

Informatics 1

Functional Programming Lecture 11

Monday 3 November 2008

Map, filter, fold

Philip Wadler

University of Edinburgh

Part I

Risks to the Public from the Use of Computers



No entry for heavy
goods vehicles.
Residential site only



Nid wyf yn y swyddfa
ar hyn o bryd. Anfonwch
unrhyw waith i'w gyfieithu.

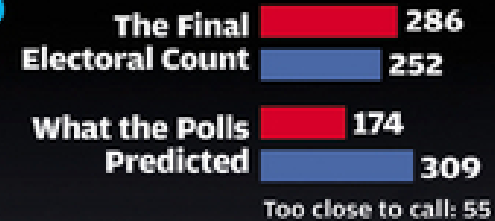


The Welsh reads:

"I am not in the office at the moment. Send any work to be translated."

The Tale of the Exit Polls

On Election Day, exit polls commissioned by six leading news organizations showed Kerry winning handily in four crucial states: Nevada, New Mexico, Florida and Ohio. Since the poll results were beyond the margin of error, Bush's odds of victory were less than one in 450,000. But when the ballots were tallied, the four states "flipped" to Bush, depriving Kerry of fifty-seven electoral votes - and the White House.



NEVADA

Kerry led by: 7.5%
Bush won by: 2.6%

Sproul & Associates, a GOP-paid consultancy, shredded Democratic voter registrations. Electronic voting machines in the state's two most populous, Democratic-leaning counties recorded no presidential vote on 10,000 ballots.



FLORIDA

Kerry led by: 2.6%
Bush won by: 5.0%

Some 58,000 ballots mailed to absentee voters in Broward County were never delivered. Thousands of Floridians were denied the vote simply because they made inconsequential errors on their registration forms.

NEW MEXICO

Kerry led by: 7.0%
Bush won by: 0.8%

In a race decided by fewer than 6,000 votes, New Mexico had the highest rate of ballots - 20,000 - that mysteriously registered no vote for president. Election officials certified 2,087 "phantom votes" - recording more presidential ballots than there were voters.

OHIO

Kerry led by: 8.8%
Bush won by: 2.1%

The GOP illegally targeted black voters, attempting to knock 35,000 citizens off the rolls - almost half in the Democratic stronghold of Cleveland. Unequal distribution of voting machines forced black voters to wait in lines almost three times longer than whites.

Was the 2004 Election Stolen? Robert F. Kennedy Jr., *Rolling Stone*, 1 June 2006.

Part II

Required reading

Required reading

Haskell: The Craft of Functional Programming, Second Edition,
Simon Thompson, Addison-Wesley, 1999.

Thompson, Chapters 1–3 (pp. 1–52): by Mon 29 Sep 2008.

Thompson, Chapters 4–5 (pp. 53–95): by Mon 6 Oct 2008.

Thompson, Chapters 6–7 (pp. 96–134): by Mon 13 Oct 2008.

Thompson, Chapters 8–9 (pp. 135–166): by Mon 20 Oct 2008.

Thompson, Chapters 10–11 (pp. 167–209): by Mon 3 Nov 2008.

Thompson, Chapters 12–14 (pp. 210–241): by Mon 10 Nov 2006.

Thompson and other books available in ITO.

Part III

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 square xs
  where
    square x = x*x
```

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 IV

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 positive xs
  where
    positive 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 : isDigit xs  
              | otherwise = isDigit xs
```

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

Part V

Map and Filter, together

Squares of Positives

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

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

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

```
squarePositives :: [Int] -> [Int]
squarePositives xs = map square (filter positive xs)
  where
    square x      = x*x
    positive x    = x > 0
```

Ords of Digits

```
*Main> ordDigits "a2c3"
```

```
[50,51]
```

```
ordDigits :: [Char] -> [Int]
```

```
ordDigits xs = [ ord x | x <- xs, isDigit x ]
```

```
ordDigits :: [Char] -> [Int]
```

```
ordDigits [] = []
```

```
ordDigits (x:xs) | isDigit x = ord x : ordDigits xs  
                 | otherwise = ordDigits p xs
```

```
ordDigits :: [Char] -> [Int]
```

```
ordDigits xs = map ord (filter isDigit xs)
```

Part VI

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 a [] = a
```

```
foldr f a (x:xs) = f x (foldr f a 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
```

Product, revisited

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

```
24
```

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

```
product [] = 1
```

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

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

```
product xs = foldr (*) 1 xs
```

Concatenate, revisited

```
*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
```

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

Part VII

Fold, generalised

Reverse

```
*Main> reverse [1,2,3]  
[3,2,1]
```

```
*Main> reverse "abc"  
"cba"
```

```
reverse :: [a] -> [a]  
reverse []      = []  
reverse (x:xs) = reverse xs ++ [x]
```


Insertion Sort

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

```
insert :: Int -> [Int] -> [Int]  
insert x [] = []  
insert x (y:ys) | x > y = y : insert x ys  
                | otherwise = x : y : ys
```

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

```
iSort :: [Int] -> [Int]  
iSort [] = []  
iSort (x:xs) = insert x (iSort xs)
```


Foldr, generalized

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

Reverse, revisited

```
*Main> reverse [1,2,3]
[3,2,1]
```

```
*Main> reverse "abc"
"cba"
```

```
reverse :: [a] -> [a]
reverse []      = []
reverse (x:xs) = reverse xs ++ [x]
```

```
reverse :: [a] -> [a]
reverse xs = foldr snoc [] xs
  where
    snoc x xs = xs ++ [x]
```

Insertion Sort, revisited

```
insert :: Int -> [Int] -> [Int]
insert x [] = []
insert x (y:ys) | x > y = y : insert x ys
                | otherwise = x : y : ys
```

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

```
iSort :: [Int] -> [Int]
iSort [] = []
iSort (x:xs) = insert x (iSort xs)
```

```
iSort :: [Int] -> [Int]
iSort xs = foldr insert [] xs
```

takeWhile and dropWhile

```
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
```

```
*Main> takeWhile isLower "goodBye"
"good"
```

```
*Main> dropWhile isLower "goodBye"
"Bye"
```

Insert, revisited

```
insert :: Int -> [Int] -> [Int]
insert x ys
  = takeWhile xGreater ys ++ [x] ++ dropWhile xGreater ys
  where
    xGreater y = x > y
```