Programação Funcional — BCC222

Aulas 3,4

Listas e Recursão

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

List comprehensions — Geradores

```
Prelude> [ x*x | x <- [1,2,3] ]
[1,4,9]

Prelude> [ toLower c | c <- "Hello, World!"]
"hello, world!"

Prelude> [ (x, even x) | x <- [1,2,3] ]
[(1,False),(2,True),(3,False)]
```

x <- [1,2,3] é chamado de *gerador*

List comprehensions — Guardas

```
Prelude> [ x | x <- [1,2,3], odd x ]
[1,3]

Prelude> [ x*x | x <- [1,2,3], odd x ]
[1,9]

Prelude> [ x | x <- [42,-5,24,0,-3], x > 0 ]
[42,24]

Prelude> [ toLower c | c <- "Hello, World!", isAlpha c ]
"helloworld"
```

even x é chamado de *guarda*

Soma e Produto

```
Prelude> sum [1,2,3]
6

Prelude> sum []
0

Prelude> sum [ x*x | x <- [1,2,3], odd x ]
10

Prelude> product [1,2,3,4]
24

Prelude> product []
1

Prelude> let factorial n = product [1..n]
Prelude> factorial 4
24
```

Parte I

Mapeamento:

elevar ao quadrado cada elemento da lista

(*list comprehension*)

Como *list comprehension* funciona

```
squares ::  [Integer] -> [Integer]
squares xs = [ x*x | x <- xs ]


squares [1,2,3]
```

Como *list comprehension* funciona

```
squares ::  [Integer] -> [Integer]
squares xs = [ x*x | x <- xs ]


squares [1,2,3]
=      {xs = [1,2,3] }
[ x*x | x <- [1,2,3] ]
```

Como *list comprehension* funciona

```
squares ::  [Integer] -> [Integer]
squares xs = [ x*x | x <- xs ]
```

```
squares [1,2,3]
=
[ x*x | x <- [1,2,3] ]
=     {x = 1}, {x = 2}, {x = 3}
[ 1*1 ]++[ 2*2 ]++[ 3*3 ]
```

Como *list comprehension* funciona

```
squares :: [Integer] -> [Integer]
squares xs = [ x*x | x <- xs ]
```

```
squares [1,2,3]
=
[ x*x | x <- [1,2,3] ]
=
[ 1*1 ]++[ 2*2 ]++[ 3*3 ]
=
[ 1 ]++[ 4 ]++[ 9 ]
```

Como *list comprehension* funciona

```
squares :: [Integer] -> [Integer]
squares xs = [ x*x | x <- xs ]
```

```
  squares [1,2,3]
=
  [ x*x | x <- [1,2,3] ]
=
  [ 1*1 ]++[ 2*2 ]++[ 3*3 ]
=
  [ 1 ]++[ 4 ]++[ 9 ]
=
  [1,4,9]
```

Parte II

Filtragem:
selecionando os elementos impares da lista
(*list comprehension*)

Como *list comprehension* funciona

```
odds ::  [Integer] -> [Integer]
odds xs = [ x | x <- xs, odd x ]

odds [1,2,3]
```

Como *list comprehension* funciona

```
    odds ::  [Integer] -> [Integer]
    odds xs = [ x | x <- xs, odd x ]

    odds [1,2,3]
=       {xs = [1,2,3] }
    [ x | x <- [1,2,3], odd x ]
```

Como *list comprehension* funciona

```
    odds ::  [Integer] -> [Integer]
    odds xs = [ x | x <- xs, odd x ]

    odds [1,2,3]
=
    [ x | x <- [1,2,3], odd x ]
=       {x = 1}, {x = 2}, {x = 3}
    [ 1 | odd 1 ]++[ 2 | odd 2 ]++[ 3 | odd 3 ]
```

Como *list comprehension* funciona

```
    odds ::  [Integer] -> [Integer]
    odds xs = [ x | x <- xs, odd x ]

    odds [1,2,3]
=
    [ x | x <- [1,2,3], odd x ]
=
    [ 1 | odd 1 ]++[ 2 | odd 2 ]++[ 3 | odd 3 ]
=
    [ 1 | True ]++[ 2 | False ]++[ 3 | True ]
```

Como *list comprehension* funciona

```
odds ::  [Integer] -> [Integer]
odds xs = [ x | x <- xs, odd x ]

odds [1,2,3]
=
[ x | x <- [1,2,3], odd x ]
=
[ 1 | odd 1 ]++[ 2 | odd 2 ]++[ 3 | odd 3 ]
=
[ 1 | True ]++[ 2 | False ]++[ 3 | True ]
=
[1]++[]++[3]
```

Como *list comprehension* funciona

```
    odds ::  [Integer] -> [Integer]
    odds xs = [ x | x <- xs, odd x ]

    odds [1,2,3]
=
    [ x | x <- [1,2,3], odd x ]
=
    [ 1 | odd 1 ]++[ 2 | odd 2 ]++[ 3 | odd 3 ]
=
    [ 1 | True ]++[ 2 | False ]++[ 3 | True ]
=
    [1]++[]++[3]
=
    [1,3]
```

Parte III

Juntando tudo:

soma dos quadrados dos elementos impares da lista

(*list comprehension*)

Dois estilos de definição

Composição

```
f ::  [Integer] -> Integer
f xs = sum (squares (odds xs))
```

List Comprehension

```
fCom ::  [Integer] -> Integer
fCom xs = sum [ x*x | x <- xs, odd x ]
```

Como composição funciona

```
f ::  [Integer] -> [Integer]
f xs = sum (squares (odds xs))

f [1,2,3]
```

Como composição funciona

```
    f ::  [Integer] -> [Integer]
    f xs = sum (squares (odds xs))

    f [1,2,3]
=       {xs = [1,2,3] }
    sum (squares (odds [1,2,3]))
```

Como composição funciona

```
    f ::  [Integer] -> [Integer]
    f xs = sum (squares (odds xs))

    f [1,2,3]
=
    sum (squares (odds [1,2,3]))
=
    sum (squares [1,3])
```

Como composição funciona

```
    f ::  [Integer] -> [Integer]
    f xs = sum (squares (odds xs))

    f [1,2,3]
=
    sum (squares (odds [1,2,3]))
=
    sum (squares [1,3])
=
    sum [1,9]
```

Como composição funciona

```
    f ::  [Integer] -> [Integer]
    f xs = sum (squares (odds xs))

    f [1,2,3]
=
    sum (squares (odds [1,2,3]))
=
    sum (squares [1,3])
=
    sum [1,9]
=
    10
```

Como *list comprehension* funciona

```
fCom ::  [Integer] -> [Integer]
fCom xs = sum [ x*x | x <- xs, odd x ]

fCom [1,2,3]
```

Como *list comprehension* funciona

```
    fCom ::  [Integer] -> [Integer]
    fCom xs = sum [ x*x | x <- xs, odd x ]

    fCom [1,2,3]
=       {xs = [1,2,3] }
    sum [ x*x | x <- [1,2,3], odd x ]
```

Como *list comprehension* funciona

```
    fCom ::  [Integer] -> [Integer]
    fCom xs = sum [ x*x | x <- xs, odd x ]

    fCom [1,2,3]
=
    sum [ x*x | x <- [1,2,3], odd x ]
=       {x = 1}, {x = 2}, {x = 3}
    sum ([ 1*1 | odd 1 ]++[ 2*2 | odd 2 ]++[ 3*3 | odd 3 ])
```

Como *list comprehension* funciona

```
    fCom ::  [Integer] -> [Integer]
    fCom xs = sum [ x*x | x <- xs, odd x ]

    fCom [1,2,3]
=
    sum [ x*x | x <- [1,2,3], odd x ]
=
    sum ([ 1*1 | odd 1 ]++[ 2*2 | odd 2 ]++[ 3*3 | odd 3 ])
=
    sum ([ 1 | True ]++[ 4 | False ]++[ 9 | True])
```

Como *list comprehension* funciona

```
fCom ::  [Integer] -> [Integer]
fCom xs = sum [ x*x | x <- xs, odd x ]

fCom [1,2,3]
=
sum [ x*x | x <- [1,2,3], odd x ]
=
sum ([ 1*1 | odd 1 ]++[ 2*2 | odd 2 ]++[ 3*3 | odd 3 ])
=
sum ([ 1 | True ]++[ 4 | False ]++[ 9 | True])
=
sum ([1]++[]++[9])
```

Como *list comprehension* funciona

```
    fCom ::  [Integer] -> [Integer]
    fCom xs = sum [ x*x | x <- xs, odd x ]

    fCom [1,2,3]
=
    sum [ x*x | x <- [1,2,3], odd x ]
=
    sum ([ 1*1 | odd 1 ]++[ 2*2 | odd 2 ]++[ 3*3 | odd 3 ])
=
    sum ([ 1 | True ]++[ 4 | False ]++[ 9 | True])
=
    sum ([1]++[]++[9])
=
    sum ([1,9])
```

Como *list comprehension* funciona

```
    fCom ::  [Integer] -> [Integer]
    fCom xs = sum [ x*x | x <- xs, odd x ]

    fCom [1,2,3]
=
    sum [ x*x | x <- [1,2,3], odd x ]
=
    sum ([ 1*1 | odd 1 ]++[ 2*2 | odd 2 ]++[ 3*3 | odd 3 ])
=
    sum ([ 1 | True ]++[ 4 | False ]++[ 9 | True])
=
    sum ([1]++[]++[9])
=
    sum ([1,9])
=
    10
```

QuickCheck — arquivo lect03.hs

```
- lect03.hs

import Test.QuickCheck

squares ::  [Integer] -> [Integer]
squares xs = [ x*x | x <- xs ]

odds ::  [Integer] -> [Integer]
odds xs = [ x | x <- xs, odd x ]

f ::  [Integer] -> [Integer]
f xs = sum (squares (odds xs))

fCom ::  [Integer] -> [Integer]
fCom xs = sum [ x*x | x <- xs, odd x ]

prop_f ::  [Integer] -> Bool
prop_f xs = f xs == fCom xs
```

QuickCheck — executando o programa

```
% ghci lect03.hs
GHCi, version 6.10.4:  www.haskell.org/ghc/ :?  for help
Loading package ghc-prim ...  linking ...  done.
Loading package integer ...  linking ...  done.
Loading package base ...  linking ...  done.
[1 of 1] Compiling Main ( lect03.hs, interpreted )
Ok, modules loaded:  Main.
*Main> quickCheck prop_f
Loading package old-locale-1.0.0.0 ...  linking ...  done.
Loading package old-time-1.0.0.0 ...  linking ...  done.
Loading package random-1.0.0.0 ...  linking ...  done.
Loading package mtl-1.1.0.1 ...  linking ...  done.
Loading package QuickCheck-2.1 ...  linking ...  done.
+++ OK, passed 100 tests.
*Main>
```

Parte IV

Listas e Recursão

Cons e *append*

Cons tem como argumentos um elemento e uma lista.
Append tem como argumentos duas listas.

```
(:)  ::  a -> [a] -> [a]
(++) ::  [a] -> [a] -> [a]

1 :  [2,3]      =   [1,2,3]
[1] ++ [2,3]    =   [1,2,3]
[1,2] ++ [3]    =   [1,2,3]
'l' :  "ist"    =   "list"
"l"++ "ist"     =   "list"
"li"++ "st"     =   "list"

[1] :  [2,3]            - type error!
1 ++ [2,3]              - type error!
[1,2] ++ 3              - type error!
"l":  "ist"             - type error!
'l' ++ "ist"            - type error!
```

(:) pronuncia-se cons, de construct
(++) pronuncia-se append

Listas

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

```
[1,2,3]   =  1 :  (2 :  (3 :  []))
"list"    =  ['l','i','s','t']
          =  'l' :  ('i' :  ('s' :  ('tv :  [])))
```

Uma definição *recursiva*: Uma lista é

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

Se xs é uma lista não vazia, então

- head xs é a cabeça da lista
- tail xs é a cauda da lista

Uma lista de números

```
Prelude> null [1,2,3]
False
Prelude> head [1,2,3]
1
Prelude> tail [1,2,3]
[2,3]
Prelude> null [2,3]
False
Prelude> head [2,3]
2
Prelude> tail [2,3]
[3]
Prelude> null [3]
False
Prelude> head [3]
3
Prelude> tail [3]
[]
Prelude> null []
True
```

Parte V

Mapeamento:
elevar ao quadrado cada elemento da lista
(recursão)

Dois estilos de definição

List comprehension

```
squares ::  [Integer] -> [Integer]
squares xs = [ x*x | x <- xs ]
```

Recursion

```
squaresRec ::  [Integer] -> [Integer]
squaresRec []     = []
squaresRec (x:xs) = x*x :  squaresRec xs
```

Casamento de padrão e condicionais

Casamento de padrão

```
squaresRec ::  [Integer] -> [Integer]
squaresRec []     = []
squaresRec (x:xs) = x*x :  squaresRec xs
```

Condicionais e definições locais

```
squaresCond ::  [Integer] -> [Integer]
squaresCond ws =
  if null ws then
    []
  else
    let
      x = head ws
      xs= tail ws
    in
      x*x :  squaresCond xs
```

Como recursão funciona — squaresRec

```
squaresRec ::  [Integer] -> [Integer]
squaresRec []     = []
squaresRec (x:xs) = x*x :  squaresRec xs
```

```
squaresRec [1,2,3]
```

Como recursão funciona — squaresRec

```
squaresRec ::  [Integer] -> [Integer]
squaresRec []     = []
squaresRec (x:xs) = x*x :  squaresRec xs
```

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

Como recursão funciona — squaresRec

```
squaresRec ::  [Integer] -> [Integer]
squaresRec []     = []
squaresRec (x:xs) = x*x :  squaresRec xs
```

```
  squaresRec [1,2,3]
=
  squaresRec (1 :  (2 :  (3 :  [])))
=      { x = 1, xs = (2 :  (3 :  [])) }
  1*1 :  squaresRec (2 :  (3 :  []))
```

Como recursão funciona — squaresRec

```
squaresRec ::  [Integer] -> [Integer]
squaresRec []     = []
squaresRec (x:xs) = x*x :  squaresRec xs
```

```
  squaresRec [1,2,3]
=
  squaresRec (1 :  (2 :  (3 :  [])))
=
  1*1 :  squaresRec (2 :  (3 :  []))
=      { x = 2, xs = (3 :  []) }
  1*1 :  (2*2 :  squaresRec (3 :  []))
```

Como recursão funciona — squaresRec

```
squaresRec ::  [Integer] -> [Integer]
squaresRec []     = []
squaresRec (x:xs) = x*x :  squaresRec xs
```

```
  squaresRec [1,2,3]
=
  squaresRec (1 :  (2 :  (3 :  [])))
=
  1*1 :  squaresRec (2 :  (3 :  []))
=
  1*1 :  (2*2 :  squaresRec (3 :  []))
=     { x = 3, xs = [] }
  1*1 :  (2*2 :  (3*3 :  squaresRec []))
```

Como recursão funciona — squaresRec

```
squaresRec :: [Integer] -> [Integer]
squaresRec []     = []
squaresRec (x:xs) = x*x : squaresRec xs
```

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

Como recursão funciona — squaresRec

```
squaresRec ::  [Integer] -> [Integer]
squaresRec []     = []
squaresRec (x:xs) = x*x :  squaresRec xs
```

```
 squaresRec [1,2,3]
=
 squaresRec (1 :  (2 :  (3 :  [])))
=
 1*1 :  squaresRec (2 :  (3 :  []))
=
 1*1 :  (2*2 :  squaresRec (3 :  []))
=
 1*1 :  (2*2 :  (3*3 :  squaresRec []))
=
 1*1 :  (2*2 :  (3*3 :  []))
=
 1 :  (4 :  (9 :  []))
```

Como recursão funciona — squaresRec

```
squaresRec ::  [Integer] -> [Integer]
squaresRec []     = []
squaresRec (x:xs) = x*x :  squaresRec xs
```

```
  squaresRec [1,2,3]
=
  squaresRec (1 :  (2 :  (3 :  [])))
=
  1*1 :  squaresRec (2 :  (3 :  []))
=
  1*1 :  (2*2 :  squaresRec (3 :  []))
=
  1*1 :  (2*2 :  (3*3 :  squaresRec []))
=
  1*1 :  (2*2 :  (3*3 :  []))
=
  1 :  (4 :  (9 :  []))
=
  [1,4,9]
```

QuickCheck — arquivo lect03.hs

```
-- lect03.hs

import Test.QuickCheck

squares :: [Integer] -> [Integer]
squares xs = [ x*x | x <- xs ]

squaresRec :: [Integer] -> [Integer]
squaresRec []     = []
squaresRec (x:xs) = x*x : squaresRec xs

prop_squares :: [Integer] -> Bool
prop_squares xs = squares xs == squaresRec xs
```

QuickCheck — executando o programa

```
% ghci lect03.hs
GHCi, version 6.10.4:  www.haskell.org/ghc/ :?  for help
Loading package ghc-prim ...  linking ...  done.
Loading package integer ...  linking ...  done.
Loading package base ...  linking ...  done.
[1 of 1] Compiling Main ( lect03.hs, interpreted )
Ok, modules loaded:  Main.
*Main> quickCheck prop_squares
Loading package old-locale-1.0.0.0 ...  linking ...  done.
Loading package old-time-1.0.0.0 ...  linking ...  done.
Loading package random-1.0.0.0 ...  linking ...  done.
Loading package mtl-1.1.0.1 ...  linking ...  done.
Loading package QuickCheck-2.1 ...  linking ...  done.
+++ OK, passed 100 tests.
*Main>
```

Parte VI

Filtragem:
selecionando os elementos impares da lista
(recursão)

Dois estilos de definição

List comprehension

```
odds ::  [Integer] -> [Integer]
odds xs = [ x | x <- xs, odd x ]
```

Recursion

```
oddsRec ::  [Integer] -> [Integer]
oddsRec [] = []
oddsRec (x:xs)
   | odd x = x :  oddsRec xs
   | otherwise = oddsRec xs
```

Casamento de padrão e condicionais

Casamento de padrão com guardas

```
oddsRec ::  [Integer] -> [Integer]
oddsRec [] = []
oddsRec (x:xs)
   | odd x = x :  oddsRec xs
   | otherwise = oddsRec xs
```

Condicionais e definições locais

```
oddsCond ::  [Integer] -> [Integer]
oddsCond ws =
  if null ws then []
  else
    let
      x = head ws
      xs= tail ws
    in
      if odd x then
        x :  oddsCond xs
      else
        oddsCond xs
```

Como recursão funciona — oddsRec

```
oddsRec ::  [Integer] -> [Integer]
oddsRec [] = []
oddsRec (x:xs)
   | odd x = x :  oddsRec xs
   | otherwise = oddsRec xs
```

```
oddsRec [1,2,3]
```

Como recursão funciona — oddsRec

```
oddsRec ::  [Integer] -> [Integer]
oddsRec [] = []
oddsRec (x:xs)
   | odd x = x :  oddsRec xs
   | otherwise = oddsRec xs
```

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

Como recursão funciona — oddsRec

```
oddsRec ::  [Integer] -> [Integer]
oddsRec [] = []
oddsRec (x:xs)
   | odd x = x :  oddsRec xs
   | otherwise = oddsRec xs
```

```
   oddsRec [1,2,3]
=
   oddsRec (1 :  (2 :  (3 :  [])))
=       { x = 1, xs = (2 :  (3 :  [])), odd 1 = True }
   1:  oddsRec (2 :  (3 :  []))
```

Como recursão funciona — oddsRec

```
oddsRec ::  [Integer] -> [Integer]
oddsRec [] = []
oddsRec (x:xs)
   | odd x = x :  oddsRec xs
   | otherwise = oddsRec xs
```

```
   oddsRec [1,2,3]
=
   oddsRec (1 :  (2 :  (3 :  [])))
=
   1:  oddsRec (2 :  (3 :  []))
=      { x = 2, xs = (3 :  []), odd 2 = False }
   1 :  (oddsRec (3 :  []))
```

Como recursão funciona — oddsRec

```
oddsRec :: [Integer] -> [Integer]
oddsRec [] = []
oddsRec (x:xs)
   | odd x = x : oddsRec xs
   | otherwise = oddsRec xs
```

```
  oddsRec [1,2,3]
=
  oddsRec (1 : (2 : (3 : [])))
=
  1: oddsRec (2 : (3 : []))
=
  1 : (oddsRec (3 : []))
=     { x = 3, xs = [], odd 3 = True }
  1 : (3 : oddsRec [])
```

Como recursão funciona — oddsRec

```
oddsRec :: [Integer] -> [Integer]
oddsRec [] = []
oddsRec (x:xs)
   | odd x = x : oddsRec xs
   | otherwise = oddsRec xs
```

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

Como recursão funciona — oddsRec

```
oddsRec ::  [Integer] -> [Integer]
oddsRec [] = []
oddsRec (x:xs)
   | odd x = x :  oddsRec xs
   | otherwise = oddsRec xs
```

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

Parte VII

Somatório:
somando os elementos de uma lista

Soma dos elementos de uma lista — sum

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

```
sum [1,2,3]
```

Soma dos elementos de uma lista — sum

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

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

Soma dos elementos de uma lista — sum

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

```
   sum [1,2,3]
=
   sum (1 :  (2 :  (3 :  [])))
=       { x = 1, xs = (2 :  (3 :  [])) }
   1 + sum (2 :  (3 :  []))
```

Soma dos elementos de uma lista — sum

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

```
  sum [1,2,3]
=
  sum (1 :  (2 :  (3 :  [])))
=
  1 + sum (2 :  (3 :  []))
=       { x = 2, xs = (3 :  []) }
  1 + (2 + sum (3 :  []))
```

Soma dos elementos de uma lista — sum

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

```
   sum [1,2,3]
=
   sum (1 :  (2 :  (3 :  [])))
=
   1 + sum (2 :  (3 :  []))
=
   1 + (2 + sum (3 :  []))
=       { x = 3, xs = [] }
   1 + (2 + 3 + sum [])
```

Soma dos elementos de uma lista — sum

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

```
   sum [1,2,3]
=
   sum (1 :  (2 :  (3 :  [])))
=
   1 + sum (2 :  (3 :  []))
=
   1 + (2 + sum (3 :  []))
=
   1 + (2 + 3 + sum [])
=
   1 + (2 + (3 + 0))
```

Soma dos elementos de uma lista — sum

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

```
  sum [1,2,3]
=
  sum (1 :  (2 :  (3 :  [])))
=
  1 + sum (2 :  (3 :  []))
=
  1 + (2 + sum (3 :  []))
=
  1 + (2 + 3 + sum [])
=
  1 + (2 + (3 + 0))
=
  6
```

Soma dos elementos de uma lista — sum

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

```
   sum [1,2,3]
=
   sum (1 :  (2 :  (3 :  [])))
=
   1 + sum (2 :  (3 :  []))
=
   1 + (2 + sum (3 :  []))
=
   1 + (2 + 3 + sum [])
=
   1 + (2 + (3 + 0))
=
   6
```

Soma dos elementos de uma lista — sum

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

```
  sum [1,2,3]
=
  sum (1 :  (2 :  (3 :  [])))
=
  1 + sum (2 :  (3 :  []))
=
  1 + (2 + sum (3 :  []))
=
  1 + (2 + 3 + sum [])
=
  1 + (2 + (3 + 0))
=
  6
```

Produto dos elementos de uma lista — product

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

```
  product [1,2,3]
=
  product (1 :  (2 :  (3 :  [])))
=
  1 * product (2 :  (3 :  []))
=
  1 * (2 * product (3 :  []))
=
  1 * (2 * 3 * product [])
=
  1 * (2 * (3 * 1))
=
  6
```

Parte VIII

Juntando tudo:
soma dos quadrados dos elementos impares da lista
(recursão)

Três estilos de definição

Composição

```
f ::  [Integer] -> Integer
f xs = sum (squares (odds xs))
```

List Comprehension

```
fCom ::  [Integer] -> Integer
fCom xs = sum [ x*x | x <- xs, odd x ]
```

Recursão

```
fRec ::  [Integer] -> Integer
fRec []     = 0
fRec (x:xs) =
   | odd x      = x*x + fRec xs
   | otherwise = fRec xs
```

Como recursão funciona — fRec

```
fRec ::  [Integer] -> Integer
fRec []     = 0
fRec (x:xs) =
   | odd x      = x*x + fRec xs
   | otherwise = fRec xs
```

```
fRec [1,2,3]
```

Como recursão funciona — fRec

```
fRec ::  [Integer] -> Integer
fRec []     = 0
fRec (x:xs) =
   | odd x      = x*x + fRec xs
   | otherwise = fRec xs
```

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

Como recursão funciona — fRec

```
fRec ::  [Integer] -> Integer
fRec []     = 0
fRec (x:xs) =
   | odd x      = x*x + fRec xs
   | otherwise = fRec xs
```

```
   fRec [1,2,3]
=
   fRec (1 :  (2 :  (3 :  [])))
=       { x = 1, xs = (2 :  (3 :  [])), odd 1 = True }
   1*1 + fRec (2 :  (3 :  []))
```

Como recursão funciona — fRec

```
fRec ::  [Integer] -> Integer
fRec []     = 0
fRec (x:xs) =
   | odd x      = x*x + fRec xs
   | otherwise = fRec xs
```

```
  fRec [1,2,3]
=
  fRec (1 :  (2 :  (3 :  [])))
=
  1*1 + fRec (2 :  (3 :  []))
=       { x = 2, xs = (3 :  []), odd 2 = False }
  1*1 + (fRec (3 :  []))
```

Como recursão funciona — fRec

```
fRec ::  [Integer] -> Integer
fRec []     = 0
fRec (x:xs) =
   | odd x      = x*x + fRec xs
   | otherwise = fRec xs
```

```
  fRec [1,2,3]
=
  fRec (1 :  (2 :  (3 :  [])))
=
  1*1 + fRec (2 :  (3 :  []))
=
  1*1 + (fRec (3 :  []))
=       { x = 3, xs = [], odd 3 = True }
  1*1 + (3*3 + fRec [])
```

Como recursão funciona — fRec

```
fRec ::  [Integer] -> Integer
fRec []     = 0
fRec (x:xs) =
   | odd x      = x*x + fRec xs
   | otherwise = fRec xs
```

```
  fRec [1,2,3]
=
  fRec (1 :  (2 :  (3 :  [])))
=
  1*1 + fRec (2 :  (3 :  []))
=
  1*1 + (fRec (3 :  []))
=
  1*1 + (3*3 + fRec [])
=
  1*1 + (3*3 + 0)
```

Como recursão funciona — fRec

```
fRec ::  [Integer] -> Integer
fRec []     = 0
fRec (x:xs) =
   | odd x      = x*x + fRec xs
   | otherwise = fRec xs
```

```
  fRec [1,2,3]
=
  fRec (1 :  (2 :  (3 :  [])))
=
  1*1 + fRec (2 :  (3 :  []))
=
  1*1 + (fRec (3 :  []))
=
  1*1 + (3*3 + fRec [])
=
  1*1 + (3*3 + 0)
=
  1 + (9 + 0)
```

Como recursão funciona — fRec

```
fRec ::  [Integer] -> Integer
fRec []     = 0
fRec (x:xs) =
   | odd x      = x*x + fRec xs
   | otherwise = fRec xs
```

```
  fRec [1,2,3]
=
  fRec (1 :  (2 :  (3 :  [])))
=
  1*1 + fRec (2 :  (3 :  []))
=
  1*1 + (fRec (3 :  []))
=
  1*1 + (3*3 + fRec [])
=
  1*1 + (3*3 + 0)
=
  1 + (9 + 0)
=
  10
```

Como recursão funciona — fRec

```
fRec ::  [Integer] -> Integer
fRec []     = 0
fRec (x:xs) =
   | odd x      = x*x + fRec xs
   | otherwise = fRec xs
```

```
  fRec [1,2,3]
=
  fRec (1 :  (2 :  (3 :  [])))
=
  1*1 + fRec (2 :  (3 :  []))
=
  1*1 + (fRec (3 :  []))
=
  1*1 + (3*3 + fRec [])
=
  1*1 + (3*3 + 0)
=
  1 + (9 + 0)
=
  10
```