Programare declarativă[1]

Funcții, liste, recursie

Traian Florin Șerbănuță

Departamentul de Informatică, FMI, UNIBUC
traian.serbanuta@fmi.unibuc.ro

14 octombrie 2016

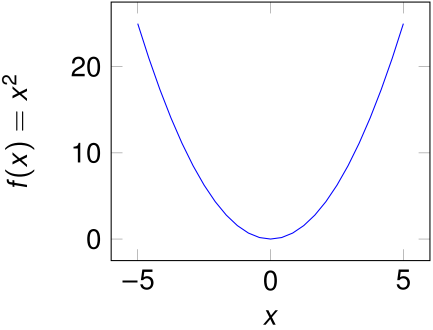
[1] bazat pe cursul Informatics 1: Functional Programming de la University of Edinburgh

Funcții și recursie

Ce e o funcție?

Ce e o funcție?

- DEX(online): Mărime variabilă care depinde de una sau de mai multe mărimi variabile independente
- O rețetă pentru a obține ieșiri din intrări: „Ridică un număr la pătrat”
- O relație între intrări și ieșiri $\{(1,1),(2,4),(3,9),(4,16),\ldots\}$
- O ecuație algebrică $f(x) = x^2$
- Un grafic reprezentând ieșirile pentru intrările posibile



Tipuri de date

pentru intrări/ieșiri ale funcțiilor

- Integer: 4, 0, -5
- Float: 3.14
- Char: ’a’
- String: "abc"
- Bool: True, False

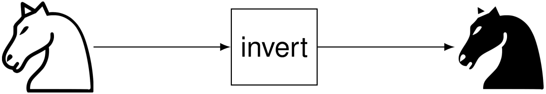
Tipuri de date

pentru intrări/ieșiri ale funcțiilor

- Integer: 4, 0, -5
- Float: 3.14
- Char: ’a’
- String: "abc"
- Bool: True, False
- Picture: , 

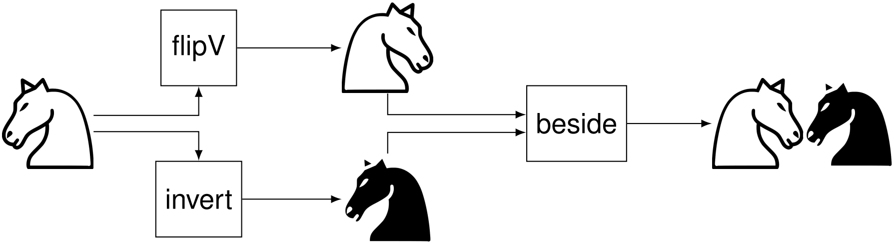
Tipuri de funcții și aplicarea lor

```
invert :: Picture -> Picture

knight :: Picture

invert knight
```



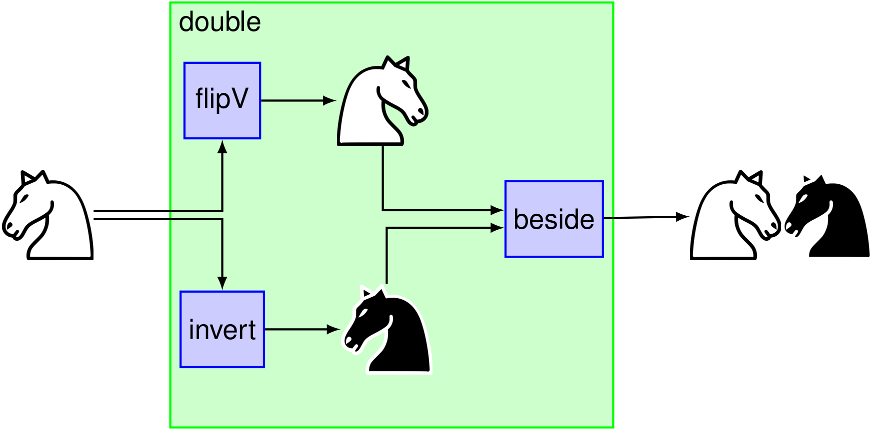
Compunerea funcțiilor

```
beside :: Picture -> Picture -> Picture
flipV  :: Picture -> Picture
invert :: Picture -> Picture

knight :: Picture

beside (flipV knight) (invert knight)
```



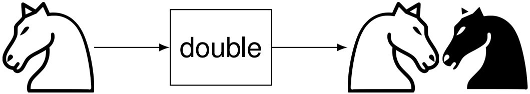
Definirea unei funcții noi

```
double :: Picture -> Picture
double p = beside (flipV p) (invert p)

double knight
```



Definirea unei funcții noi

```
double :: Picture -> Picture
double p = beside (flipV p) (invert p)

double knight
```



Terminologie

Prototipul funcției double :: Picture -> Picture

- Numele funcției
- Signatura funcției

Definiția funcției double p = beside (flipV p) (invert p)

- numele funcției
- parametrul formal
- corpul funcției

Aplicarea funcției double knight

- numele funcției
- parametrul actual (argumentul)

Liste

Operatorii : și ++

Mod de folosire

```
Prelude> :t (:)
(:) :: a -> [a] -> [a]

Prelude> 1 : [2,3]
[1,2,3]

Prelude> :t "bcd"
"bc" :: [Char]
Prelude> 'a' : "bcd"
"abcd"
```

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

Prelude> [1] ++ [2,3]
[1,2,3]
Prelude> [1,2] ++ [3]
[1,2,3]
Prelude> "a" ++ "bcd"
"abcd"
Prelude> "ab" ++ "cd"
"abcd"
```

- `:` (cons) Construiește o listă nouă având primul argument ca prim element și continuând cu al doilea argument ca restul listei.
- `++` (append) Construiește o listă nouă obținută prin alipirea celor două liste argument
- `[Char]` Șirurile de caractere (String) sunt liste de caractere (Char)

Operatorii : și ++

Erori de începător

```
Prelude> :t (:)
(:) :: a -> [a] -> [a]

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

```
Prelude> [1,2] : 3
  -- eroare de tipuri
Prelude> 1 ++ [2,3]
  -- eroare de tipuri
Prelude> [1] : [2,3]
  -- eroare de tipuri
```

```
Prelude> "ab" : 'c'
  -- eroare de tipuri
Prelude> 'a' ++ "bc"
  -- eroare de tipuri
Prelude> "a" : "bc"
  -- eroare de tipuri
```

Liste
Definiție

Observație

Orice listă poate fi scrisă folosind doar constructorul (:) și lista vidă []

- [1,2,3] == 1 : (2 : (3 : [])) == 1 : 2 : 3 : []
- "abcd" == ['a','b','c','d'] == 'a' : ('b' : ('c' : ('d' : []))) == 'a' : 'b' : 'c' : 'd' : []

Definiție recursivă

O listă este

- vidă, notată []; sau
- compusă, notată x:xs, dintr-un un element x numit capul listei (head) și o listă xs numită coada listei (tail).

Procesarea listelor

```
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
```

Transformarea fiecărui element dintr-o listă

Problemă și abordare

Definiți o funcție care pentru o listă de numere întregi dată ridică la pătrat fiecare element din lista.

Soluție descriptivă

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

Soluție recursivă

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

Variante recursive

Ecuațional (pattern matching)

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

Condițional (cu operatori de legare)

```haskell
squaresCond :: [Int] -> [Int]
squaresCond ys =
  if null ys then []
  else let
         x  = head ys
         xs = tail ys
       in
         x*x : squaresCond xs
```

Recursia în acțiune

```
squaresRec :: [Int] -> [Int]

squaresRec []     = []
squaresRec (x:xs) = x*x : squaresRec xs
```

squaresRec [1,2,3]

Recursia în acțiune

```
squaresRec :: [Int] -> [Int]
```

```
squaresRec []     = []
squaresRec (x:xs) = x*x : squaresRec xs
```

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

Recursia în acțiune

```
squaresRec :: [Int] -> [Int]
```

```
squaresRec []     = []
squaresRec (x:xs) = x*x : squaresRec xs
```

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

$\{x \mapsto 1, xs \mapsto 2 : (3 : [])\}$

Recursia în acțiune

```
squaresRec :: [Int] -> [Int]

squaresRec []     = []
squaresRec (x:xs) = x*x : squaresRec xs
```

squaresRec [1,2,3]

=

squaresRec (1 : (2 : (3 : [])))

=

1 * 1 : squaresRec (2 : (3 : []))

= $\{x \mapsto 2, xs \mapsto 3 : []\}$

1 * 1 : (2 * 2 : squaresRec (3 : []))

Recursia în acțiune

```
squaresRec :: [Int] -> [Int]
```

```
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 []))
```

$\{x \mapsto 3, xs \mapsto []\}$

Recursia în acțiune

```
squaresRec :: [Int] -> [Int]

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 : []))
```

Recursia în acțiune

```
squaresRec :: [Int] -> [Int]

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 : []))
```

Recursia în acțiune

```
squaresRec :: [Int] -> [Int]
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]
```

Selectarea elementelor dintr-o listă

Problemă și abordare

Definiți o funcție care dată fiind o listă de numere întregi selectează doar elementele impare din listă.

Soluție descriptivă

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

Soluție recursivă

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

Variante recursive

Ecuațional (pattern matching)

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

Condițional (cu operatori de legare)

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

Recursia în acțiune

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

oddsRec [1,2,3]

Recursia în acțiune

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

oddsRec [1,2,3]

=

oddsRec (1 : (2 : (3 : [])))

Recursia în acțiune

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

oddsRec [1,2,3]

=

oddsRec (1 : (2 : (3 : [])))

= $\{x \mapsto 1, xs \mapsto 2 : (3 : [])\}$; odd 1 = True

1 : oddsRec (2 : (3 : []))

Recursia în acțiune

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

oddsRec [1,2,3]

=

oddsRec (1 : (2 : (3 : [])))

=

1 : oddsRec (2 : (3 : []))

= $\{x \mapsto 2, xs \mapsto 3 : []\}$; odd 2 = False

1 : oddsRec (3 : [])

Recursia în acțiune

```
oddsRec :: [Int] -> [Int]     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 : [])
= $\{x \mapsto 3, xs \mapsto []\}$; odd 3 = True
1 : (3 : oddsRec [])

Recursia în acțiune

```
oddsRec :: [Int] -> [Int]     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 : [])
```

Recursia în acțiune

```
oddsRec :: [Int] -> [Int]    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]

Agregarea elementelor dintr-o listă

Problemă și abordare

Definiți o funcție care dată fiind o listă de numere întregi calculează suma elementelor din listă.

Soluție recursivă

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

Recursia în acțiune

```
suma :: [Int] -> Int
```

```
suma []     = 0
suma (x:xs) = x + suma xs
```

suma [1,2,3]

Recursia în acțiune

```
suma :: [Int] -> Int
```

```
suma []     = 0
suma (x:xs) = x + suma xs
```

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

Recursia în acțiune

```
suma :: [Int] -> Int
```

```
suma []     = 0
suma (x:xs) = x + suma xs
```

suma [1,2,3]

=

suma (1 : (2 : (3 : [])))

=

1 + suma (2 : (3 : []))

$\{x \mapsto 1, xs \mapsto 2 : (3 : [])\}$

Recursia în acțiune

```
suma :: [Int] -> Int
```

$$\text{suma } [\,] = 0$$
$$\text{suma } (x{:}xs) = x + \text{suma } xs$$

suma [1,2,3]

=

suma (1 : (2 : (3 : [])))

=

1 + suma (2 : (3 : []))

= $\{x \mapsto 2, xs \mapsto 3 : [\,]\}$

1 + (2 + suma (3 : []))

Recursia în acțiune

```
suma :: [Int] -> Int
```

suma [] $= 0$
suma (x:xs) $=$ x + suma xs

suma [1,2,3]
=
suma (1 : (2 : (3 : [])))
=
1 + suma (2 : (3 : []))
=
1 + (2 + suma (3 : []))
=
1 + (2 + (3 + suma []))

$\{x \mapsto 3, xs \mapsto []\}$

Recursia în acțiune

```
suma :: [Int] -> Int
```

```
suma []     = 0
suma (x:xs) = x + suma xs
```

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

Recursia în acțiune

```
suma :: [Int] -> Int
```

suma [] $= 0$
suma (x:xs) $=$ x + suma xs

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

Problemă și abordare

Definiți o funcție care dată fiind o listă de numere întregi calculează produsul elementelor din listă.

Soluție recursivă

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

Recursia în acțiune

```
produs :: [Int] -> Int
```

```
produs []     = 1
produs (x:xs) = x * produs xs
```

produs [1,2,3]

Recursia în acțiune

```
produs :: [Int] -> Int
```

```
produs []     = 1
produs (x:xs) = x * produs xs
```

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

Recursia în acțiune

```
produs :: [Int] -> Int
```

```
produs []     = 1
produs (x:xs) = x * produs xs
```

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

$\{x \mapsto 1, xs \mapsto 2 : (3 : [])\}$

Recursia în acțiune

```
produs :: [Int] -> Int
```

$$
\begin{aligned}
\text{produs }[\,] &= 1\\
\text{produs }(x{:}xs) &= x * \text{produs } xs
\end{aligned}
$$

produs [1,2,3]

=

produs (1 : (2 : (3 : [])))

=

1 * produs (2 : (3 : []))

=

$\{x \mapsto 2, xs \mapsto 3 : []\}$

1 * (2 * produs (3 : []))

Recursia în acțiune

```
produs :: [Int] -> Int
```

$$\text{produs } [\,] = 1$$
$$\text{produs } (x{:}xs) = x * \text{produs } xs$$

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

$\{x \mapsto 3, xs \mapsto [\,]\}$

Recursia în acțiune

```
produs :: [Int] -> Int
```

```
produs []     = 1
produs (x:xs) = x * produs xs
```

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

Recursia în acțiune

```
produs :: [Int] -> Int
```

$$\text{produs [] } = 1$$
$$\text{produs (x:xs)} = \text{x * produs xs}$$

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

Mapare, filtrare și agregare deodată

Problemă și abordare

Definiți o funcție care dată fiind o listă de numere întregi calculează suma pătratelor elementelor impare din listă.

Soluție descriptivă

```haskell
sumSqOdd :: [Int] -> Int
sumSqOdd xs = sum [ x * x | x <- xs, odd x ]
```

Soluție recursivă

```haskell
sumSqOddRec :: [Int] -> Int
sumSqOddRec []                     = 0
sumSqOddRec (x:xs) | odd x         = x * x + sumSqOddRec xs
                   | otherwise     = sumSqOddRec xs
```

Soluție combinatorială

```haskell
sumSqOdd = sum . squares . odds
```

Recursia în acțiune

```
oddsRec :: [Int] -> [Int]
sumSqOddRec []                          = 0
sumSqOddRec (x:xs)  |  odd x            = x*x + sumSqOddRec xs
                    |  otherwise        = sumSqOddRec xs
```

sumSqOddRec [1,2,3]

Recursia în acțiune

```
oddsRec :: [Int] -> [Int]
sumSqOddRec []                         = 0
sumSqOddRec (x:xs)   |   odd x         = x*x + sumSqOddRec xs
                     |   otherwise     = sumSqOddRec xs
```

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

Recursia în acțiune

```
oddsRec :: [Int] -> [Int]
sumSqOddRec []                       = 0
sumSqOddRec (x:xs)   |   odd x       = x*x + sumSqOddRec xs
                     |   otherwise   = sumSqOddRec xs
```

sumSqOddRec [1,2,3] =

sumSqOddRec (1 : (2 : (3 : [])))

= $\{x \mapsto 1, xs \mapsto 2 : (3 : [])\}$; odd 1 = True

1 * 1 + sumSqOddRec (2 : (3 : []))

Recursia în acțiune

```
oddsRec :: [Int] -> [Int]
sumSqOddRec []                           = 0
sumSqOddRec (x:xs)   |   odd x           = x*x + sumSqOddRec xs
                     |   otherwise       = sumSqOddRec xs
```

sumSqOddRec [1,2,3] =

sumSqOddRec (1 : (2 : (3 : [])))

=

1 * 1 + sumSqOddRec (2 : (3 : []))

= $\{x \mapsto 2, xs \mapsto 3 : []\}$; odd 2 = False

1 * 1 + sumSqOddRec (3 : [])

Recursia în acțiune

```
oddsRec :: [Int] -> [Int]
sumSqOddRec []                      = 0
sumSqOddRec (x:xs)  |  odd x        = x*x + sumSqOddRec xs
                    |  otherwise    = sumSqOddRec xs
```

sumSqOddRec [1,2,3] =
sumSqOddRec (1 : (2 : (3 : [])))
=
1 * 1 + sumSqOddRec (2 : (3 : []))
=
1 * 1 + sumSqOddRec (3 : [])
= $\{x \mapsto 3, xs \mapsto []\}$; odd 3 = True
1 * 1 + (3 * 3 + sumSqOddRec [])

Recursia în acțiune

```
oddsRec :: [Int] -> [Int]
sumSqOddRec []                       = 0
sumSqOddRec (x:xs)  |  odd x         = x*x + sumSqOddRec xs
                    |  otherwise     = sumSqOddRec xs
```

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

Recursia în acțiune

```
oddsRec :: [Int] -> [Int]
sumSqOddRec []                           = 0
sumSqOddRec (x:xs)   |   odd x           = x*x + sumSqOddRec xs
                     |   otherwise       = sumSqOddRec xs
```

sumSqOddRec [1,2,3] =
sumSqOddRec (1 : (2 : (3 : [])))
=
1 * 1 + sumSqOddRec (2 : (3 : []))
=
1 * 1 + sumSqOddRec (3 : [])
=
1 * 1 + (3 * 3 + sumSqOddRec [])
=
1 * 1 + (3 * 3 + 0) = 10