Informatics 1

Functional Programming Lectures 7 and 8

Monday 17 and Tuesday 18 October 2011

Map, filter, fold

Philip Wadler

University of Edinburgh

Class test

2:00–2:50pm Monday 24 October 2011

Appleton Tower, Lecture Theatre 4

Past exams available on website

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

Drop-in labs—longer lab hours

Monday	3:30–4:30pm	Computer Lab West
Tuesday	2–4pm	Computer Lab West
Wednesday	2–4pm	Computer Lab West
Thursday	2–4pm	Computer Lab West
Friday	3:30–4:30pm	Computer Lab North

Computer Lab West and North – Appleton Tower, fifth floor

If you are not getting through the tutorials,
show up in the labs *early* and *often*.

Tutorials—extra tutorial

4–5pm Wednesday 19 October

Appleton Tower 4.12

Attempt the 2010 Class Exam *in advance*.

Print out and bring your solutions.

Required text and reading

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

Reading assignment

Monday 26 September 2011	Chapters 1–3 (pp. 1–66)
Monday 3 October 2011	Chapters 4–7 (pp. 67–176)
Monday 10 October 2011	Chapters 8–9 (pp. 177–212)
Monday 17 October 2011	Chapters 10–12 (pp. 213–286)
Monday 24 October 2011	*Class test*
Monday 31 October 2011	Chapters 13–14 (pp. 287–356)
Monday 7 November 2011	Chapters 15–16 (pp. 357–414)
Monday 14 November 2011	Chapters 17–21 (pp. 415–534)

Phil

Phil's tie

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 : isDigit xs
              | otherwise         =  isDigit 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

Currying

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

  (f 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

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

means the same as

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

and

```
f 3 4
```

means the same as

```
(f 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 (++) []
```

Part VII

Lambda expressions

λ

A failed attempt to simplify

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

The above *cannot* be simplified to the following:

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

A successful attempt to simplify

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

The above *can* be simplified to the following:

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

Lambda calculus

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

The character \ stands for λ, the Greek letter *lambda*.

Logicians write

`\x -> x > 0` as $\lambda x.\, x > 0$

`\x -> x * x` as $\lambda x.\, x \times x$.

Lambda calculus is due to the logician *Alonzo Church* (1903–1995).

Evaluating lambda expressions

```
    (\x -> x > 0) 3
=
    let x = 3 in x > 0
=
    3 > 0
=
    True

   (\x -> x * x) 3
=
   let x = 3 in x * x
=
   3 * 3
=
   9
```

Lambda expressions and currying

```
    (\x -> \y -> x + y) 3 4
=
    ((\x -> (\y -> x + y)) 3) 4
=
    (let x = 3 in \y -> x + y) 4
=
    (\y -> 3 + y) 4
=
    let y = 4 in 3 + y
=
    3 + 4
=
    7
```

Evaluating lambda expressions

The general rule for evaluating lambda expressions is

$$\begin{array}{c} (\lambda x.\, N)\; M \\ = \\ (\texttt{let}\; x = M\; \texttt{in}\; N) \end{array}$$

This is sometimes called the β rule (or beta rule).

Part VIII

Sections

Sections

`(> 0)` is shortand for `(\x -> x > 0)`

`(2 *)` is shortand for `(\x -> 2 * x)`

`(+ 1)` is shortand for `(\x -> x + 1)`

`(2 ^)` is shortand for `(\x -> 2 ^ x)`

`(^ 2)` is shortand for `(\x -> x ^ 2)`

Sections

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

f :: [Int] -> Int
f xs  =  foldr (+) 0 (map (^ 2) (filter (> 0) xs))
```

Part IX

Composition

Composition

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x  =  f (g x)
```

Evaluation composition

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x  =  f (g x)

sqr :: Int -> Int
sqr x  =  x * x

pos :: Int -> Bool
pos x  =  x > 0

   (pos . sqr) 3
=
  pos (sqr 3)
=
  pos 9
=
  True
```

Compare and contrast

```
possqr :: Int -> Bool
possqr x  =  pos (sqr x)

  possqr 3
=
  pos (sqr 3)
=
  pos 9
=
  True
```

```
possqr :: Int -> Bool
possqr =  pos . sqr

  possqr 3
=
  (pos . sqr) 3
=
  pos (sqr 3)
=
  pos 9
=
  True
```

Composition is associative

```
(f . g) . h = f . (g . h)
```

```
    ((f . g) . h) x
=
    (f . g) (h x)
=
    f (g (h x))
=
    f ((g . h) x)
=
    (f . (g . h)) x
```

Thinking functionally

```
f :: [Int] -> Int
f xs  =  foldr (+) 0 (map (^ 2) (filter (> 0) xs))

f :: [Int] -> Int
f  =  foldr (+) 0 . map (^ 2) . filter (> 0)
```

Applying the function

```
f :: [Int] -> Int
f  =  foldr (+) 0 . map (^ 2) . filter (> 0)

   f [1, -2, 3]
=
   (foldr (+) 0 . map (^ 2) . filter (> 0)) [1, -2, 3]
=
   foldr (+) 0 (map (^ 2) (filter (> 0) [1, -2, 3]))
=
   foldr (+) 0 (map (^ 2) [1, 3])
=
   foldr (+) 0 [1, 9]
=
   10
```

Part X

Variables and binding

Variables

```
x = 2
y = x+1
z = x+y*y

*Main> z
11
```

Variables—binding

```
x = 2
y = x+1
z = x+y*y

*Main> z
11
```

Binding occurrence

Bound occurrence

Scope of binding

Variables—binding

```
x = 2
y = x+1
z = x+y*y

*Main> z
11
```

Binding occurrence

Bound occurrence

Scope of binding

Variables—binding

```
x = 2
y = x+1
z = x+y*y

*Main> z
11
```

Binding occurrence

Bound occurrence

Scope of binding

Variables—renaming

```
xavier = 2
yolanda = xavier+1
zeuss = xavier+yolanda*yolanda

*Main> zeuss
11
```

Part XI

Functions and binding

Functions—binding

```
f x = g x (x+1)
g x y = x+y*y

*Main> f 2
11
```

Functions—binding

```
f x = g x (x+1)
g x y = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Functions—binding

```
f x = g x (x+1)
g x y = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

There are two *unrelated* uses of `x`!

Functions—binding

```
f x = g x (x+1)
g x y = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Functions—binding

```
f x = g x (x+1)
g x y = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Functions—binding

```
f x = g x (x+1)
g x y = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Functions—formal and actual parameters

```
f x = g x (x+1)
g x y = x+y*y

*Main> f 2
11
```

Formal parameter

Actual parameter

Functions—formal and actual parameters

```
f x = g x (x+1)
g x y = x+y*y

*Main> f 2
11
```

Formal parameter

Actual parameter

Functions—formal and actual parameters

```
f x = g x (x+1)
g x y = x+y*y

*Main> f 2
11
```

Formal parameter

Actual parameter

Functions—renaming

```
fred xavier = george xavier (xavier+1)
george xerox yolanda = xerox+yolanda*yolanda

*Main> fred 2
11
```

Different uses of `x` renamed to `xavier` and `xerox`.

Part XII

Variables in a where clause and binding

Variables in a where clause

```
f x = z
    where
    y = x+1
    z = x+y*y

*Main> f 2
11
```

Variables in a where clause—binding

```
f x = z
    where
    y = x+1
    z = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Variables in a where clause—binding

```
f x = z
    where
    y = x+1
    z = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Variables in a where clause—binding

```
f x = z
    where
    y = x+1
    z = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Variables in a where clause—binding

```
f x = z
    where
    y = x+1
    z = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Variables in a where clause—hole in scope

```
f x = z
    where
    y = x+1
    z = x+y*y

y = 5

*Main> y
5
```

Binding occurrence

Bound occurrence

Scope of binding

Part XIII

Functions in a where clause and binding

Functions in a where clause

```
f x = g (x+1)
    where
    g y = x+y*y

*Main> f 2
11
```

Functions in a where clause—binding

```
f x = g (x+1)
    where
    g y = x+y*y
```

```
*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Variable `x` is still in scope within `g`!

Functions in a where clause—binding

```
f x = g (x+1)
    where
    g y = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Functions in a where clause—binding

```
f x = g (x+1)
    where
    g y = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Functions in a where clause—binding

```
f x = g (x+1)
    where
    g y = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Functions in a where clause—hole in scope

```
f x = g (x+1)
    where
    g y = x+y*y

g z = z*z*z

*Main> g 2
8
```

Binding occurrence

Bound occurrence

Scope of binding

Functions in a where clause—pathological case

```
f x = f (x+1)
    where
    f y = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Functions in a where clause—pathological case

```
f x = f (x+1)
    where
    f y = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Functions in a where clause—formals and actuals

```
f x = g (x+1)
    where
    g y = x+y*y

*Main> f 2
11
```

Formal parameter

Actual parameter

Functions in a where clause—formals and actuals

```
f x = g (x+1)
    where
    g y = x+y*y

*Main> f 2
11
```

Formal parameter

Actual parameter

Part XIV

Lambda expressions and binding

A wrong attempt to simplify

```
f :: [Int] -> [Int]
f xs  =  map (x * x) (filter (x > 0) xs)
```

This makes no sense—no binding occurrence of variable!

Lambda expressions

```
f :: [Int] -> [Int]
f xs  =
   map (\x -> x * x) (filter (\x -> x > 0) xs)
```

The character \ stands for λ, the Greek letter *lambda*.

Logicians write

`(\x -> x * x)` as $(\lambda x.\, x \times x)$

`(\x -> x > 0)` as $(\lambda x.\, x > 0)$

Lambda expressions—binding

```
f :: [Int] -> [Int]
f xs  =  map (\x -> x*x) (filter (\x -> x > 0) xs)
```

Binding occurrence

Bound occurrence

Scope of binding

Lambda expressions—binding

```
f :: [Int] -> [Int]
f xs  =  map (\x -> x*x) (filter (\x -> x > 0) xs)
```

Binding occurrence

Bound occurrence

Scope of binding

Part XV

Lambda expressions explain binding

Lambda expressions explain binding

A variable binding can be rewritten using a lambda expression and an application:

$$
\begin{aligned}
& (N \; \texttt{where} \; x = M) \\
= & \\
& (\lambda x.\, N)\; M \\
= & \\
& (\texttt{let} \; x = M \; \texttt{in} \; N)
\end{aligned}
$$

A function binding can be written using an application on the left or a lambda expression on the right:

$$
\begin{aligned}
& (M \; \texttt{where} \; f\; x = N) \\
= & \\
& (M \; \texttt{where} \; f = \lambda x.\, N)
\end{aligned}
$$

Lambda expressions and binding constructs

```
    f 2
    where
    f x   =  x+y*y
         where
         y = x+1
=
    f 2
    where
    f   =  \x -> (x+y*y where y = x+1)
=
    f 2
    where
    f   =  \x -> ((\y -> x+y*y) (x+1))
=
    (\f -> f 2) (\x -> ((\y -> x+y*y) (x+1)))
```

Evaluating lambda expressions

```
    (\f -> f 2) (\x -> ((\y -> x+y*y) (x+1)))
=
    (\x -> ((\y -> x+y*y) (x+1))) 2
=
    (\y -> 2+y*y) (2+1)
=
    (\y -> 2+y*y) 3
=
    2+3*3
=
    11
```

Part XVI

Additional material: Lambda expressions and binding, revisited

Lambda expressions—binding

```
(\f -> f 2) (\x -> ((\y -> x+y*y) (x+1)))
```

Binding occurrence

Bound occurrence

Scope of binding

Lambda expressions—binding

```
(\f -> f 2) (\x -> ((\y -> x+y*y) (x+1)))
```

Binding occurrence

Bound occurrence

Scope of binding

Lambda expressions—binding

```
(\f -> f 2) (\x -> ((\y -> x+y*y) (x+1)))
```

Binding occurrence

Bound occurrence

Scope of binding

Lambda expressions—formals and actuals

```
(\f -> f 2) (\x -> ((\y -> x+y*y) (x+1)))
```

Formal parameter

Actual parameter

Lambda expressions—formals and actuals

```
(\x -> ((\y -> x+y*y) (x+1))) 2
```

Formal parameter

Actual parameter

Lambda expressions—formals and actuals

```
(\y -> 2+y*y) (2+1)
```

Formal parameter

Actual parameter

Part XVII

Additional material: Comprehensions and binding

Comprehensions

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

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

Comprehensions—binding

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

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

Binding occurrence

Bound occurrence

Scope of binding

Comprehensions—binding

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

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

Binding occurrence

Bound occurrence

Scope of binding

Comprehensions—pathological case

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

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

Binding occurrence

Bound occurrence

Scope of binding – Note hole in scope!

Squares of Positives—pathological case

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

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

Binding occurrence

Bound occurrence

Scope of binding

List comprehension with two qualifiers

```
f n  =  [ (i,j) | i <- [1..n], j <- [i..n] ]

*Main> f 3
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

List comprehension with two qualifiers—binding

```
f n  =  [ (i,j) | i <- [1..n], j <- [i..n] ]

*Main> f 3
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Binding occurrence

Bound occurrence

Scope of binding

List comprehension with two qualifiers—binding

```
f n  =  [ (i,j) | i <- [1..n], j <- [i..n] ]

*Main> f 3
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Binding occurrence

Bound occurrence

Scope of binding

List comprehension with two qualifiers—binding

```
f n  =  [ (i,j) | i <- [1..n], j <- [i..n] ]

*Main> f 3
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Binding occurrence

Bound occurrence

Scope of binding

Part XVIII

Additional material:

Higher-order functions and binding

Higher-order functions

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

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

Higher order functions—binding

```
f xs    =  map sqr (filter pos xs)
  where
  sqr x   =   x*x
  pos x   =   x > 0

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

Binding occurrence

Bound occurrence

Scope of binding

Higher-order functions—binding

```
f xs   =  map sqr (filter pos xs)
  where
  sqr x   =  x*x
  pos x   =  x > 0

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

Binding occurrence

Bound occurrence

Scope of binding

Higher-order functions—binding

```
f xs   =  map sqr (filter pos xs)
  where
  sqr x   =   x*x
  pos x   =   x > 0

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

Binding occurrence

Bound occurrence

Scope of binding

Higher-order functions—binding

```
f xs   =  map sqr (filter pos xs)
  where
  sqr x   =   x*x
  pos x   =   x > 0

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

Binding occurrence

Bound occurrence

Scope of binding

Higher-order functions—binding

```
f xs   =  map sqr (filter pos xs)
  where
  sqr x   =  x*x
  pos x   =  x > 0

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

Binding occurrence

Bound occurrence

Scope of binding

Higher-order functions—binding

```
f xs   =  map sqr (filter pos xs)
  where
  sqr x   =   x*x
  pos x   =   x > 0

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

Binding occurrence

Bound occurrence

Scope of binding

Higher-order functions—binding

```
f xs   =  map sqr (filter pos xs)
  where
  sqr x   =  x*x
  pos x   =  x > 0

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

Binding occurrence—not shown (in standard prelude)

Bound occurrence

Scope of binding

Higher-order functions—binding

```
f xs   =  map sqr (filter pos xs)
  where
  sqr x   =   x*x
  pos x   =   x > 0

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

Binding occurrence—not shown (in standard prelude)

Bound occurrence

Scope of binding