Informatics 1

Functional Programming Lectures 7 and 8

Monday 19 and Tuesday 20 October 2009

Map, filter, fold

Philip Wadler

University of Edinburgh

Required text and reading

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

Reading assignments:

Ch. 1–3 (pp. 1–52), by Fri 25 Sep 2009.
Ch. 4–5 & 7 (pp. 53–95, 115–134), Mon 5 Oct 2009.
Ch. 6 & 8 (pp. 96–114, 135–148), by Mon 12 Oct 2009.
Ch. 9–11 (pp. 152–209), Mon 19 Oct 2009.
(Class test) Mon 26 Oct 2009.
Ch. 12–14 (pp. 210–279), Mon 2 Nov 2009.
Ch. 15–16 (pp. 280–336), Mon 9 Nov 2009.
Ch. 17–20 (pp. 337–441), Mon 16 Nov 2009.
(Mock exam) Mon 23 Nov 2009.
(Last week of lectures) Mon 30 Nov 2009.

Part I

Currying

How to add two numbers

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

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

Currying

```
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  =  f
  where
  f y  =  x + y

  (add 3) 4
=
  f 4
    where
    f y = 3 + y
=
  3 + 4
=
  7
```

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).

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 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 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] -> [Int]
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 IV

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
```

Converting Digits to Integers

```
*Main> digitsToInts "a2c3"
[2,3]

digitsToInts :: [Char] -> [Int]
digitsToInts xs  =  [ digitToInt x | x <- xs, isDigit x ]

digitsToInts :: [Char] -> [Int]
digitsToInts []                        =  []
digitsToInts (x:xs) | isDigit x   =  ord x : digitsToInts xs
                    | otherwise   =  digitsToInts p xs

digitsToInts :: [Char] -> [Int]
digitsToInts xs  =  map ord (filter isDigit xs)
```

Part V

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 add 0 xs
  where
  add x y   =  x + y
```

How sum works

```
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 add 0 xs
  where
  add x y  =  x + y

  sum [1,2,3,4]
=
  foldr add 0 [1,2,3,4]
=
  add 1 (add 2 (add 3 (add 4 0)))
=
  1 + (2 + (3 + (4 + 0)))
=
  10
```

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 add 0 xs
  where
  add x y  =  x + y
```

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 add 0
  where
  add x y  =  x + y
```

Compare and contrast

```
sum :: [Int] -> Int
sum xs  =  foldr add 0 xs
  where
  add x y  =  x + y

  sum [1,2,3,4]
=
  foldr add 0 [1,2,3,4]
```

```
sum :: [Int] -> Int
sum =  foldr add 0
  where
  add x y  =  x + y

  sum [1,2,3,4]
=
  foldr add 0 [1,2,3,4]
```

Sum, Product, Concat

```
sum  :: [Int] -> Int
sum  =  foldr add 0
  where
  add x y  =  x + y

product  :: [Int] -> Int
product  =  foldr times 1
  where
  times x y  =  x * y

concat  :: [[a]] -> [a]
concat  =  foldr append []
  where
  append xs ys  =  xs ++ ys
```

Part VI

Map, Filter, and Fold

All together now!

Sum of Squares of Positives

```
*Main> f [1,-2,3]
10

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 add 0 (map square (filter positive xs))
  where
  add x y      =  x + y
  square x     =  x * x
  positive x   =  x > 0
```

Part VII

Lambda expressions

A failed attempt to simplify

```
f :: [Int] -> [Int]
f xs  =  foldr add 0 (map square (filter positive xs))
  where
  add x y      =  x + y
  square x     =  x * y
  positive x   =  x > 0
```

The above *cannot* be simplified to the following:

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

A successful attempt to simplify

```
f :: [Int] -> [Int]
f xs  =  foldr add 0 (map square (filter positive xs))
  where
  add x y      =  x + y
  square x     =  x * x
  positive x   =  x > 0
```

The above *can* be simplified to the following:

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

Lambda calculus

```
f :: [Int] -> [Int]
f xs  =  foldr (\x -> \y -> x + y) 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	$\lambda x.\, x \times x$
`\x -> \y -> x + y`	as	$\lambda x.\, \lambda y.\, x + y$.

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

Evaluating lambda expressions

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

   (\x -> x * x) 3
=
   3 * 3
=
   9

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

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)`

`(+)` is shorthand for `(\x -> \y -> x + y)`

`(*)` is shorthand for `(\x -> \y -> x * y)`

`(++)` is shorthand for `(\xs -> \ys -> xs ++ ys)`

Sections

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

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

Sections

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

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

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

Lambda the Ultimate!

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)

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

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

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

Thinking functionally

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

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

Part X

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)

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

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

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

Thinking functionally

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

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

Part XI

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 XII

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 XIII

Variables in a where clause

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 XIV

Functions in a where clause

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 XV

Comprehensions

Comprehensions

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

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

Comprehensions—binding

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

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

Binding occurrence

Bound occurrence

Scope of binding

Comprehensions—binding

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

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

Binding occurrence

Bound occurrence

Scope of binding

Comprehensions—pathological case

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

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

Binding occurrence

Bound occurrence

Scope of binding – Note hole in scope!

Squares of Positives—pathological case

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

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

Binding occurrence

Bound occurrence

Scope of binding

Part XVI

Higher order functions

Higher-order functions

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

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

Higher order functions—binding

```
squarePositives xs  =  map square (filter positive xs)
  where
  square x     =  x*x
  positive x   =  x > 0

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

Binding occurrence

Bound occurrence

Scope of binding

Higher-order functions—binding

```
squarePositives xs  =  map square (filter positive xs)
  where
  square x     =  x*x
  positive x   =  x > 0

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

Binding occurrence

Bound occurrence

Scope of binding

Higher-order functions—binding

```
squarePositives xs  =  map square (filter positive xs)
  where
  square x     =  x*x
  positive x   =  x > 0

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

Binding occurrence

Bound occurrence

Scope of binding

Higher-order functions—binding

```
squarePositives xs  =  map square (filter positive xs)
  where
  square x     =  x*x
  positive x   =  x > 0

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

Binding occurrence

Bound occurrence

Scope of binding

Higher-order functions—binding

```
squarePositives xs  =  map square (filter positive xs)
  where
  square x     =  x*x
  positive x   =  x > 0

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

Binding occurrence

Bound occurrence

Scope of binding

Higher-order functions—binding

```
squarePositives xs  =  map square (filter positive xs)
  where
  square x    =  x*x
  positive x  =  x > 0

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

Binding occurrence

Bound occurrence

Scope of binding

Higher-order functions—binding

```
squarePositives xs  =  map square (filter positive xs)
  where
  square x     =  x*x
  positive x   =  x > 0

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

Binding occurrence—not shown (in standard prelude)

Bound occurrence

Scope of binding

Higher-order functions—binding

```
squarePositives xs  =  map square (filter positive xs)
  where
  square x     =  x*x
  positive x   =  x > 0

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

Binding occurrence—not shown (in standard prelude)

Bound occurrence

Scope of binding

Part XVII

Lambda expressions

A wrong attempt to simplify

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

This makes no sense—no binding occurrence of variable!

Lambda expressions

```
squarePositives :: [Int] -> [Int]
squarePositives 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

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

Binding occurrence

Bound occurrence

Scope of binding

Lambda expressions—binding

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

Binding occurrence

Bound occurrence

Scope of binding

Part XVIII

Lambda expressions and binding constructs

Lambda expressions and binding constructs

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

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

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

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

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
```

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 XIX

List comprehensions with two qualifiers

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

Evaluating a list comprehension

```
    [ (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

$$
\begin{array}{lcl}
[\,e \mid x \leftarrow l,\ q\,] & = & \texttt{concat}\,(\texttt{map}\,(\lambda x.\,[\,e \mid q\,])\,l) \\
[\,e \mid b,\ q\,] & = & \texttt{if}\,b\,\texttt{then}\,[\,e \mid q\,]\,\texttt{else}\,[\,] \\
[\,e \mid x \leftarrow l\,] & = & \texttt{map}\,(\lambda x.\,e)\,l) \\
[\,e \mid b\,] & = & \texttt{if}\,b\,\texttt{then}\,[\,e\,]\,\texttt{else}\,[\,]
\end{array}
$$

Example 1

```
      [ x*x | x <- xs ]
=
     map (\x -> x*x) xs
```

Example 2

```
    [ x*x | x <- xs, x > 0 ]
=
    concat
      (map
        (\x -> [ x*x | x > 0])
      xs)
=
    concat
      (map
        (\x -> if x > 0 then [x*x] else [])
      xs)
```

Example 3

```
      [ (i,j) | i <- [1..n], j <- [i..n] ]
   =
      concat
        (map
          (\i -> [ (i,j) | j <- [i..n] ])
        [1..n])
   =
      concat
        (map
          (\i -> map (\j -> (i,j)) [i..n])
        [1..n])
```

Example 4

```
    [ (i,j) | i <- [1..n], j <- [1..n], i < j ]
=
    concat
      (map
        (\i -> [ (i,j) | j <- [1..n], i < j ])
      [1..n])
=
    concat
      (map
        (\i ->
          concat
            (map (\j -> [ (i,j) | i < j]) [1..n]))
      [1..n])
=
    concat
      (map
        (\i ->
          concat
            (map
              (\j -> if i < j then [(i,j)] else [])
            [1..n]))
      [1..n])
```