Programação Funcional — BCC222

Aulas 7,8

Map, filter, fold

Lucília Camarão de Figueiredo
Departamento de Ciência da Computação
Universidade Federal de Ouro Preto

Parte I

Currying

Como somar dois números

```
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
```

Uma função com dois argumentos
é o mesmo que
uma função do primeiro argumento
que retorna
uma função do segundo argumento.

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
```

O nome dado a essa idéia é uma homenagem a *Haskell Curry* (1900–1982).
Essa idéia também aparece no trabalho de *Moses Schönfinkel* (1889–1942),
e de *Gottlob Frege* (1848–1925).

Parte 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, de novo

```
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, de novo

```
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
```

Parte III

Filter

Positivos

```
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
```

Digitos

```
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
```

Positivos, de novo

```
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
```

Digitos, de novo

```
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

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

Parte IV

Map e Filter juntos

Quadrados de Positivos

```
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
```

Convertendo Digitos para 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 = digitToInt x :  digitsToInt
                    | otherwise = digitsToInts p xs

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

Parte V

Foldr

Soma

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

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

Produto

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

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

Concatenação

```
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, de novo

```
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
```

Como sum funciona

```
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
```

Usando currying

```
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
```

é equivalente a

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

Compare

```
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]
```

Soma, Produto, Concatenação

```
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
```

Parte VI

Map, Filter e Foldr

Todos juntos!

Soma dos Quadrados de Positivos

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

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

f ::  [Int] -> Int
f [] = 0
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
```

Parte VII

Expressões Lambda

Uma tentativa mal sucedida de simplificação

```
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
```

A definição acima não pode ser simplificada para:

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

Uma tentativa correta de simplificação

```
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
```

A definição acima pode ser simplificada para:

```
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))
```

O caractere \ é usado em lugar de λ, a letra grega *lambda*.

O lógicos escrevem

`\x -> x > 0`	como	$\lambda x.\, x > 0$
`\x -> x * x`	como	$\lambda x.\, x * x$
`\x -> \y -> x + y`	como	$\lambda x.\, \lambda y. x + y$

Lambda calculus foi desenvolvido pelo lógico *Alonzo Church* (1903–1995).

Avaliando expressões lambda

```
  (\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
```

Parte VII

Seções

Seções

`(> 0)`	é uma abreviação para	`(\x -> x > 0)`
`(2 *)`	é uma abreviação para	`(\x -> 2 * x)`
`(+ 1)`	é uma abreviação para	`(\x -> x + 1)`
`(2 ^)`	é uma abreviação para	`(\x -> 2 ^ x)`
`(^ 2)`	é uma abreviação para	`(\x -> x ^ 2)`
`(+)`	é uma abreviação para	`(\x -> \y -> x + y)`
`(*)`	é uma abreviação para	`(\x -> \y -> x * y)`
`(++)`	é uma abreviação para	`(\xs -> \ys -> xs ++ ys)`

Seções

```
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))
```

Seções

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

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

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

Parte IX

Composição

Composição

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

Avaliação de Composição

```
(.)  ::  (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

```
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
```

Pense funcionalmente

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

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

Parte X

Variáveis e *binding*

Variáveis

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

Main> z
11
```

Variáveis — *binding*

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

Main> z
11
```

Binding occurrence
Bound occurrence
Scope of binding

Variáveis — *binding*

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

Main> z
11
```

Binding occurrence

Bound occurrence

Scope of binding

Variáveis — *binding*

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

Main> z
11
```

Binding occurrence
Bound occurrence
Scope of binding

Variáveis — renomeação

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

Main> zeuss
11
```

Parte XI

Funções e *binding*

Funções — *binding*

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

Main> f 2
11
```

Funções — *binding*

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

Main> f 2
11
```

Binding occurrence
Bound occurrence
Scope of binding

Funções — *binding*

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

Main> f 2
11
```

Binding occurrence
Bound occurrence
Scope of binding

Existem dois usos de `x` *não relacionados!*

Funções — *binding*

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

Main> f 2
11
```

Binding occurrence
Bound occurrence
Scope of binding

Funções — *binding*

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

Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Funções — *binding*

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

Main> f 2
11
```

Binding occurrence
Bound occurrence
Scope of binding

Funções — parametros formais e parametros reais

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

Main> f 2
11
```

Parametro formal

Parametro real (ou argumento)

Funções — parametros formais e parametros reais

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

Main> f 2
11
```

Parametro formal

Parametro real (ou argumento)

Funções — parametros formais e parametros reais

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

Main> f 2
11
```

Parametro formal

Parametro real (ou argumento)

Funções — renomeação

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

```
Main> fred 2
11
```

Diferentes usos de `x` renomeados para `xavier` e `xerox`.

Parte XII

Variáveis em cláusulas where

Variáveis em uma cláusula where

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

Main> f 2
11
```

Variáveis em uma cláusula where

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

Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Variáveis em uma cláusula where

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

Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Variáveis em uma cláusula where

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

Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Variáveis em uma cláusula where

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

Main> f 2
11
```

Binding occurrence
Bound occurrence
Scope of binding

Variáveis em uma cláusula where

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

 y = 5

Main> y
5
```

Binding occurrence
Bound occurrence
Scope of binding

Parte XIII

Funções em cláusulas **where**

Funções em uma cláusula where

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

Main> f 2
11
```

Funções em uma cláusula where — *binding*

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

Main> f 2
11
```

Binding occurrence
Bound occurrence
Scope of binding

Variável `x` ainda está no escopo no corpo de `g`!

Funções em uma cláusula where — *binding*

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

Main> f 2
11
```

Binding occurrence
Bound occurrence
Scope of binding

Funções em uma cláusula where — *binding*

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

```
Main> f 2
11
```

Binding occurrence
Bound occurrence
Scope of binding

Funções em uma cláusula where — *binding*

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

Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Funções em uma cláusula where — escopo global

```
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

Funções em uma cláusula where — caso patológico

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

Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Funções em uma cláusula where — caso patológico

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

Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Funções em uma cláusula where — parametros formais e reais

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

Main> f 2
11
```

Parametro formal

Parametro real (ou argumento)

Funções em uma cláusula where — parametros formais e reais

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

Main> f 2
11
```

Parametro formal

Parametro real (ou argumento)

Parte XIV

List Comprehensions

List Comprehensions

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

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

List Comprehensions

```
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

List Comprehensions

```
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

List Comprehensions — caso patológico

```
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

List Comprehensions — caso patológico

```
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 O escopo não é toda a expressão!

Parte XV

Funções de ordem superior

Funções de ordem superior

```
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]
```

Funções de ordem superior — *binding*

```
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]
```

Binding occurrence

Bound occurrence

Scope of binding

Funções de ordem superior — *binding*

```
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]
```

Binding occurrence

Bound occurrence

Scope of binding

Funções de ordem superior — *binding*

```
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]
```

Binding occurrence

Bound occurrence

Scope of binding

Funções de ordem superior — *binding*

```
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]
```

Binding occurrence
Bound occurrence
Scope of binding

Funções de ordem superior — *binding*

```
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]
```

Binding occurrence

Bound occurrence

Scope of binding

Funções de ordem superior — *binding*

```
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]
```

Binding occurrence
Bound occurrence
Scope of binding

Funções de ordem superior — *binding*

```
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]
```

Binding occurrence —- na biblioteca padrão Prelude.hs

Bound occurrence

Scope of binding

Funções de ordem superior — *binding*

```
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]
```

Binding occurrence —- na biblioteca padrão Prelude.hs

Bound occurrence

Scope of binding

Parte XV

Expressões Lambda

Uma tentativa errada de simplificação

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

Isso não faz sentido — as ocorrências da variável não são ligadas!

Expressões lambda

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

O caractere \ é usado em lugar de λ, a letra Grega lambda.

Lógicos escrevem:

`(\x -> x * x)`	como	$(\lambda x.\, x \times x)$
`(\x -> x > 0)`	como	$(\lambda x.\, x > 0)$

Expressões Lambda — *binding*

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

Binding occurrence

Bound occurrence

Scope of binding

Expressões Lambda — *binding*

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

Binding occurrence

Bound occurrence

Scope of binding

Parte XVI

Expressões Lambda e construções de *binding*

Expressões Lambda e construções de *binding*

Uma *ligação* (*binding*) pode ser reescrita usando lambda abstração e aplicação:

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

Uma ligação de função pode ser escrita usando uma aplicação à esquerda ou uma lambda abstração à direita:

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

Expressões Lambda e construções de *binding*

```
  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)))
```

Avaliando expressões Lambda

```
  (\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
```

Expressões Lambda — *binding*

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

Binding occurrence

Bound occurrence

Scope of binding

Expressões Lambda — *binding*

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

Binding occurrence

Bound occurrence

Scope of binding

Expressões Lambda — *binding*

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

Binding occurrence

Bound occurrence

Scope of binding

Expressões Lambda — parametro formal e parametro real

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

Parametro formal

Parametro real (ou argumento)

Expressões Lambda — parametro formal e parametro real

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

Parametro formal

Parametro real (ou argumento)

Expressões Lambda — parametro formal e parametro real

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

Parametro formal

Parametro real (ou argumento)

Parte XVIII

List comprehensions, de novo

List comprehensions com dois qualificadores

```
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 comprehensions com dois qualificadores — *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 comprehensions com dois qualificadores — *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 comprehensions com dois qualificadores — *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

Avaliando list comprehensions

```
  [ (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)]
```

Outro exemplo

```
[ (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)]
```

Definindo list comprehensions

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

Exemplo 1

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

Exemplo 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)
```

Exemplo 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])
```

Exemplo 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])
```