Informatics 1

Functional Programming Lecture 7

# Map, filter, fold

Don Sannella

University of Edinburgh

# Part I

# Map

Now we're going to look at some examples of Haskell programs in an attempt to find common patterns.
Then we'll see how to generalise by writing a single Haskell program that has all of the examples as instances.

## Squares

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

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

Squares of the elements in a list, using comprehension

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

... using recursion

## Ords

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

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

Numerical codes of the characters in a list, using comprehension

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

... using recursion

ord :: Char -> Int converts a character to its numerical code, for instance ord 'A' = 65

## 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
```

We can capture that pattern like this - first using comprehension, then the same thing using recursion.
There's a function, f, that we want to apply to every element of a list xs.
f :: a -> b, xs :: [a], result :: [b]
For squares, a and b are both Int. For ords, a is Char and b is Int.

We call this map - it's built into Haskell's standard prelude (same as filter and foldr, coming up)

## 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
```

Now we have THREE ways to define squares: using comprehension, using recursion, using map.
For squares, the function takes x to x*x, defined as a helper function using "where".

## 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]
```

Here's how squares written using map works, using the comprehension definition of map.
The main point is that f is replaced by sqr and then everything works just as before.

## 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]
```

Ditto, for the recursive definition of map.

## 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
```

Here is how ords is written using map.
The function is ord, which is already defined, so we don't need to define it ourselves.

# Part II

# Filter

Another common pattern is extracting all of the elements of a list that have some property.

## 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
```

Finding the positive numbers in a list using comprehension

... using recursion

## 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
```

Finding the characters in a list that are digits ('0' to '9'), using comprehension

... using recursion

## 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
```

Filter takes a PREDICATE p (a function producing a result of type Bool)
which says whether or not an element in the list belongs to the result.

The predicate is used as a guard in both the comprehension and the recursive definitions of filter.

## 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
```

Now we can define positives in a third way, using filter.

## 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
```

Likewise for digits, using the built-in isDigit function.

The predicate can be as complicated as you want. You just need to define it as a function.

## Part III

# Fold

## Sum

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

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

Sum of a list of integers. sum [] = 0, because 0 is the identity for +.

## Product

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

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

Product of a list of integers. product [] = 1, because 1 is the identity for *.

## 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
```

Concatenate a list of lists by appending the head (a list) to the result of concatenating all of the lists in the tail.
concat [] = [] because [] is the identity for ++.

## Foldr

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

Here's the pattern. We take TWO things:
- a binary function f :: a -> a -> a
- a value v :: a (which is often the identity for f)

and return the result of combining the elements using f, with v as the result when we get to the end.
Think of v as the STARTING VALUE for combining the elements using f.

## 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)
```

It might be easier to understand when f is written in infix.

sum [a1, ..., an] = a1 + ... + an + 0
product [a1, ..., an] = a1 * ... * an * 1
concat [xs1, ..., xsn] = xs1 ++ ... ++ xsn ++ []
foldr f v [x1, ..., xn] = x1 `f` ... `f` xn `f` v

# 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.

Here is sum defined using foldr. f is (+), v is 0.

## 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
```

Similarly with product and concat.

## 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
```

Here's how sum works using the recursive definition of sum.

## 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
```

Here's how sum works using the definition of sum in terms of foldr.

foldr means "fold, bracketing to the RIGHT"
sum [x1, x2, ..., xn] = x1 + (x2 + ( ... + (xn + 0) ... ))
foldr f v [x1, x2, ..., xn] = x1 `f` (x2 `f` ( ... `f` (xn `f` v) ... ))

```
As a diagram:           :     -> foldr f v ->     f
                       / \                       / \
                     x1   :                    x1   f
                         / \                       / \
                       x2  ...                   x2  ...
                             \                         \
                              :                         f
                             / \                       / \
                           xn   [ ]                  xn   v
```

If f is associative, it doesn't matter that it brackets to the right.
If f isn't associative, the result is different if you bracket to the left.
There is a function (foldl) that does that. Try to define it - answer in the book.

Also: foldr's type is more general than I've indicated. See if you can figure it out. Answer in the book.
Hint: consider v :: b and xs :: [a]. What type does f need to have?

foldr is called "reduce" in some other functional languages.
Google's Mapreduce, for doing distributed computing in warehouses full of servers, is a combination of map and foldr.
Cf Apache's Hadoop.

## Part IV

# Map, Filter, and Fold

## All together now!

# Sum of Squares of Positives

Definition by combining functions

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

... or using comprehension

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

... or using recursion

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

... or using foldr, map and filter

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

Functions like map, filter and foldr are called HIGHER ORDER FUNCTIONS: they take other functions as arguments.
Functions that don't do this are called FIRST ORDER FUNCTIONS.

Functional programming is about defining functions, but also about (like here) using functions as data.
This gives surprising power!

# Part V

# Currying

If a function takes two arguments, we write

```
f :: t -> s -> v
```

and we apply f by writing

```
f x y
```

Now it's time to explain why.

## How to add two numbers

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

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

Consider the function add. Where are the implicit parentheses?
That is, how does Haskell group the parts of the type and the function definition?

# How to add two numbers

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

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

Here they are.
-> associates to the right
function application associates to the left.
(This matches - think about it!)
add applied to 3, applied to 4

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

add 3 makes sense by itself - it's the function that adds 3 to things
This shows how we can define 2-argument functions in terms of 1-argument functions.

## 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
```

Here's the same thing written in a different way.
The result of applying add to x is now an explicitly-defined function.

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
```

This is not functions taking functions AS ARGUMENTS
but functions returning functions AS RESULTS.

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).

Currying: transforming a 2-argument function into a 1-argument function that produces a 1-argument function.
Uncurrying is the opposite.

## 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
```

foldr (+) 0 is a function taking xs to the result

The point here is not that we can save 4 characters when defining sum.
The point is that we can PARTIALLY APPLY functions.

## 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 (++) []
```

Here is the same thing applied to the earlier definitions of functions using foldr.
This is sometimes called POINTLESS or POINT-FREE style.
(In contrast to POINTED style, referring to definitions in terms of "points" in the space of possible arguments.)