Programação Funcional — BCC222

Aulas 11,12

Tipos de Dados Algébricos

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

Parte I

Revisão

Diferenças — quatro estilos

List comprehension

```
differences x ys = [ x-y | y <- ys, x>y ]
```

Funções de ordem superior, definições locais

```
differences x ys = map sub (filter gtr ys)
  where sub y = x-y
        gtr y = x>y
```

Funções de ordem superior, expressões lambda

```
differences x ys = map (\y -> x-y) (filter (\y -> x>y) ys)
```

Funções de ordem superior, seções

```
differences x ys = map (x-) (filter (x>) ys)
```

List comprehension com dois qualificadiores

```
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 com dois qualificadiores — 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 com dois qualificadiores — 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 com dois qualificadiores — 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

Definindo list comprehensions

$$
\begin{aligned}
[e \mid x \leftarrow [l_1 \cdots l_n], q] &= [e_x^{l_1} \mid q_x^{l_1}] \;\; ++ \;\; \ldots \;\; ++ \;\; [e_x^{l_n} \mid q_x^{l_n}] \\
[e \mid \texttt{True}, q] &= [e \mid q] \\
[e \mid \texttt{False}, q] &= [\,] \\
[e \mid \bullet] &= [\,e\,]
\end{aligned}
$$

Avaliando list comprehension

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

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)|True]
  =
    [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Traduzindo list comprehension

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

Parte II

Tipos de Dados Algébricos

Tudo é tipo de dado algébrico

```
data Bool = False | True

data Season = Winter | Spring | Summer | Fall

data Shape = Circle Float | Rectangle Float Float

data Exp = Lit Int | Add Exp Exp | Mul Exp Exp

data List a = Nil | Cons a (List a)

data Tree a = Empty | Leaf a | Branch (Tree a) (Tree a)

data Maybe a = Nothing | Just a

data Pair a b = Pair a b

data Sum a b = Left a | Right b

data Nat = Zero | Succ Nat
```

Parte III

Tipo Boolean

Boolean

```
data Bool = False | True

not ::  Bool -> Bool
not False = True
not True  = False

(&&) ::  Bool -> Bool -> Bool
False && q = False
True  && q = q

(||) ::  Bool -> Bool -> Bool
False || q = q
True  || q = True
```

Boolean — eq and show

```
eqBool ::  Bool -> Bool -> Bool
eqBool False False = True
eqBool False True  = False
eqBool True  False = False
eqBool True  True  = True

showBool ::  Bool -> String
showBool False = "False"
showBool True  = "True"
```

Parte IV

Season

Season

```
data Season = Winter | Spring | Summer | Fall

next ::  Season -> Season
next Winter = Spring
next Spring = Summer
next Summer = Fall
next Fall   = Winter
```

Season — eq e show

```
eqSeason ::  Season -> Season -> Bool
eqSeason Winter Winter = True
eqSeason Spring Spring = True
eqSeason Summer Summer = True
eqSeason Fall   Fall   = True
eqSeason x      y      = False

showSeason ::  Season -> String
showSeason Winter  = "Winter"
showSeason Spring  = "Spring"
showSeason Summer  = "Summer"
showSeason Fall    = "Fall"
```

Season e Integer

```
data Season = Winter | Spring | Summer | Fall

toInt ::  Season -> Int
toInt Winter = 0
toInt Spring = 1
toInt Summer = 2
toInt Fall   = 3

fromInt ::  Int -> Season
fromInt 0 = Winter
fromInt 1 = Spring
fromInt 2 = Summer
fromInt 3 = Fall

next ::  Season -> Season
next x = fromInt ((toInt x + 1) `mod` 4)

eqSeason ::  Season -> Season -> Bool
eqSeason x y = (toInt x == toInt y)
```

Parte V

Shape

Shape

```
type Radius = Float
type Width  = Float
type Height = Float

data Shape  = Circle Radius
            | Rect Width Height

area ::  Shape -> Float
area (Circle r) = pi * r^2
area (Rect w h) = w * h
```

Shape — eq e show

```
eqShape ::  Shape -> Shape -> Bool
eqShape (Circle r) (Circle r')  = (r == r')
eqShape (Rect w h) (Rect w' h') = (w == w') && (h == h')
eqShape x          y            = False

showShape ::  Shape -> String
showShape (Circle r) = "Circle " ++ showF r
showShape (Rect w h) = "Rect " ++ showF w ++ " " ++ showF h

showF ::  Float -> String
showF x | x >= 0 = show x
        | otherwise = "(" ++ show x ++ ")"
```

Parte VI

Listas

Listas

Com nomes

```
data List a = Nil
            | Cons a (List a)

append ::  List a -> List a -> List a
append Nil          ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Listas

Com nomes

```
data List a = Nil
            | Cons a (List a)

append ::  List a -> List a -> List a
append Nil          ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Com notação built-in

```
data [a] = []
         | a :  [a]

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

Parte VII

Árvore de Expressões

Árvore de expressões

```
data Exp = Lit Int
         | Add Exp Exp
         | Mul Exp Exp

evalExp ::  Exp -> Int
evalExp (Lit n)   = n
evalExp (Add e f) = evalExp e + evalExp f
evalExp (Mul e f) = evalExp e * evalExp f

showExp ::  Exp -> String
showExp (Lit n)   = show n
showExp (Add e f) = par (showExp e ++ "+" ++ showExp f)
showExp (Mul e f) = par (showExp e ++ "*" ++ showExp f)

par ::  String -> String
par s = "(" ++ s ++ ")"
```

Árvore de expressões

```
e0, e1 ::  Exp
e0 = Add (Lit 2) (Mul (Lit 3) (Lit 3))
e1 = Mul (Add (Lit 2) (Lit 3)) (Lit 3)

Main> showExp e0
"(2+(3*3))"

Main> evalExp e0
11

Main> showExp e1
"((2+3)*3)"

Main> evalExp e1
15
```

Árvore de expressões — notação infixada

```
data Exp = Lit Int
         | Exp `Add` Exp
         | Exp `Mul` Exp

evalExp ::  Exp -> Int
evalExp (Lit n)      = n
evalExp (e `Add` f) = evalExp e + evalExp f
evalExp (e `Mul` f) = evalExp e * evalExp f

showExp ::  Exp -> String
showExp (Lit n)      = show n
showExp (e `Add` f) = par (showExp e ++ "+" ++ showExp f)
showExp (e `Mul` f) = par (showExp e ++ "*" ++ showExp f)

par ::  String -> String
par s = "(" ++ s ++ ")"
```

Árvore de expressões — notação infixada

```
e0, e1 ::  Exp
e0 = Lit 2 `Add` (Lit 3 `Mul` Lit 3)
e1 = (Lit 2 `Add` Lit 3) `Mul` Lit 3

Main> showExp e0
"(2+(3*3))"

Main> evalExp e0
11

Main> showExp e1
"((2+3)*3)"

Main> evalExp e1
15
```

Árvore de expressões — simbolos

```
data Exp = Lit Int
         | Exp :+:  Exp
         | Exp :*:  Exp

evalExp ::  Exp -> Int
evalExp (Lit n)     = n
evalExp (e :+:  f) = evalExp e + evalExp f
evalExp (e :*:  f) = evalExp e * evalExp f

showExp ::  Exp -> String
showExp (Lit n)     = show n
showExp (e :+:  f) = par (showExp e ++ "+" ++ showExp f)
showExp (e :*:  f) = par (showExp e ++ "*" ++ showExp f)

par ::  String -> String
par s = "(" ++ s ++ ")"
```

Árvore de expressões — simbolos

```
e0, e1 ::  Exp
e0 = Lit 2 :+:  (Lit 3 :*:  Lit 3)
e1 = (Lit 2 :+:  Lit 3) :*:  Lit 3

Main> showExp e0
"(2+(3*3))"

Main> evalExp e0 11 Main> showExp e1
"((2+3)*3)"

Main> evalExp e1
15
```

Parte VIII

Proposições

Proposições

```
type Name = String
data Prop = Var Name
          | F
          | T
          | Not Prop
          | Prop :|:  Prop
          | Prop :&:  Prop
          deriving (Eq, Ord)

type Names = [Name]
type Env = [(Name,Bool)]
```

Proposições — show

```
showProp ::  Prop -> String
showProp (Var x)     = x
showProp (F)         = "F"
showProp (T)         = "T"
showProp (Not p)     = par ("~" ++ showProp p)
showProp (p :|:  q) = par (showProp p ++ "|" ++ showProp q)
showProp (p :&:  q) = par (showProp p ++ "&" ++ showProp q)

par ::  String -> String
par s = "(" ++ s ++ ")"
```

Nomes em uma proposição

```
names ::  Prop -> Names
names (Var x)     = [x]
names (F)         = []
names (T)         = []
names (Not p)     = names p
names (p :|:  q) = nub (names p ++ names q)
names (p :&:  q) = nub (names p ++ names q)
```

Avaliando uma proposição

```
eval ::  Env -> Prop -> Bool
eval e (Var x)     = lookUp e x
eval e (F)         = False
eval e (T)         = True
eval e (Not p)     = not (eval e p)
eval e (p :|:  q) = eval e p || eval e q
eval e (p :&:  q) = eval e p && eval e q

lookUp ::  Eq a => [(a,b)] -> a -> b
lookUp xys x = the [ y | (x',y) <- xys, x == x' ]
   where the [x] = x
```

Proposições

```
p0 ::  Prop
p0 = (Var "a" :&:  Var "b") :|:
     (Not (Var "a") :&:  Not (Var "b"))

env0 ::  Env
env0 = [("a",False), ("b",False)]

Main> showProp p0
(a & b) | (~a & ~b)

Main> names p0
["a","b"]

Main> eval env0 p0
True

Main> lookUp env0 "a"
False
```

Todos os possíveis *environments*

```
envs ::  Names -> [Env]
envs []     = [[]]
envs (x:xs) = [ (x,False):e | e <- envs xs ] ++
              [ (x,True ):e | e <- envs xs ]
```

Alternative

```
envs ::  Names -> [Env]
envs []     = [[]]
envs (x:xs) = [ (x,b):e | b <- bs, e <- envs xs ]
   where bs = [False,True]
```

Todos os possíveis *environments*

```
  envs []
= [[]]

  envs ["b"]
= [("b",False):[]] ++ [("b",True ):[]]
= [[("b",False)],
   [("b",True )]]

  envs ["a","b"]
= [("a",False):e | e <- envs ["b"] ] ++
   [("a",True ):e | e <- envs ["b"] ]
= [("a",False):[("b",False)],("a",False):[("b",True )]] ++
   [("a",True ):[("b",False)],("a",True ):[("b",True )]]
= [[("a",False),("b",False)],
   [("a",False),("b",True )],
   [("a",True ),("b",False)],
   [("a",True ),("b",True )]]
```

Proposição satisfazível

```
satisfiable ::  Prop -> Bool
satisfiable p = or [ eval e p | e <- envs (names p) ]
```

Proposições

```
p0 ::  Prop
p0 = (Var "a":&:  Var "b") :|:
     (Not (Var "a") :&:  Not (Var "b"))

Main> envs (names p0)
[[("a",False),("b",False)],
[("a",False),("b",True)],
[("a",True ),("b",False)],
[("a",True ),("b",True )]]

Main> [ eval e p0 | e <- envs (names p0) ]
[True,
 False,
 False,
 True]

Main> satisfiable p0
True
```

Outro exemplo: todas as subslistas de uma lista

```
subs ::  [a] -> [[a]]
subs []     = [[]]
subs (x:xs) = subs xs ++ [ x:ys | ys <- subs xs ]
```

Outro exemplo: todas as subslistas de uma lista

```
  subs []
= [[]]

  subs ["b"]
= subs [] ++ ["b":ys | ys <- subs []]
= [[]] ++ ["b":[]]
= [[], ["b"]]

  subs ["a","b"]
= subs ["b"] ++ ["a":ys | ys <- subs ["b"]]
= [[], ["b"]] ++ ["a":[], "a":["b"]]
= [[], ["b"], ["a"], ["a","b"]]
```