Informatics 1

Functional Programming Lecture 7

Monday 7 October 2013

Map, filter, fold

Don Sannella

University of Edinburgh

Required text and reading

Haskell: The Craft of Functional Programming (Third Edition),
Simon Thompson, Addison-Wesley, 2011.

or

Learn You a Haskell for Great Good!
Miran Lipovača, No Starch Press, 2011.

Reading assignment

Monday 23 September 2013	Thompson: parts of Chap. 1-3 and 5
	Lipovača: parts of intro, Chap. 1-3
Monday 30 September 2013	Thompson: parts of Chap. 4-7 and 17
	Lipovača: rest of Chap. 1 and 3, Chap. 4
Monday 7 October 2013	Thompson: parts of Chap. 10 and 11
	Lipovača: Chap. 5

The assigned reading covers the material very well with plenty of examples.

There will be no lecture notes, just the books. *Get one of them and read it!*

Tutorials—revision tutorials

Every Wednesday starting 2–3pm Wednesday 9th October
Appleton Tower, Computer Lab West (5.05)

Attempt the 2012 class test *in advance*.
Print out and bring your solutions.

Class test

2:10–3:00pm Monday 21 October 2013

George Square Lecture Theatre

Past exams available on website

http://www.inf.ed.ac.uk/teaching/courses/inf1/fp/

Part I

List comprehensions, revisited

Evaluating a list comprehension: generator

```
    [ x*x | x <- [1..3] ]
=
    [ 1*1 ] ++ [ 2*2 ] ++ [ 3*3 ]
=
    [ 1 ]   ++ [ 4 ]   ++ [ 9 ]
=
    [1, 4, 9]
```

Evaluating a list comprehension: generator and filter

```
    [ x*x | x <- [1..3], odd x ]
=
    [ 1*1 | odd 1 ] ++ [ 2*2 | odd 2 ] ++ [ 3*3 | odd 3 ]
=
    [ 1 | True ]     ++ [ 4 | False ]    ++ [ 9 | True ]
=
    [ 1 ]            ++ [ ]              ++ [ 9 ]
=
    [1, 9]
```

Evaluating a list comprehension: two generators

```
    [ (i,j) | i <- [1..3], j <- [i..3] ]
=
    [ (1,j) | j <- [1..3] ] ++
    [ (2,j) | j <- [2..3] ] ++
    [ (3,j) | j <- [3..3] ]
=
    [ (1,1) ] ++ [ (1,2) ] ++ [ (1,3) ] ++
                 [ (2,2) ] ++ [ (2,3) ] ++
                              [ (3,3) ]
=
    [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Another example

```
    [ (i,j) | i <- [1..3], j <- [1..3], i <= j ]
=
    [ (1,j) | j <- [1..3], 1 <= j ] ++
    [ (2,j) | j <- [1..3], 2 <= j ] ++
    [ (3,j) | j <- [1..3], 3 <= j ]
=
    [(1,1)|1<=1] ++ [(1,2)|1<=2] ++ [(1,3)|1<=3] ++
    [(2,1)|2<=1] ++ [(2,2)|2<=2] ++ [(2,3)|2<=3] ++
    [(3,1)|3<=1] ++ [(3,2)|3<=2] ++ [(3,3)|3<=3]
=
    [(1,1)] ++ [(1,2)] ++ [(1,3)] ++
    []      ++ [(2,2)] ++ [(2,3)] ++
    []      ++ []      ++ [(3,3)]
=
    [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Defining list comprehensions

$$q \ ::= \ x \leftarrow l\texttt{,}\ q \ \mid\ b\texttt{,}\ q \ \mid\ \star$$

$$\texttt{[}\ e \ \texttt{|}\ \star\ \texttt{]}$$
$$= \ \texttt{[}\ e\ \texttt{]}$$
$$\texttt{[}\ e \ \texttt{|}\ x \leftarrow \texttt{[}\ l_1\texttt{,}\ \ldots\texttt{,}\ l_n\ \texttt{]}\texttt{,}\ q\ \texttt{]}$$
$$= \ (\texttt{let}\ x = l_1\ \texttt{in}\ \texttt{[}\ e \ \texttt{|}\ q\ \texttt{]})\ \texttt{++} \cdots \texttt{++}\ (\texttt{let}\ x = l_n\ \texttt{in}\ \texttt{[}\ e \ \texttt{|}\ q\ \texttt{]})$$
$$\texttt{[}\ e \ \texttt{|}\ b\texttt{,}\ q\ \texttt{]}$$
$$= \ \texttt{if}\ b\ \texttt{then}\ \texttt{[}\ e \ \texttt{|}\ q\ \texttt{]}\ \texttt{else}\ \texttt{[ ]}$$

Another example, revisited

```
    [ (i,j) | i <- [1..3], j <- [1..3], i <= j, * ]
=
    [ (1,j) | j <- [1..3], 1 <= j, * ] ++
    [ (2,j) | j <- [1..3], 2 <= j, * ] ++
    [ (3,j) | j <- [1..3], 3 <= j, * ]
=
    [(1,1)|1<=1,*] ++ [(1,2)|1<=2,*] ++ [(1,3)|1<=3,*] ++
    [(2,1)|2<=1,*] ++ [(2,2)|2<=2,*] ++ [(2,3)|2<=3,*] ++
    [(3,1)|3<=1,*] ++ [(3,2)|3<=2,*] ++ [(3,3)|3<=3,*]
=
    [(1,1)|*] ++ [(1,2)|*] ++ [(1,3)|*] ++
    []        ++ [(2,2)|*] ++ [(2,3)|*] ++
    []        ++ []        ++ [(3,3)|*]
=
    [(1,1)] ++ [(1,2)] ++ [(1,3)] ++
    []      ++ [(2,2)] ++ [(2,3)] ++
    []      ++ []      ++ [(3,3)]
=
    [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Part II

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 III

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 IV

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)
```

Foldr, with infix notation

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

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 a []        =  a
foldr f a (x:xs)    =  x `f` (foldr f a 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 V

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 VI

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 a []        =  a
foldr f a (x:xs)    =  f x (foldr f a xs)

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

is equivalent to

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