Informatics 1
Functional Programming Lectures 3 and 4
Monday 1 and Tuesday 2 October 2012

Lists and Recursion

Don Sannella
University of Edinburgh
Part I

List Comprehensions
Lists — Some examples

someNumbers :: [Integer]
someNumbers = [1,2,3]

someChars :: [Char]
    -- equivalent: someChars :: String
someChars :: ['I','n','f','1']
    -- equivalent: someChars = "Inf1"

someLists :: [[[Integer]]]
someLists = [[[1],[2,4,2],[]],[3,5]]

someFunctions :: [Picture -> Picture]
someFunctions = [invert,flipV]

someStuff = [1,"Inf1",[2,3]]       -- type error!
List comprehensions — Generators

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

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

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

x <- [1,2,3] is called a generator

<- is pronounced drawn from
List comprehensions — Guards

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

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

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

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

isOdd x is called a guard
Sum, Product

Prelude> sum [1,2,3]
6

Prelude> sum []
0

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

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

Prelude> product []
1

Prelude> let factorial n = product [1..n]
Prelude> factorial 4
24
Example uses of comprehensions

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

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

sumSqOdd :: [Integer] -> Integer
sumSqOdd xs = sum [ x*x | x <- xs, isOdd x ]
QuickCheck, a program

-- sumSqOdd.hs

import Test.QuickCheck

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

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

sumSqOdd :: [Integer] -> [Integer]
sumSqOdd xs = sum [ x*x | x <- xs, isOdd x ]

prop_sumSqOdd :: [Integer] -> Bool
prop_sumSqOdd xs = sum (squares (odds xs)) == sumSqOdd xs
QuickCheck, running the program

[melchior@dts: ghci sumSqOdd.hs
GHCi, version 6.8.3: http://www.haskell.org/ghc/ :? for help
Loading package base ... linking ... done.
[1 of 1] Compiling Main ( sumSqOdd.hs, interpreted )
* Main> quickCheck prop_sumSqOdd
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>
Part II

Lists and Recursion
Cons and append

Cons takes an element and a list.
Append takes two lists.

( : ) :: 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!
" l " : "ist" -- type error!
' l ' ++ "ist" -- type error!

( : ) is pronounced cons, for construct
( ++ ) is pronounced append
Lists

Every list can be written using only (: ) and [ ].

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

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

A recursive definition: A list is either

- null, written [], or

- constructed, written \( x : xs \), with head \( x \) (an element), and tail \( xs \) (a list).
A list of numbers

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
Part III

Mapping: Square every element of a list
Two styles of definition—squares

Comprehension

\[\text{squares} :: \text{[Integer]} \to \text{[Integer]}\]
\[\text{squares } x\mathbf{s} = [ x\times x \mid x \leftarrow x\mathbf{s} ]\]

Recursion

\[\text{squaresRec} :: \text{[Integer]} \to \text{[Integer]}\]
\[\text{squaresRec } [] = []\]
\[\text{squaresRec } (x:x\mathbf{s}) = x\times x : \text{squaresRec } x\mathbf{s}\]
Pattern matching and conditionals

Pattern matching

\[ \text{squaresRec} :: [\text{Integer}] \rightarrow [\text{Integer}] \]
\[ \text{squaresRec} \ [\] = [\] \]
\[ \text{squaresRec} \ (x:xs) = x \times x : \text{squaresRec} \ xs \]

Conditionals with binding

\[ \text{squaresCond} :: [\text{Integer}] \rightarrow [\text{Integer}] \]
\[ \text{squaresCond} \ ws = \]
\[ \text{if null} \ ws \ \text{then} \]
\[ [\] \]
\[ \text{else} \]
\[ \text{let} \]
\[ x = \text{head} \ ws \]
\[ xs = \text{tail} \ ws \]
\[ \text{in} \]
\[ x \times x : \text{squaresCond} \ xs \]
How recursion works—squaresRec

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

squaresRec [1,2,3]
How recursion works—squaresRec

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

squaresRec [1,2,3]
=
squaresRec (1 : (2 : (3 : []))))
How recursion works—\textit{squaresRec}

\begin{verbatim}

squaresRec :: [Integer] \rightarrow [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 : []))

\end{verbatim}

How recursion works—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 : []))
How recursion works—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 []))
How recursion works—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 : []))
How recursion works—\textit{squaresRec}

\begin{verbatim}
  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 : []))
\end{verbatim}
How recursion works—squaresRec

\[
squaresRec :: [Integer] \rightarrow [Integer]
\]
\[
squaresRec [] = []
\]
\[
squaresRec (x:xs) = x*x : squaresRec xs
\]

\[
squaresRec [1,2,3] = squaresRec (1 : (2 : (3 : []))) = 1*1 : squaresRec (2 : (3 : [])) = 1*1 : (2*2 : squaresRec (3 : [])) = 1*1 : (2*2 : (3*3 : squaresRec [])) = 1*1 : (2*2 : (3*3 : [])) = 1 : (4 : (9 : [])) = [1,4,9]
\]
How recursion works—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

-- squares.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

[melchior]dts: ghci squares.hs
GHCi, version 6.8.3: http://www.haskell.org/ghc/ :? for help
*Main> quickCheck prop_squares
+++ OK, passed 100 tests.
*Main>
Part IV

Filtering: Select odd elements from a list
Two styles of definition—odds

Comprehension

\[
\text{odds} :: [\text{Integer}] \rightarrow [\text{Integer}] \\
\text{odds} \ \text{xs} \ = \ [ \ x \mid x \leftarrow \text{xs}, \ \text{isOdd} \ x ]
\]

Recursion

\[
\text{oddsRec} :: [\text{Integer}] \rightarrow [\text{Integer}] \\
\text{oddsRec} \ [] \ = \ [] \\
\text{oddsRec} \ (x:xs) \ | \ \text{isOdd} \ x \ = \ x : \ \text{oddsRec} \ xs \\
| \ \text{otherwise} \ = \ \text{oddsRec} \ xs
\]
Pattern matching and conditionals

Pattern matching with guards

\[
\text{oddsRec} :: [\text{Integer}] \rightarrow [\text{Integer}]
\]

\[
\text{oddsRec} [] = [] \\
\text{oddsRec} (x:xs) \mid \text{isOdd} \ x = x : \text{oddsRec} \ xs \\
\mid \text{otherwise} = \text{oddsRec} \ xs
\]

Conditionals with binding

\[
\text{oddsCond} :: [\text{Integer}] \rightarrow [\text{Integer}]
\]

\[
\text{oddsCond} \ ws = \\
\quad \text{if null} \ ws \ \text{then} \\
\quad \quad [] \\
\quad \text{else} \\
\quad \quad \text{let} \\
\quad \quad \quad \mathit{x} = \text{head} \ ws \\
\quad \quad \quad \mathit{xs} = \text{tail} \ ws \\
\quad \quad \in \\
\quad \quad \quad \text{in} \\
\quad \quad \quad \quad \text{if isOdd} \ \mathit{x} \ \text{then} \\
\quad \quad \quad \quad \quad \mathit{x} : \text{oddsCond} \ \mathit{xs} \\
\quad \quad \quad \quad \text{else} \\
\quad \quad \quad \quad \text{oddsCond} \ \mathit{xs}
\]
How recursion works—oddsRec

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

oddsRec [1,2,3]
How recursion works—oddsRec

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

oddsRec [1,2,3] = oddsRec (1 : (2 : (3 : [])))
```
How recursion works—oddsRec

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

oddsRec [1,2,3] =

```haskell
oddsRec (1 : (2 : (3 : []))
= { x = 1, xs = (2 : (3 : [])), isOdd 1 = True }
1 : oddsRec (2 : (3 : []))
```
How recursion works—oddsRec

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

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

```
= 
  oddsRec (1 : (2 : (3 : [])))
= 
  1 : oddsRec (2 : (3 : []))
= 
  { x = 2, xs = (3 : []), isOdd 2 = False }
  1 : oddsRec (3 : [])
```
How recursion works—oddsRec

```haskell
oddsRec :: [Integer] -> [Integer]
oddsRec [] = []
oddsRec (x:xs) | isOdd 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 = [], isOdd 3 = True }  
  1 : (3 : oddsRec [])
```
How recursion works—oddsRec

oddsRec :: [Integer] -> [Integer]
oddsRec [] = []

oddsRec (x:xs) | isOdd 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 : [])
How recursion works—oddsRec

oddsRec :: [Integer] -> [Integer]
oddsRec [] = []
oddsRec (x:xs) | isOdd 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]
How recursion works—oddsRec

oddsRec :: [Integer] -> [Integer]
oddsRec [] = []
oddsRec (x:xs) | isOdd 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]
QuickCheck

-- odds.hs
import Test.QuickCheck

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

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

prop_odds :: [Integer] -> Bool
prop_odds xs = odds xs == oddsRec xs

[melchior]dts: ghci odds.hs
GHCi, version 6.8.3: http://www.haskell.org/ghc/ :: for help
*Main> quickCheck prop_odds
+++ OK, passed 100 tests.
*Main>
Part V

Accumulation: Sum a list
Sum

\[
\text{sum :: [Integer] -> Integer}
\]
\[
\text{sum []} = 0
\]
\[
\text{sum (x:xs)} = x + \text{sum xs}
\]

\[
\text{sum [1,2,3]}
\]
Sum

\[\text{sum} :: \text{[Integer]} \rightarrow \text{Integer}\]
\[\text{sum} \ [\] \quad = \quad 0\]
\[\text{sum} \ (x:\text{xs}) \quad = \quad x + \text{sum} \ \text{xs}\]

\[\text{sum} \ [1,2,3]\]
\[=\]
\[\text{sum} \ (1 : (2 : (3 : [])))\]
Sum

\[
\text{sum} :: [\text{Integer}] \rightarrow \text{Integer}
\]
\[
\text{sum} \ [\ ] \ = \ 0
\]
\[
\text{sum} \ (x:xs) \ = \ x + \text{sum} \ xs
\]

\[
\text{sum} \ [1,2,3]
\]
\[
= \quad \text{sum} \ (1 : (2 : (3 : [])))
\]
\[
= \quad \{x = 1, \ xs = (2 : (3 : []))\}
\]
\[
1 + \text{sum} \ (2 : (3 : []))
\]
Sum

\[
\begin{align*}
\text{sum} & \::\ [\text{Integer}] \rightarrow \text{Integer} \\
\text{sum}\ [\] & = 0 \\
\text{sum}\ (x:xs) & = x + \text{sum}\ xs \\
\text{sum}\ [1,2,3] & = \text{sum}\ (1 : (2 : (3 : []))) \\
& = 1 + \text{sum}\ (2 : (3 : [])) \\
& = 1 + (2 + \text{sum}\ (3 : [])) \\
& = 1 + (2 + 0) \\
& = 3
\end{align*}
\]
Sum

\[
\text{sum} :: \text{[Integer]} \rightarrow \text{Integer}
\]

\[
\text{sum} \ [\ ] = 0
\]

\[
\text{sum} \ (x : xs) = x + \text{sum} \ xs
\]

\[
\text{sum} \ [1,2,3]
\]

\[
= \text{sum} \ (1 : (2 : (3 : [\ ])))
\]

\[
= 1 + \text{sum} \ (2 : (3 : [\ ]))
\]

\[
= 1 + (2 + \text{sum} \ (3 : [\ ]))
\]

\[
= \{ x = 3, xs = [\] \}
\]

\[
1 + (2 + (3 + \text{sum} \ [\ ]))
\]
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))
Sum

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

\[
\begin{align*}
\text{sum } [1,2,3] & = \text{sum } (1 : (2 : (3 : []))) \\
& = 1 + \text{sum } (2 : (3 : [])) \\
& = 1 + (2 + \text{sum } (3 : [])) \\
& = 1 + (2 + (3 + \text{sum } [])) \\
& = 1 + (2 + (3 + 0)) \\
& = 6
\end{align*}
\]
Sum

\[ \text{sum} :: [\text{Integer}] \to \text{Integer} \]
\[ \text{sum} \; [] \; = \; 0 \]
\[ \text{sum} \; (x:xs) \; = \; x \; + \; \text{sum} \; xs \]

\[ \text{sum} \; [1,2,3] \]
\[ = \]
\[ \text{sum} \; (1 : \; (2 : \; (3 : \; []))) \]
\[ = \]
\[ 1 \; + \; \text{sum} \; (2 : \; (3 : \; [])) \]
\[ = \]
\[ 1 \; + \; (2 \; + \; \text{sum} \; (3 : \; [])) \]
\[ = \]
\[ 1 \; + \; (2 \; + \; (3 \; + \; \text{sum} \; [])) \]
\[ = \]
\[ 1 \; + \; (2 \; + \; (3 \; + \; 0)) \]
\[ = \]
\[ 6 \]
Sum

\[
\text{sum} :: [\text{Integer}] \rightarrow \text{Integer} \\
\text{sum} [] = 0 \\
\text{sum} (x:xs) = x + \text{sum} \ xs
\]

\[
\text{sum} [1,2,3] \\
= \\
\text{sum} (1 : (2 : (3 : [])))) \\
= \\
1 + \text{sum} (2 : (3 : []))) \\
= \\
1 + (2 + \text{sum} (3 : []))) \\
= \\
1 + (2 + (3 + \text{sum} [])) \\
= \\
1 + (2 + (3 + 0)) \\
= 6
\]
Product

\[
\text{product} :: [\text{Integer}] \rightarrow \text{Integer} \\
\text{product} [] = 1 \\
\text{product} (x:xs) = x \ast \text{product} \; xs
\]

\[
\text{product} [1,2,3] \\
= \\
\text{product} (1 : (2 : (3 : []))) \\
= \\
1 \ast \text{product} (2 : (3 : [])) \\
= \\
1 \ast (2 \ast \text{product} (3 : [])) \\
= \\
1 \ast (2 \ast (3 \ast \text{product} [])) \\
= \\
1 \ast (2 \ast (3 \ast 1)) \\
= \\
6
\]
Part VI

Putting it all together:
Sum of the squares of the odd numbers in a list
Two styles of definition

Comprehension

\[
\text{sumSqOdd} :: [\text{Integer}] \rightarrow \text{Integer} \\
\text{sumSqOdd} \ \text{xs} \ = \ \text{sum} \ [ \ x^2 \mid x \leftarrow \text{xs}, \ \text{isOdd} \ x ]
\]

Recursion

\[
\text{sumSqOddRec} :: [\text{Integer}] \rightarrow \text{Integer} \\
\text{sumSqOddRec} \ [\] \ = \ 0 \\
\text{sumSqOddRec} \ (x:xs) \mid \text{isOdd} \ x \ = \ x^2 + \text{sumSqOddRec} \ xs \\
\mid \ \text{otherwise} \ = \ \text{sumSqOddRec} \ xs
\]
How recursion works—sumSqOddRec

sumSqOddRec :: [Integer] -> Integer
sumSqOddRec [] = []
sumSqOddRec (x:xs) | isOdd x = x*x + sumSqOddRec xs
| otherwise = sumSqOddRec xs

sumSqOddRec [1,2,3]
How recursion works—sumSqOddRec

\[
\text{sumSqOddRec} :: [\text{Integer}] \rightarrow \text{Integer} \\
\text{sumSqOddRec} \; [] = [] \\
\text{sumSqOddRec} \; (x:xs) | \text{isOdd} \; x = x \times x + \text{sumSqOddRec} \; xs \\
| \text{otherwise} = \text{sumSqOddRec} \; xs \\
\]

\[
\text{sumSqOddRec} \; [1,2,3] \\
= \\
\text{sumSqOddRec} \; (1 : (2 : (3 : [])))
\]
How recursion works—sumSqOddRec

\[
\text{sumSqOddRec} :: [\text{Integer}] \rightarrow \text{Integer} \\
\text{sumSqOddRec} [] = 0 \\
\text{sumSqOddRec} (x:xs) \mid \text{isOdd } x = x^2 + \text{sumSqOddRec } xs \\
\mid \text{otherwise } = \text{sumSqOddRec } xs
\]

\[
\text{sumSqOddRec } [1,2,3] \\
= \\
\text{sumSqOddRec} (1 : (2 : (3 : []))) \\
= \{ x = 1, xs = (2 : (3 : [])), \text{isOdd } 1 = \text{True } \} \\
1^2 + \text{sumSqOddRec } (2 : (3 : []))
\]
How recursion works—\texttt{sumSqOddRec}

\begin{verbatim}
sumSqOddRec :: [Integer] -> Integer
sumSqOddRec [] = 0
sumSqOddRec (x:xs) | isOdd x = x*x + sumSqOddRec xs
                  | otherwise = sumSqOddRec xs

sumSqOddRec [1,2,3]
  = sumSqOddRec (1 : (2 : (3 : [])))
  = 1*1 + sumSqOddRec (2 : (3 : []))
  = \{ x = 2, xs = (3 : []), isOdd 2 = False \}
  1*1 + sumSqOddRec (3 : [])
\end{verbatim}
How recursion works—sumSqOddRec

sumSqOddRec :: [Integer] -> Integer
sumSqOddRec [] = 0
sumSqOddRec (x:xs) | isOdd 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 = 3, xs = [], isOdd 3 = True }
1*1 + (3*3 : sumSqOddRec [])
How recursion works—sumSqOddRec

\[
\text{sumSqOddRec} :: [\text{Integer}] \rightarrow \text{Integer} \\
\text{sumSqOddRec} \ [\] = 0 \\
\text{sumSqOddRec} \ (x:xs) \mid \text{isOdd} \ x = x \times x + \text{sumSqOddRec} \ xs \\
\mid \text{otherwise} = \text{sumSqOddRec} \ xs
\]

\[
\text{sumSqOddRec} \ [1,2,3] \\
= \\
\text{sumSqOddRec} \ (1 : (2 : (3 : []))) \\
= \\
1 \times 1 + \text{sumSqOddRec} \ (2 : (3 : [])) \\
= \\
1 \times 1 + \text{sumSqOddRec} \ (3 : []) \\
= \\
1 \times 1 + (3 \times 3 + \text{sumSqOddRec} \ []) \\
= \\
1 \times 1 + (3 \times 3 + 0)
\]
How recursion works—sumSqOddRec

sumSqOddRec :: [Integer] -> Integer
sumSqOddRec [] = 0
sumSqOddRec (x:xs) | isOdd 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) =
1 + (9 + 0)
How recursion works—sumSqOddRec

sumSqOddRec :: [Integer] -> Integer
sumSqOddRec [] = 0
sumSqOddRec (x:xs) | isOdd 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)
  =
    1 + (9 + 0)
  =
    10
How recursion works—sumSqOddRec

sumSqOddRec :: [Integer] -> Integer
sumSqOddRec [] = 0
sumSqOddRec (x:xs) | isOdd 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)
= 1 + (9 + 0)
= 10