

Informatics 1
Functional Programming Lecture 6

Map, filter, fold

Ohad Kammar
The University of Edinburgh

Part I

Map

Squares

```
> 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

```
> 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

```
> 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

```
> 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

Odds

```
> odds [1,2,3]  
[1,3]
```

```
odds :: [Int] -> [Int]  
odds xs = [ x | x <- xs, odd x]
```

```
odds :: [Int] -> [Int]  
odds [] = []  
odds (x:xs) | odd x = x : odds xs  
            | otherwise = odds xs
```

Digits

```
> 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
```

Odds, revisited

```
> odds [1,2,3]  
[1,3]
```

```
odds :: [Int] -> [Int]  
odds xs = [ x | x <- xs, odd x ]
```

```
odds :: [Int] -> [Int]  
odds [] = []  
odds (x:xs) | odd x = x : odds xs  
            | otherwise = odds xs
```

```
odds :: [Int] -> [Int]  
odds xs = filter odd xs
```

Digits, revisited

```
> 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

```
> sum [1,2,3,4]  
10
```

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

Product

```
> product [1,2,3,4]  
24
```

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

Concatenate

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

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

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

And

```
> and [True, True, True]
```

```
True
```

```
> and [True, False, True]
```

```
False
```

```
and :: [Bool] -> Bool
```

```
and [] = True
```

```
and (x:xs) = x && and xs
```

Or

```
> or [False, False, False]
False
```

```
> or [False, True, False]
True
```

```
or :: [Bool] -> Bool
or []      = False
or (x:xs)  = x || or xs
```

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

```
> 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, And, Or

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

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

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

```
and       :: [Bool] -> Bool
and xs    = foldr (&&) True xs
```

```
or        :: [Bool] -> Bool
or xs     = foldr (||) False 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 Odds

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

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

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

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