Informatics 1

Functional Programming Lectures 3 and 4

Monday 5 and Tuesday 6 October 2009

Lists and Recursion

Philip Wadler

University of Edinburgh

Tutorials

Tutorials start this week!

Tuesday/Wednesday	Computation and Logic
Thursday/Friday	Functional Programming

Do the tutorial work *before* the tutorial!

(You do not do the tutorial work *during* the tutorial!)

Bring a *printout* of your work to the tutorial!

Labs and Lab week

Drop-in laboratories

Computer Lab West, Appleton Tower, level 5

Mondays	3–5pm
Tuesdays	2–5pm
Wednesdays	2–5pm
Thursdays	2–5pm
Fridays	3–5pm

Did you do your lab week exercise?

Required text and reading

Haskell: The Craft of Functional Programming, Second Edition,
Simon Thompson, Addison-Wesley, 1999.

Reading assignment:

Thompson, Chapters 1–3 (pp. 1–52)
by Friday 25 September 2009.
Thompson, Chapters 4–5 & 7 (pp. 53–95, 115–134)
by Monday 5 October 2009.
Thompson, Chapters 6 & 8 (pp. 96–114, 135–148)
by Monday 12 October 2009.

List comprehensions — Generators

```
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]` is called a *generator*

`<-` is pronounced *drawn from*

List comprehensions — Guards

```
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` is called a *guard*

Sum, Product

```
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
```

Part I

Mapping: Square every element of a list (comprehension)

How comprehensions work—squares

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

```
  squares [1,2,3]
```

How comprehensions work—squares

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

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

How comprehensions work—squares

```
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 ]
```

How comprehensions work—squares

```
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 ]
```

How comprehensions work—squares

```
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]
```

How comprehensions work—squares

```
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]
```

Part II

Filtering: Select odd elements from a list (comprehension)

How comprehensions work—odds

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

   odds [1,2,3]
```

How comprehensions work—odds

```
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 ]
```

How comprehensions work—odds

```
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 ]
```

How comprehensions work—odds

```
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 ]
```

How comprehensions work—odds

```
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]
```

How comprehensions work—odds

```
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]
```

How comprehensions work—odds

```
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]
```

Part III

Putting it all together:

Sum of the squares of the odd numbers in a list

(comprehension)

Two styles of definition

Composition

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

Comprehension

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

How composition works—f

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

   f [1,2,3]
```

How composition works—f

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

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

How composition works—f

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

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

How composition works—f

```
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]
```

How composition works—f

```
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
```

How composition works—f

```
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
```

How comprehensions work—fCom

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

   fCom [1,2,3]
```

How comprehensions work—fCom

```
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 ]
```

How comprehensions work—fCom

```
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 ])
```

How comprehensions work—fCom

```
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 ])
```

How comprehensions work—fCom

```
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])
```

How comprehensions work—fCom

```
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]
```

How comprehensions work—fCom

```
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
```

How comprehensions work—fCom

```
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, a program

```
-- 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, running the program

```
[culross]wadler: ghci lect03.hs
GHCi, version 6.8.3: http://www.haskell.org/ghc/ :? for help
Loading package base ... linking ... done.
[1 of 1] Compiling Main             ( lect03.hs, interpreted )
*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>
```

Part IV

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!
[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 V

Mapping: Square every element of a list (recursion)

Two styles of definition—squares

Comprehension

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

Recursion

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

Pattern matching and conditionals

Pattern matching

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

Conditionals with binding

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

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—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 : []))
```

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]
```

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, a program

```
-- 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, running a program

```
[culross]wadler: ghci lect03.hs
GHCi, version 6.8.3: http://www.haskell.org/ghc/ :? for help
Loading package base ... linking ... done.
[1 of 1] Compiling Main             ( lect03.hs, interpreted )
*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>
```

Part VI

Filtering: Select odd elements from a list (recursion)

Two styles of definition—odds

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
```

Pattern matching and conditionals

Pattern matching with guards

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

Conditionals with binding

```
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
```

How recursion works—oddsRec

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

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

How recursion works—oddsRec

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

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

How recursion works—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 : []))
```

How recursion works—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 : [])
```

How recursion works—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 [])
```

How recursion works—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 : [])
```

How recursion works—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]
```

How recursion works—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]
```

Part VII

Accumulation: Sum a list

Sum

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

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

Sum

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

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

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 : []))
```

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 : []))
```

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 []))
```

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

    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
```

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
```

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
```

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
```

Part VIII

Putting it all together:

Sum of the squares of the odd numbers in a list

(recursion)

Three styles of definition

Composition

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

Comprehension

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

Recursion

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

How recursion works—fRec

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

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

How recursion works—fRec

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

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

How recursion works—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 : []))
```

How recursion works—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 : [])
```

How recursion works—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 [])
```

How recursion works—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)
```

How recursion works—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)
```

How recursion works—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
```

How recursion works—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
```