

Informatics 1
Functional Programming Lecture 6

Map, filter, fold

Philip Wadler
University of Edinburgh

Informatics Student Support: InfBase

<http://informaticsstudentsupport.wordpress.com>

**Help desk for any
course**

Drop-in sessions

Monday-Friday
Week 3- revision week

AP 7.03

	Tutor	Inf1 - Introduction to Computation	Inf2 - Introduction to Algorithms and Data Structures	Inf2C - Introduction to Software Engineering	Inf2C - Introduction to Computer Systems	Discrete Mathematics and mathematical reasoning
MONDAY						
09:00						
10:00						
11:10						
12:10	Kim	x	x	x	x	x
13:10	Kim	x	x	x	x	x
14:10	Uday	x	x		x	
15:10	Uday	x	x		x	
16:10						
17:10						
TUESDAY						
09:00						
10:00						
11:10						
12:10	Zhicheng	x	x	x	x	x
13:10	Zhicheng	x	x	x	x	x
14:10	Ben	x	x	x	x	
15:10	Ben	x	x	x	x	
16:10						
17:10						
WEDNESDAY						
09:00						
10:00						
11:00	Oliver	x	x		x	

Final exam

During revision week

Thursday 5th December

09.30

14.30

Friday 6th December

09.30

Part I

Map

Squares

```
*Main> squares [1,-2,3]  
[1,4,9]
```

```
squares :: [Int] -> [Int]  
squares xs = [ x*x | x <- xs ]
```

```
squares :: [Int] -> [Int]  
squares [] = []  
squares (x:xs) = x*x : squares xs
```

Ords

```
*Main> ords "a2c3"  
[97,50,99,51]
```

```
ords :: [Char] -> [Int]  
ords xs = [ ord x | x <- xs ]
```

```
ords :: [Char] -> [Int]  
ords [] = []  
ords (x:xs) = ord x : ords xs
```

Map

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

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

Squares, revisited

```
*Main> squares [1,-2,3]  
[1,4,9]
```

```
squares :: [Int] -> [Int]  
squares xs = [ x*x | x <- xs ]
```

```
squares :: [Int] -> [Int]  
squares [] = []  
squares (x:xs) = x*x : squares xs
```

```
squares :: [Int] -> [Int]  
squares xs = map sqr xs  
  where  
    sqr x = x*x
```


Map—how it works

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

```
map sqr [1,2,3]
=
[  sqr x | x <- [1,2,3] ]
=
[  sqr 1 ] ++ [  sqr 2 ] ++ [  sqr 3 ]
=
[1, 4, 9]
```

Map—how it works

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

```
map sqr [1,2,3]
=
map sqr (1 : (2 : (3 : [])))
=
sqr 1 : map sqr (2 : (3 : []))
=
sqr 1 : (sqr 2 : map sqr (3 : []))
=
sqr 1 : (sqr 2 : (sqr 3 : map sqr []))
=
sqr 1 : (sqr 2 : (sqr 3 : []))
=
1 : (4 : (9 : []))
=
[1, 4, 9]
```

Ords, revisited

```
*Main> ords "a2c3"  
[97, 50, 99, 51]
```

```
ords :: [Char] -> [Int]  
ords xs = [ ord x | x <- xs ]
```

```
ords :: [Char] -> [Int]  
ords [] = []  
ords (x:xs) = ord x : ords xs
```

```
ords :: [Char] -> [Int]  
ords xs = map ord xs
```

Part II

Filter

Positives

```
*Main> positives [1,-2,3]  
[1,3]
```

```
positives :: [Int] -> [Int]  
positives xs = [ x | x <- xs, x > 0 ]
```

```
positives :: [Int] -> [Int]  
positives [] = []  
positives (x:xs) | x > 0 = x : positives xs  
                  | otherwise = positives xs
```

Digits

```
*Main> digits "a2c3"  
"23"
```

```
digits :: [Char] -> [Char]  
digits xs = [ x | x <- xs, isDigit x ]
```

```
digits :: [Char] -> [Char]  
digits [] = []  
digits (x:xs) | isDigit x = x : digits xs  
               | otherwise = digits xs
```

Filter

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

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

Positives, revisited

```
*Main> positives [1,-2,3]  
[1,3]
```

```
positives :: [Int] -> [Int]  
positives xs = [ x | x <- xs, x > 0 ]
```

```
positives :: [Int] -> [Int]  
positives [] = []  
positives (x:xs) | x > 0 = x : positives xs  
                  | otherwise = positives xs
```

```
positives :: [Int] -> [Int]  
positives xs = filter pos xs  
  where  
    pos x = x > 0
```


Digits, revisited

```
*Main> digits "a2c3"  
"23"
```

```
digits :: [Char] -> [Char]  
digits xs = [ x | x <- xs, isDigit x ]
```

```
digits :: [Char] -> [Char]  
digits [] = []  
digits (x:xs) | isDigit x = x : digits xs  
              | otherwise = digits xs
```

```
digits :: [Char] -> [Char]  
digits xs = filter isDigit xs
```

Part III

Fold

Sum

```
*Main> sum [1,2,3,4]
```

```
10
```

```
sum :: [Int] -> Int
```

```
sum [] = 0
```

```
sum (x:xs) = x + sum xs
```

Product

```
*Main> product [1,2,3,4]
```

```
24
```

```
product :: [Int] -> Int
```

```
product [] = 1
```

```
product (x:xs) = x * product xs
```

Concatenate

```
*Main> concat [[1,2,3],[4,5]]  
[1,2,3,4,5]
```

```
*Main> concat ["con","cat","en","ate"]  
"concatenate"
```

```
concat :: [[a]] -> [a]  
concat []      = []  
concat (xs:xss) = xs ++ concat xss
```

Foldr

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

Foldr, with infix notation

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

Sum, revisited

```
*Main> sum [1,2,3,4]
```

```
10
```

```
sum :: [Int] -> Int
```

```
sum [] = 0
```

```
sum (x:xs) = x + sum xs
```

```
sum :: [Int] -> Int
```

```
sum xs = foldr (+) 0 xs
```

Recall that `(+)` is the name of the addition function,

so `x + y` and `(+) x y` are equivalent.

Sum, Product, Concat

```
sum      :: [Int] -> Int
sum xs   = foldr (+) 0 xs
```

```
product  :: [Int] -> Int
product xs = foldr (*) 1 xs
```

```
concat   :: [[a]] -> [a]
concat xs = foldr (++) [] xs
```

Sum—how it works

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

```
    sum [1,2]
=
    sum (1 : (2 : []))
=
    1 + sum (2 : [])
=
    1 + (2 + sum [])
=
    1 + (2 + 0)
=
    3
```

Sum—how it works, revisited

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

```
sum :: [Int] -> Int
sum xs = foldr (+) 0 xs
```

```
sum [1,2]
=
foldr (+) 0 [1,2]
=
foldr (+) 0 (1 : (2 : []))
=
1 + (foldr (+) 0 (2 : []))
=
1 + (2 + (foldr (+) 0 []))
=
1 + (2 + 0)
=
3
```

Part IV

Map, Filter, and Fold

All together now!

Sum of Squares of Positives

```
f :: [Int] -> Int
f xs = sum (squares (positives xs))
```

```
f :: [Int] -> Int
f xs = sum [ x*x | x <- xs, x > 0 ]
```

```
f :: [Int] -> Int
f [] = []
f (x:xs)
  | x > 0      = (x*x) + f xs
  | otherwise  = f xs
```

```
f :: [Int] -> Int
f xs = foldr (+) 0 (map sqr (filter pos xs))
  where
    sqr x = x * x
    pos x = x > 0
```

Part V

Currying

How to add two numbers

```
add :: Int -> Int -> Int  
add x y = x + y
```

```
    add 3 4  
=  
    3 + 4  
=  
    7
```

How to add two numbers

```
add :: Int -> (Int -> Int)
(add x) y = x + y
```

```
(add 3) 4
=
3 + 4
=
7
```

A function of two numbers
is the same as
a function of the first number that returns
a function of the second number.

Currying

```
add :: Int -> (Int -> Int)
```

```
add x = g
```

```
  where
```

```
    g y = x + y
```

```
(add 3) 4
```

```
=
```

```
g 4
```

```
  where
```

```
    g y = 3 + y
```

```
=
```

```
3 + 4
```

```
=
```

```
7
```

A function of two numbers
is the same as
a function of the first number that returns
a function of the second number.

Currying

```
add :: Int -> Int -> Int
add x y = x + y
```

means the same as

```
add :: Int -> (Int -> Int)
add x = g
  where
    g y = x + y
```

and

```
add 3 4
```

means the same as

```
(add 3) 4
```

This idea is named for *Haskell Curry* (1900–1982).

It also appears in the work of *Moses Schönfinkel* (1889–1942),
and *Gottlob Frege* (1848–1925).

Putting currying to work

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

```
sum :: [Int] -> Int
sum xs = foldr (+) 0 xs
```

is equivalent to

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

```
sum :: [Int] -> Int
sum = foldr (+) 0
```

Compare and contrast

```
sum :: [Int] -> Int
sum xs = foldr (+) 0 xs
```

```
sum [1,2,3,4]
=
foldr (+) 0 [1,2,3,4]
```

```
sum :: [Int] -> Int
sum = foldr (+) 0
```

```
sum [1,2,3,4]
=
(foldr (+) 0) [1,2,3,4]
```

Sum, Product, Concat

```
sum      :: [Int] -> Int
sum      = foldr (+) 0
```

```
product  :: [Int] -> Int
product  = foldr (*) 1
```

```
concat   :: [[a]] -> [a]
concat   = foldr (++) []
```