functions which use other functions
  - In particular, `map`, `filter`, `foldr` and `foldl`
  - vs general recursion
- Use anonymous functions
- Understand function composition
- Understand partial application

Chapter 7 and 4.5-4.6 from Hutton's book

Higher-order functions vs curried functions

- Curried functions (of multiple arguments):

  ```
  f :: a -> b -> c
  ```

  read

  ```
  f :: a -> (b -> c)
  ```

- Higher-order functions:

  ```
  f :: (a -> b) -> c
  ```

- Exercise: come up with some examples from high school mathematics

What can higher-order functions do?

- How can we use argument-functions?
- Can we pattern match on them?
- Can we inspect their source code from a higher-order function?

What can higher-order functions do?

- How can we use argument-functions?
  - By applying them! That's it!
- Can we pattern match on them?
  - No! But we can feed them inputs and pattern match on the results!
- Can we inspect their source code from a higher-order function?
  - No! Only their input-output behaviour!

Usage of map

From the previous lectures...

- map applies a function uniformly over a list
  - The function to apply is an *argument* to map

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

```
> map length ["a", "abc", "ab"]
[1,3,2]
```

- It is very similar to a list comprehension

```
> [length s | s <- ["a", "abc", "ab"]]
[1,3,2]
```

Cooking `map`

1. Define the type

```
map :: _
```

2. Enumerate the cases

   - We **cannot** pattern match on functions

```
map f []     = _
map f (x:xs) = _
```

Try it yourself!

Cooking `map`

1. Define the type

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

2. Enumerate the cases
   - We **cannot** pattern match on functions

```
map f []     = _
map f (x:xs) = _
```

3. Define the simple (base) cases

```
map f []     = []
```

Cooking `map`

4. Define the other (recursive) cases
    - The current element needs to be transformed by `f`
    - The rest are transformed uniformly by `map`

```
map f (x:xs) = f x : map f xs
```

It makes **no difference** whether the function we use is global or is an argument

Usage of `filter`

`filter p xs` leaves only the elements in `xs` which satisfy the predicate p

- A predicate is a function which returns `True` or `False`
- In other words, p must return `Bool`

```
> even x = x `mod` 2 == 0
> filter even [1 .. 4]
[2,4]
> largerThan10 x = x > 10
> filter largerThan10 [1 .. 4]
[]
```

Cooking `filter`

1. Define the type

   ```
   filter :: _
   ```

2. Enumerate the cases

   ```
   filter p []     = _
   filter p (x:xs) = _
   ```

Try it yourself!

Cooking `filter`

1. Define the type

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

2. Enumerate the cases

   ```
   filter p []     = _
   filter p (x:xs) = _
   ```

3. Define the simple (base) cases

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

Cooking `filter`

4. Define the other (recursive) cases
   - We have to distinguish whether the predicate holds
   - Version 1, using conditionals

```
filter p (x:xs) = if p x
                     then x : filter p xs
                     else     filter p xs
```

   - Version 2, using guards

```
filter p (x:xs) | p x       = x : filter p xs
                | otherwise =     filter p xs
```

Alternative definitions using comprehensions

`map` and `filter` can be easily defined using comprehensions

```
map    f xs = [f x | x <- xs]

filter p xs = [x   | x <- xs, p x]
```

The recursive definitions are better to reason about code

(Ab)use of local definitions

Suppose we want to double the numbers in a list

- We can define a `double` function and apply it to the list

```
double n = 2 * n
doubleList xs = map double xs
```

(Ab)use of local definitions

Suppose we want to double the numbers in a list

- We can define a `double` function and apply it to the list

```
double n = 2 * n
doubleList xs = map double xs
```

- This pollutes the code, so we can put it in a `where`

```
doubleList xs = map double xs
  where double n = 2 * n
```

(Ab)use of local definitions

Suppose we want to double the numbers in a list

- We can define a `double` function and apply it to the list

```
double n = 2 * n
doubleList xs = map double xs
```

- This pollutes the code, so we can put it in a `where`

```
doubleList xs = map double xs
  where double n = 2 * n
```

- But we are still using too much code for such a simple and small function!
  - Each call to `map` or `filter` may require one of those

Anonymous functions

\ arguments -> code

Haskell allows you to define functions without a name

```
doubleList xs = map (\x -> 2 * x) xs
```

- They are called **anonymous functions** or **(lambda) abstractions**
- The `\` symbol resembles a Greek λ

Anonymous functions

\ arguments -> code

Haskell allows you to define functions without a name

```
doubleList xs = map (\x -> 2 * x) xs
```

- They are called **anonymous functions** or **(lambda) abstractions**
- The `\` symbol resembles a Greek λ

Historical note: the theoretical basis for functional programming is called λ-calculus and was introduced in the 1930s by the American mathematician Alonzo Church

Anonymous functions are just functions

- They have a type, which is always a function type

```
> :t \x -> 2 * x
\x -> 2 * x :: Num a => a -> a
```

Anonymous functions are just functions

- They have a type, which is always a function type

```
> :t \x -> 2 * x
\x -> 2 * x :: Num a => a -> a
```

- You can use it everywhere you need a function

```
> (\x -> 2 * x) 3
6
> filter (\x -> x > 10) [1 .. 20]
[11,12,13,14,15,16,17,18,19,20]
```

Anonymous functions are just functions

- They have a type, which is always a function type

```
> :t \x -> 2 * x
\x -> 2 * x :: Num a => a -> a
```

- You can use it everywhere you need a function

```
> (\x -> 2 * x) 3
6
> filter (\x -> x > 10) [1 .. 20]
[11,12,13,14,15,16,17,18,19,20]
```

- Even when you define a function

```
double = \x -> 2 * x
```

Functions which return functions

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f = _
```

Functions which return functions

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f = \y x -> f x y
```

- This function is called a **combinator**
  - It creates a function from another function
- The resulting function may get more arguments
  - They appear in reverse order from the original

```
> flip map [1,2,3] (\x -> 2 * x)
[2,4,6]
```

Functions are curried

- In Haskell, functions take one argument at a time
  - The result might be another function

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

  - We say functions in Haskell are **curried**
- A two-argument function is actually a one-argument function which returns yet another function which takes the next argument and produces a result

Different ways to write

Take a function with three arguments

```haskell
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

Parentheses in functions associate to the right

```haskell
addThree :: Int -> (Int -> (Int -> Int))
```

Different ways to write

Take a function with three arguments

```
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

Parentheses in functions associate to the right

```
addThree :: Int -> (Int -> (Int -> Int))
```

We can define the function in these other ways

```
addThree x y =             \z -> x + y + z
addThree x   =       \y -> \z -> x + y + z
addThree     = \x -> \y -> \z -> x + y + z
addThree     = \x    y     z -> x + y + z
```

Partial application

- Since Haskell functions take one argument at a time, we can provide less than the ones stated in the signature
  - The result is yet another function
  - We say the function has been **partially appplied**

```
> :t map (\x -> 2 * x)
map (\x -> 2 * x) :: ???
```

Partial application

- Since Haskell functions take one argument at a time, we can provide less than the ones stated in the signature
  - The result is yet another function
  - We say the function has been **partially appplied**

```
> :t map (\x -> 2 * x)
map (\x -> 2 * x) :: Num b => [b] -> [b]
```

```
> :{
| let doubleList = map (\x -> 2 * x)
| in doubleList [1,2,3]
| :}
[2,4,6]
```

Definition by partial application

Instead of writing out all the arguments

```
doubleList xs = map (\x -> 2 * x) xs
```

Haskells make use of partial application if possible

```
doubleList    = map (\x -> 2 * x)
```

Note that `xs` has been dropped from **both** sides

Definition by partial application

Instead of writing out all the arguments

```haskell
doubleList xs = map (\x -> 2 * x) xs
```

Haskells make use of partial application if possible

```haskell
doubleList    = map (\x -> 2 * x)
```

Note that `xs` has been dropped from **both** sides

Technical note: this is called η (eta) reduction

Sections

Sections are shorthand for partial application of operators

```
(x #) = \y -> x # y  -- Application of 1st arg.
(# y) = \x -> x # y  -- Application of 2nd arg.
```

They help us remove even more clutter

```
doubleList    = map (2 *)
largerThan10 = filter (> 10)
```

Sections

Sections are shorthand for partial application of operators

```
(x #) = \y -> x # y  -- Application of 1st arg.
(# y) = \x -> x # y  -- Application of 2nd arg.
```

They help us remove even more clutter

```
doubleList    = map (2 *)
largerThan10 = filter (> 10)
```

Warning! Order matters in sections

```
> filter (> 10) [1 .. 20]
[11,12,13,14,15,16,17,18,19,20]
> filter (10 >) [1 .. 20]
[1,2,3,4,5,6,7,8,9]
```

Example: working with a list of functions

Apply a list of functions in order to a starting argument

```
> applyAll [(+ 1), (* 2), (\x -> x - 3)] 3
5  -- ((3 + 1) * 2) - 3
```

- Define the function
- What is the type of `applyAll`?

Try it yourself!

Example: working with a list of functions

```
applyAll [f]      x = f x
applyAll (f : fs) x = applyAll fs (f x)
```

Let’s think harder about the base case!

Example: working with a list of functions

```
applyAll [f]      x = f x
applyAll (f : fs) x = applyAll fs (f x)
```

Let’s think harder about the base case!

```
applyAll []       x = x
applyAll (f : fs) x = applyAll fs (f x)
```

Example: working with a list of functions

```
applyAll [f]       x = f x
applyAll (f : fs) x = applyAll fs (f x)
```

Let's think harder about the base case!

```
applyAll []        x = x
applyAll (f : fs) x = applyAll fs (f x)
```

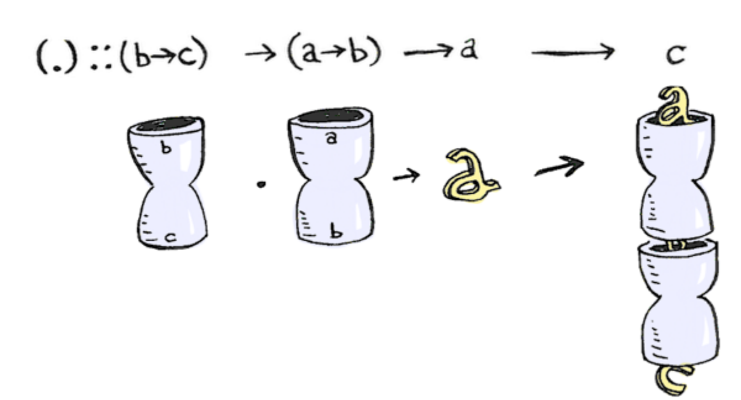
```
> :t applyAll
applyAll :: [a -> a] -> a -> a
```

Function composition

Another example of function combinator

- g *composed with* `f`, or g *after* `f`

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
g . f = _
```

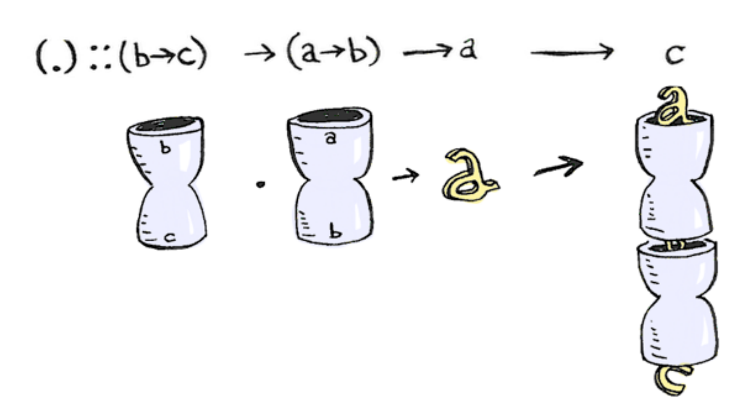


Function composition

Another example of function combinator

- g *composed with* `f`, or g *after* `f`

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
g . f = \x -> g (f x)
```



Examples of function composition

```haskell
not  :: Bool -> Bool
even :: Int  -> Bool

odd x = not (even x)
odd   = not . even  -- Better

-- Remove all elements which satisfy the predicate
filterNot :: (a -> Bool) -> [a] -> [a]
```

Try it yourself!

Examples of function composition

```haskell
not  :: Bool -> Bool
even :: Int  -> Bool

odd x = not (even x)
odd   = not . even  -- Better

-- Remove all elements which satisfy the predicate
filterNot :: (a -> Bool) -> [a] -> [a]
filterNot p xs = filter (\x -> not (p x)) xs
filterNot p xs = filter (not . p) xs  -- Better
filterNot p    = filter (not . p)     -- Even better
```

Function pipelines

You can define many functions as a **pipeline**

- Sequence of functions composed one after the other
- This style of coding is called *point-free*
  - Even though it actually has more point symbols!

```
maxAverage :: [[Float]] -> Float
maxAverage
  = maximum . map average . filter (not . null)
  where average xs
          = sum xs / fromIntegral (length xs)
```

Point-free craziness

You can go even further in this point-free style by using more combinators

```
where average = (/) <$> sum
                    <*> (fromIntegral . length)
```

```
(<$>) ::      (a -> b) -> (c -> a) -> (c -> b)
(<*>) :: (c -> a -> b) -> (c -> a) -> (c -> b)
```

Warning! Don't overdo it!

- This definition of `average` is less readable

Question

Write `applyAll` in point-free style

```
applyAll []       x = x
applyAll (f : fs) x = applyAll fs (f x)
```

Hint: for the first case remember that `id x = x`

Question

Write `applyAll` in point-free style

```
applyAll []       x = x
applyAll (f : fs) x = applyAll fs (f x)
```

Hint: for the first case remember that `id x = x`

```
applyAll []       = id
applyAll (f : fs) = applyAll fs . f
```

Folds

Similar functions

```
sum []     = 0
sum (x:xs) = x + sum xs

product []     = 1
product (x:xs) = x * product xs

and []     = True
and (x:xs) = x && and xs
```

Similar functions

```
sum []     = 0
sum (x:xs) = x + sum xs

product []     = 1
product (x:xs) = x * product xs

and []     = True
and (x:xs) = x && and xs
```

- The three return a *value* in the `[]` case
- For the `x:xs` case, they *combine* the head with the result for the rest of the list
  - `(+)` for `sum`, `(*)` for `product`, `(&&)` for and

Avoid duplication, abstract!

```haskell
sum []     = 0
sum (x:xs) = x + sum xs
```

Let's replace the moving parts with arguments `f` and v

- First-class functions are key for abstraction

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ v []     = v
foldr f v (x:xs) = f x (foldr f v xs)
                 = x `f` foldr f v xs  -- Infix
```

Avoid duplication, abstract!

- The previous definitions become much shorter
- The use of `foldr` conveys an intention
  - They all compute a result by iteratively applying a function over all the elements in the list

```
sum     = foldr (+)  0
product = foldr (*)  1
and     = foldr (&&) True
```

foldr is for "fold right"

```
foldr (+) 0 (x : y : z : [])
=
x + foldr (+) 0 (y : z : [])
=
x + (y + foldr (+) 0 (z : []))
=
x + (y + (z + foldr 0 []))
=
x + (y + (z + 0))
```

- `foldr` introduces parentheses "to the right"
- Initial value is in innermost parentheses

Another view of `foldr`

```
foldr (+) 0 [x, y, z]
=
foldr (+) 0 (x : (y : (z : [ ])))
                 |    |    |  |
                 |    |    |  |
                 ↓    ↓    ↓  ↓
             (x + (y + (z +  0 )))
```

- `(:)` is replaced by the combination function
- `[]` is replaced by the initial value

`length` as a right fold

```haskell
length []     = 0
length (_:xs) = 1 + length xs

foldr _ v []     = v
foldr f v (x:xs) = f x (foldr f v xs)
```

We want to find `f` and v such that

$$length = foldr\ f\ v$$

Try it yourself!

`length` as a right fold

- Case of empty list, `[]`

```
length [] = 0
          = v = foldr f v []
```

length as a right fold

- Case of empty list, `[]`

```
length [] = 0
          = v = foldr f v []
```

- Case of cons, `x:xs`

```
length (x:xs) = 1 + length xs
              = f x (foldr f v xs)
              = -- Assuming we know it for xs
                f x (length xs)
```

  - We need to have a function such that

```
f x (length xs) = 1 + length xs
===> f x y = 1 + y
===> f     = \x y -> 1 + y
```

length as a right fold

In conclusion,

```
length = foldr (\_ y -> 1 + y) 0

length [1,2,3]
= -- definition of length
foldr (\_ y -> 1 + y) [1,2,3]
= -- application of foldr
1 + (1 + (1 + 0))
= -- perform addition
3
```

Left folds

```
foldr (+) 0 [x,y,z]
= (x + (y + (z + 0)))
```

Is it possible to have a "mirror" function `foldl`?

```
foldl (+) 0 [x,y,z]
= (((0 + x) + y) + z)
```

- Parenthesis associate to the left
- Initial value still in the innermost position

Calculating `foldl`

- The case for empty lists is the same as `foldr`

```
foldl f v [] = v
```

Calculating `foldl`

- The case for empty lists is the same as `foldr`

```
foldl f v [] = v
```

- For the general case, notice this fact:

```
  foldl (+)    0                 [x,y,z]
= foldl (+)   (0 + x)            [y,z]
= foldl (+)  ((0 + x) + y)       [z]
= foldl (+) (((0 + x) + y) + z)  []
```

  - The second argument works as an *accumulator*

```
foldl f v (x:xs) = foldl f (f v x) xs
```

foldr versus foldl

```
  foldr (+) 0 [1, 2, ..., n]
= 1 + foldr (+) 0 [2, ..., n]
= ... = 1 + (2 + (... + (n + 0)))
      = 1 + (2 + (... + n)) = ...

  foldl (+) 0 [1, 2, ..., n]
= foldl (+) (0 + 1) [2, ..., n]
= ... = foldl (+) (((0 + 1) + ...) + n) []
= (((0 + 1) + ...) + n)
= ((1 + ...) + n)  = ...
```

- With `foldr` and `foldl` you wait until the end to start combining

foldr versus foldl

```
  foldl' (+) 0 [1, 2, ..., n]
= foldl' (+) (0 + 1) [2, ..., n]
= foldl' (+) 1 [2, ..., n]  -- (!)
= foldl' (+) (1 + 2) [..., n]
= foldl' (+) 3 [..., n]     -- (!)
```

- With `foldr` and `foldl` you wait until the end to start combining
- With `foldl'` you compute the value “on the go”
  - `foldl'` is usually more efficient

`foldr` versus `foldl`

In the case of `(+)`, the result is the same

```
> foldr (+) 0 [1,2,3]
6
> foldl (+) 0 [1,2,3]
6
```

This is not the case for every function

```
> foldr (-) 0 [1,2,3]
2
> foldl (-) 0 [1,2,3]
-6
```

Monoids

One possible set of properties which ensure that the direction of folding does not matter

Monoids

One possible set of properties which ensure that the direction of folding does not matter

1. The initial value does not affect the outcome

   ```
   f v x = x = f v x      0 + x = x = x + 0
   ```

   - We say that `v` is an *identity* for `f`

Monoids

One possible set of properties which ensure that the direction of folding does not matter

1. The initial value does not affect the outcome

   ```
   f v x = x = f v x     0 + x = x = x + 0
   ```

   - We say that `v` is an *identity* for `f`

2. The way we parenthesize does not affect the outcome

   ```
   f (f x y) z = f x (f y z)
   (x + y) + z = x + (y + z)
   ```

   - We say that the operation `f` is *associative*

A data type with such an operation is called a **monoid**

Avoid explicit recursion

- `map`, `filter`, `foldr` and `foldl` abstract common *recursion patterns* over lists
  - Most functions can be written as a combination of those
- *Good style*: prefer using those functions over recursion

Why?

Avoid explicit recursion

- `map`, `filter`, `foldr` and `foldl` abstract common *recursion patterns* over lists
    - Most functions can be written as a combination of those
- *Good style*: prefer using those functions over recursion
    - The intention of the code is clearer
    - Less code written means less code to debug
    - Complex recursion suggest that you might be doing too much in one function
    - Primitive rather than general recursion: always terminates!

Avoid explicit recursion, example

`count p xs` counts how many elements in `xs` satisfy `p`

```
count :: (a -> Bool) -> [a] -> Int
count _ []     = 0
count p (x:xs) | p x       = 1 + count p xs
               | otherwise =     count p xs
```

Try it yourself!

Avoid explicit recursion, example

`count p xs` counts how many elements in `xs` satisfy `p`

```
count :: (a -> Bool) -> [a] -> Int
count _ []      = 0
count p (x:xs) | p x        = 1 + count p xs
               | otherwise =     count p xs
```

```
count p xs = length (filter p xs)
```

```
count p    = length . filter p
```

applyAll as a fold

```
applyAll []       x = x
applyAll (f : fs) x = applyAll fs (f x)
```

Is applyAll as a right or a left fold?

applyAll as a fold

```
applyAll []       x = x
applyAll (f : fs) x = applyAll fs (f x)
```

Is applyAll as a right or a left fold?

```
> applyAll [f1,f2,f3] x
f3 (f2 (f1 x))  -- start from the left value
```

applyAll as a fold

```
applyAll []       x = x
applyAll (f : fs) x = applyAll fs (f x)
```

Is applyAll as a right or a left fold?

```
> applyAll [f1,f2,f3] x
f3 (f2 (f1 x))  -- start from the left value
```

```
-- Solution 1
applyAll fs x = foldl (\y f -> f y) x fs
```

applyAll as a fold

```
applyAll []       = id
applyAll (f : fs) = applyAll fs . f
```

We can also see it as a series of compositions

```
> applyAll [f1,f2,f3]
id . (f3 . (f2 . f1))
```

applyAll as a fold

```haskell
applyAll []       = id
applyAll (f : fs) = applyAll fs . f
```

We can also see it as a series of compositions

```haskell
> applyAll [f1,f2,f3]
id . (f3 . (f2 . f1))
```

```haskell
-- Solution 2
applyAll fs = foldr (\r f -> f . r) id fs
```

Can we make it look better?

applyAll as a fold

```haskell
applyAll fs = foldr (\r f -> f . r)   id fs
-- Drop the argument in both sides
applyAll    = foldr (\r f -> f . r)   id
-- Use "normal" application order for (.)
applyAll    = foldr (\r f -> (.) f r) id
-- Use the flip combinator
applyAll    = foldr (flip (.))        id
-- "flip (.)" has a name for itself
applyAll    = foldr (>>>)             id
```

Important concepts

- Higher-order functions *use* functions
- Curried functions *return* functions

Important concepts

- Higher-order functions *use* functions
- Curried functions *return* functions

- Anonymous functions are introduced by `\x -> ...`
- All multi-argument functions in Haskell are curried
  - They take one parameter at a time

    ```
    f :: A -> (B -> (C -> D))
    ```

  - Functions can be partially applied

Important concepts

- Higher-order functions *use* functions
- Curried functions *return* functions

- Anonymous functions are introduced by `\x -> ...`
- All multi-argument functions in Haskell are curried
  - They take one parameter at a time

    ```
    f :: A -> (B -> (C -> D))
    ```
  - Functions can be partially applied

- `map`, `filter`, `foldr` and `foldl` describe common recursion patterns over lists

Acknowledgements

Function composition image taken from

`adit.io/posts/2013-07-22-lenses-in-pictures.html`

A type inference question

Given a list of numbers, let’s create a list of “adders”, each of them adding this number to another given one

```
adders = map (\n -> \x -> n + x)
       = -- eta reducation
         map (\n -> (n +))
       = -- eta reduction
         map (+)
```

```
> [a 5 | a <- adders [1,2,3]]
[6,7,8]
```

A type inference question

Let us look at the types of the functions involved

```
(+) :: Int -> (Int -> Int)

-- Generalized type
map :: (a -> b) -> [a] -> [b]

-- In our case a      = Int
--              a -> b = Int -> (Int -> Int)
--        Thus,      b =         Int -> Int
map :: (Int -> Int -> Int)
    -> [Int] -> [Int -> Int]
```