Programação Funcional — BCC222

Aulas 5,6

Mais sobre recursão

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

Parte I

Booleanos e Caracteres

Operadores Booleanos

```
not ::  Bool -> Bool
(&&), (||) ::  Bool -> Bool -> Bool

not False = True
not True  = False

False && False = False
False && True  = False
True  && False = False
True  && True  = True

False || False = False
False || True  = True
True  || False = True
True  || True  = True
```

Operadores Booleanos

O operador `&&` poderia também ser definido do seguinte modo:

```
(&&) ::  Bool -> Bool -> Bool

True  && True  = True
_     && _     = False
```

Operadores Booleanos

O operador `&&` poderia também ser definido do seguinte modo:

```
(&&) ::  Bool -> Bool -> Bool

True  && True  = True
_     && _     = False
```

Entretanto, a seguinte definição é ainda mais eficiente:

```
False && _     = False
True  && b     = b
```

Operadores Booleanos

O operador `&&` poderia também ser definido do seguinte modo:

```
(&&) ::  Bool -> Bool -> Bool

True  && True  = True
_     && _     = False
```

Entretanto, a seguinte definição é ainda mais eficiente:

```
False && _     = False
True  && b     = b
```

De maneira análoga, o operador `||` é definido como:

```
True  || _     = True
False || b     = b
```

Definindo operações sobre caracteres

```
isLower ::  Char -> Bool
isLower x = 'a' <= x && x <= 'z'

isUpper ::  Char -> Bool
isUpper x = 'A' <= x && x <= 'Z'

isDigit ::  Char -> Bool
isDigit x = '0' <= x && x <= '9'

isAlpha ::  Char -> Bool
isAlpha x = isLower x || isUpper x
```

Definindo operações sobre caracteres

```
digitToInt ::  Char -> Int
digitToInt c | isDigit c = ord c - ord '0'

intToDigit ::  Int -> Char
intToDigit d | 0 <= d && d <= 9 = chr (ord '0' + d)

toLower ::  Char -> Char
toLower c | isUpper c = chr (ord c - ord 'A' + ord 'a')
          | otherwise = c

toUpper ::  Char -> Char
toUpper c | isLower c = chr (ord c - ord 'a' + ord 'A')
          | otherwise = c
```

Parte II

Condicionais and Associatividade

Equações Condicionais

```
max ::  Int -> Int -> Int
max x y | x >= y    = x
        | y >= x    = y

max3 ::  Int -> Int -> Int -> Int
max3 x y z | x >= y && x >= z = x
           | y >= x && y >= z = y
           | z >= x && z >= y = z
```

Equações Condicionais

```
max ::  Int -> Int -> Int
max x y | x >= y    = x
        | otherwise = y

max3 ::  Int -> Int -> Int -> Int
max3 x y z | x >= y && x >= z = x
           | y >= x && y >= z = y
           | otherwise        = z
```

Equações Condicionais

```
max ::  Int -> Int -> Int
max x y | x >= y    = x
        | otherwise = y

max3 ::  Int -> Int -> Int -> Int
max3 x y z | x >= y && x >= z = x
           | y >= x && y >= z = y
           | otherwise        = z

otherwise ::  Bool
otherwise = True
```

Expressões Condicionais

```
max ::  Int -> Int -> Int
max x y = if x >= y then x else y

max3 ::  Int -> Int -> Int -> Int
max3 x y z = if x >= y && x >= z then x
             else if y >= x && y >= z then y
             else z
```

Outra maneira de definir max3

```
max3 ::  Int -> Int -> Int -> Int
max3 x y z = if x >= y then
                 if x >= z then x else z
             else
                 if y >= x then y else z
```

Observações importantes sobre condicionais

- Como sempre: escreva seu programa de maneira que ele seja fácil de ser entendido. Não se preocupe (por enquanto) com questões de eficiência: otimizações prematuras constituem a origem de muitos problemas.
- Condicionais são nossos amigos: sem eles, programas não fariam muita coisa interessante.
- Condicionais são nossos inimigos: cada condicional dobra o número de casos de teste a serem considerados. Um programa com cinco condicionais de dois ramos requer $2^5 = 32$ casos de teste, de modo a testar cada possível caminho no programa. Um programa com dez condicionais de dois ramos requer $2^{10} = 1024$ casos de teste.

Uma maneira melhor de definir max3

```
max3 ::  Int -> Int -> Int -> Int
max3 x y z = x `max` y `max` z
```

Uma maneira melhor de definir max3

```
max3 ::  Int -> Int -> Int -> Int
max3 x y z = x `max` y `max` z


max ::  Int -> Int -> Int
max x y | x >= y    = x
        | otherwise = y
```

Uma maneira melhor de definir max3

```
max3 ::  Int -> Int -> Int -> Int
max3 x y z = x `max` y `max` z
```

```
max ::  Int -> Int -> Int
max x y | x >= y    = x
        | otherwise = y
```

`x + y`	*significa*	`(+) x y`
`x >= y`	*significa*	`(>=) x y`
`x `max` y`	*significa*	`max x y`

Porque usar notação infixada

```
prop_max_assoc ::  Int -> Int -> Int -> Bool
prop_max_assoc x y z =
    (x `max` y) `max` z == x `max` (y `max` z)
```

Em expressões envolvendo um operador associativo os parenteses não importam, e podem ser omitidos.

Porque usar notação infixada

```
prop_max_assoc ::  Int -> Int -> Int -> Bool
prop_max_assoc x y z =
   (x `max` y) `max` z == x `max` (y `max` z)
```

Em expressões envolvendo um operador associativo os parenteses não importam, e podem ser omitidos.

```
prop_max_assoc ::  Int -> Int -> Int -> Bool
prop_max_assoc x y z =
   max (max x y) z == max x (max y z)
```

Isso é muito menos legível que a notação infixada!

Pontos importantes sobre associatividade

- Existem algumas poucas propriedaded importantes de operadores: *associatividade*, *identidade*, *commutatividade*, *distributividade*, *zero*, *idempotencia*. Você sabe o significado de cada uma dessas propriedades?
- Quando você encontrar um operador novo, a primeira pergunta que você deve fazer é “esse operador é associativo?” (A segunda é “ele possui uma identidade?”)
- Associatividade é nossa amiga, pois significa que não precisamos nos preocupar com a parenteses. O programa fica mais legível (mais fácil de ser entendido).
- Associatividade é nossa amiga, pois é a chave para escrever programas que executam duas vezes mais rápido em máquinas “dual-core”, e mil vezes mais rápido em uma máquina com mil processadores.

Parte II

Append

Append

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

```
   "abc"++ "de"
=
   'a' :  ('b' :  ('c' :  [])) ++ ('d' :  ('e' :  []))
=
   'a' :  (('b' :  ('c' :  [])) ++ ('d' :  ('e' :  [])))
=
   'a' :  ('b' :  (('c' :  []) ++ ('d' :  ('e' :  []))))
=
   'a' :  ('b' :  ('c' :  ([] ++ ('d' :  ('e' :  [])))))
=
   'a' :  ('b' :  ('c' :  ('d' :  ('e' :  []))))
=
   "abcde"
```

Append

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

```
   "abc"++ "de"
=
   'a' :  ("bc" ++ "de")
=
   'a' :  ('b' :  ("c" ++ "de"))
=
   'a' :  ('b' :  ('c' :  (" " ++ "de")))
=
   'a' :  ('b' :  ('c' :  "de"))
=
   "abcde"
```

Propriedades de Append

```
prop_append_assoc ::  [Int] -> [Int] -> [Int] -> Bool
prop_append_assoc xs ys zs =
   (xs ++ ys) ++ zs == xs ++ (ys ++ zs)

prop_append_ident ::  [Int] -> Bool
prop_append_ident xs =
   xs ++ [] == xs && xs == [] ++ xs

prop_append_cons ::  Int -> [Int] -> Bool
prop_append_cons x xs =
   [x] ++ xs == x :  xs
```

Parte IV

Contagem

Contagem

```
Prelude> [1..3]
[1,2,3]
Prelude> enumFromTo 1 3
[1,2,3]
```

Recursão

```
enumFromTo ::  Int -> Int -> [Int]
enumFromTo m n | m > n  = []
               | m <= n = m :  enumFromTo (m+1) n
```

Como enumFromTo funciona (recursão)

```
enumFromTo :: Int -> Int -> [Int]
enumFromTo m n | m > n  = []
               | m <= n = m : enumFromTo (m+1) n
```

```
   enumFromTo 1 3
=
   1 : enumFromTo 2 3
=
   1 : (2 : enumFromTo 3 3)
=
   1 : (2 : (3 : enumFromTo 4 3))
=
   1 : (2 : (3 : []))
=
   [1,2,3]
```

Fatorial

```
Main> factorial 3
```

Definição da biblioteca Prelude.hs

```
factorial ::  Int -> Int
factorial n = product [1..n]
```

Fatorial

```
Main> factorial 3
```

Definição da biblioteca Prelude.hs

```
factorial ::  Int -> Int
factorial n = product [1..n]
```

Recursão

```
factorialRec ::  Int -> Int
factorialRec n = fact 1 n
  where
    fact ::  Int -> Int -> Int
    fact m n | m > n  = 1
             | m <= n = m * fact (m+1) n
```

Como fatorial funciona (recursão)

```
factorialRec ::  Int -> Int
factorialRec n = fact 1 n
  where
    fact ::  Int -> Int -> Int
    fact m n | m > n  = 1
             | m <= n = m * fact (m+1) n
```

```
   factorialRec 3
=
   fact 1 3
=
   1 * fact 2 3
=
   1 * (2 * fact 3 3)
=
   1 * (2 * (3 * fact 4 3))
=
   1 * (2 * (3 * 1))
=
6
```

Parte V

Zip e Search

Zip

```
zip ::  [a] -> [b] -> [(a,b)]
zip [] ys          = []
zip (x:xs) []      = []
zip (x:xs) (y:ys) = (x,y) :  zip xs ys
```

```
   zip [0,1,2] "abc"
=
   (0,'a') :  zip [1,2] "bc"
=
   (0,'a') :  ((1,'b') :  zip [2] "c")
=
   (0,'a') :  ((1,'b') :  ((2,'c') :  zip [] []))
=
   (0,'a') :  ((1,'b') :  ((2,'c') :  []))
=
   [(0,'a'),(1,'b'),(2,'c')]
```

Duas definições equivalentes de zip

Mais concisa

```
zip ::  [a] -> [b] -> [(a,b)]
zip [] ys          = []
zip (x:xs) []      = []
zip (x:xs) (y:ys) = (x,y) :  zip xs ys
```

Mais longa

```
zip ::  [a] -> [b] -> [(a,b)]
zip [] []          = []
zip [] (y:ys)      = []
zip (x:xs) []      = []
zip (x:xs) (y:ys) = (x,y) :  zip xs ys
```

Duas definições alternativas de zip

Liberal

```
zip ::  [a] -> [b] -> [(a,b)]
zip [] []         = []
zip [] (y:ys)     = []
zip (x:xs) []     = []
zip (x:xs) (y:ys) = (x,y) :  zip xs ys
```

Conservadora

```
zipHarsh ::  [a] -> [b] -> [(a,b)]
zipHarsh [] []         = []
zipHarsh (x:xs) (y:ys) = (x,y) :  zipHarsh xs ys
```

Listas de diferentes comprimentos

```
Prelude> zip [0,1,2] "abc"
[(0,'a'),(1,'b'),(2,'c')]

Prelude> zipHarsh [0,1,2] "abc"
[(0,'a'),(1,'b'),(2,'c')]

Prelude> zip [0,1,2] "abcde"
[(0,'a'),(1,'b'),(2,'c')]

Prelude> zipHarsh [0,1,2] "abcde"
error

Prelude> zip [0,1,2,3,4] "abc"
[(0,'a'),(1,'b'),(2,'c')]

Prelude> zipHarsh [0,1,2,3,4] "abc"
error
```

Mais sobre zip

```
Prelude> zip [0..]  "words"
[(0,'w'),(1,'o'),(2,'r'),(3,'d'),(4,'s')]

Prelude> let pairs xs = zip xs (tail xs)
Prelude> pairs "words"
[('w','o'),('o','r'),('r','d'),('d','s')]
```

Zip com uma lista infinita

```
zip ::  [a] -> [b] -> [(a,b)]
zip [] ys          = []
zip (x:xs) []      = []
zip (x:xs) (y:ys) = (x,y) :  zip xs ys
```

```
   zip [0..]  "abc"
=
   zip [0..]  ('a' :  ('b' :  ('c' :  [])))
=
   (0,'a') :  zip [1..]  ('b' :  ('c' :  []))
=
   (0,'a') :  ((1,'b') :  zip [2..]  ('c' :  []))
=
   (0,'a') :  ((1,'b') :  ((2,'c') :  zip [3..]  []))
=
   (0,'a') :  ((1,'b') :  ((2,'c') :  []))
=
   [(0,'a'),(1,'b'),(2,'c')]
```

Search

```
Main> search "bookshop" 'o'
[1,2,6]
```

List comprehension – definição da biblioteca

```
search ::  [a] -> a -> [Int]
search xs y = [ i | (i,x) <- zip [0..]  xs, x==y ]
```

Recursão

```
searchRec ::  [a] -> a -> [Int]
searchRec xs y = srch xs y 0
  where
    srch ::  [a] -> a -> Int -> [Int]
    srch [] y i = []
    srch (x:xs) y i
      | x == y      = i :  srch xs y (i+1)
      | otherwise   = srch xs y (i+1)
```

Como search funciona (list comprehension)

```
search ::  [a] -> a -> [Int]
search xs y = [ i | (i,x) <- zip [0..]  xs, x==y ]

  search "book" 'o'
=
  [ i | (i,x) <- zip [0..]  "book", x=='o' ]
=
  [ i | (i,x) <- [(0,'b'),(1,'o'),(2,'o'),(3,'k')], x=='o' ]
=
  [0|'b'=='o']++[1|'o'=='o']++[2|'o'=='o']++[3|'k'=='o']
=
  []++[1]++[2]++[]
=
  [1,2]
```

Como search funciona (recursão)

```
searchRec xs y = srch xs y 0
  where
    srch [] y i = []
    srch (x:xs) y i | x == y     = i :  srch xs y (i+1)
                    | otherwise = srch xs y (i+1)

   search "book"  'o'
=
   srch "book"  'o' 0
=
   srch "ook"  'o' 1
=
   1 :  srch "ok"  'o' 2
=
   1 :  (2 :  srch "k"  'o' 3)
=
   1 :  (2 :  srch ""  'o' 4)
=
   1 :  (2 :  [])
=
   [1,2]
```

Parte VI

Select, take, e drop

Select, take, e drop

```
Prelude> "words"!!  3
'd'

Prelude> take 3 "words"
"wor"

Prelude> drop 3 "words"
"ds"
```

Select, take, e drop — list comprehension

```
(!!)  ::  [a] -> Int -> a
xs !!  i = the [ x | (j,x) <- zip [0..]  xs, j == i ]
   where the [x] = x

take ::  Int -> [a] -> [a]
take i xs = [ x | (j,x) <- zip [0..]  xs, j < i ]

drop ::  Int -> [a] -> [a]
drop i xs = [ x | (j,x) <- zip [0..]  xs, j >= i ]
```

Select, take, e drop — recursão

```
(!!)  ::  [a] -> Int -> a
(x:xs) !!  0          = x
(x:xs) !!  i | i > 0 = xs !!  (i-1)

take ::  Int -> [a] -> [a]
take 0 xs             = []
take i []     | i > 0 = []
take i (x:xs) | i > 0 = x :  take (i-1) xs

drop ::  Int -> [a] -> [a]
drop 0 xs             = xs
drop i []     | i > 0 = []
drop i (x:xs) | i > 0 = drop (i-1) xs
```

Como take funciona (list comprehension)

```
take ::  Int -> [a] -> [a]
take i xs = [ x | (j,x) <- zip [0..]  xs, j < i ]
```

```
   take 3 "words"
=
   [ x | (j,x) <- zip [0..]  "words", j < 3 ]
=
   [ x | (j,x) <- [(0,'w'),(1,'o'),(2,'r'),(3,'d'),(4,'s')],
         j < 3 ]
=
   ['w'|0<3]++['o'|1<3]++['r'|2<3]++['d'|3<3]++['s'|4<3]
=
   ['w']++['o']++['r']++[]++[]
=
"wor"
```

Como take funciona (recursão)

```
take ::  Int -> [a] -> [a]
take 0 xs                = []
take n []      | n > 0 = []
take n (x:xs) | n > 0 = x :  take (n-1) xs
```

```
   take 3 "words"
=
   'w' :  take 2 "ords"
=
   'w' :  ('o' :  take 1 "rds")
=
   'w' :  ('o' :  ('r' :  take 0 "rds"))
=
> 'w' :  ('o' :  ('r' :  []))
=
   "wor"
```

Listas

Toda lista pode ser escrita usando-se apenas (:) e [].

```
[1,2,3] = 1 :  (2 :  (3 :  []))

"list"  = ['l','i','s','t']
        = 'l' :  ('i' :  ('s' :  ('t' :  [])))
```

Uma definição *recursiva*: Uma *lista* é

- *nula*, representada por `[]`, ou
- *construída*, representada por `(x:xs)`
  com *cabeça* `x` (um elemento) e *cauda* `xs` (uma lista).

Números naturais

Todo número natural pode ser escrito usando-se apenas `(+1)` e `0`.

```
3  = ((0 + 1) + 1) + 1
```

Uma definição *recursiva*: Um *número natural* é

- *zero*, representado por `0`, ou
- *successor*, representado por `n+1`
  com *predecessor* `n` (um número natural).

Select, take, e drop (recursion)

```
(!!)  ::  Int -> [a] -> a
(x:xs) !!  0          = x
(x:xs) !!  i | i > 0 = xs !!  (i-1)

take ::  Int -> [a] -> [a]
take 0 xs             = []
take i []     | i > 0 = []
take i (x:xs) | i > 0 = x :  take (i-1) xs

drop ::  Int -> [a] -> [a]
drop 0 xs             = xs
drop i []     | i > 0 = []
drop i (x:xs) | i > 0 = drop (i-1) xs
```

Select, take, e drop ($n + 1$ *patterns*)

```
(!!)  ::  Int -> [a] -> a
(x:xs) !!  0     = x
(x:xs) !!  (i+1) = xs !!  i

take ::  Int -> [a] -> [a]
take 0 xs          = []
take (i+1) []      = []
take (i+1) (x:xs) = x :  take i xs

drop ::  Int -> [a] -> [a]
drop 0 xs          = xs
drop (i+1) []      = []
drop (i+1) (x:xs) = drop i xs
```

Como take funciona — reprise

```
take ::  Int -> [a] -> [a]
take 0 xs          = []
take (n+1) []      = []
take (n+1) (x:xs) = x :  take n xs
```

```
   take 3 "words"
=
   take (((0+1)+1)+1) ('w':('o':('r':('d':('s':[])))))
=
   'w' :  take ((0+1)+1) ('o':('r':('d':('s':[]))))
=
   'w' :  ('o' :  take (0+1) ('r':('d':('s':[]))))
=
   'w' :  ('o' :  ('r' :  take 0 ('d':('s':[]))))
=
   'w' :  ('o' :  ('r' :  []))
=
   "wor"
```

Aritmética

```
(+) ::  Int -> Int -> Int
m + 0      = m
m + (n+1) = (m + n) + 1

(*) ::  Int -> Int -> Int
m * 0      = 0
m * (n+1) = (m * n) + m

(^) ::  Int -> Int -> Int
m ^ 0      = 1
m ^ (n+1) = (m ^ n) * m
```