Informatics 1

Functional Programming Lectures 3 and 4

Monday 6 and Tuesday 7 October 2006

Lists and Recursion

Philip Wadler

University of Edinburgh

Tutorials

Tutorials start this week!

Tuesday/Wednesday	Computation and Logic
Thursday/Friday	Functional Programming

Enter requests for changes into RT system; or visit ITO.

Do tutorials in advance

Bring printouts to the tutorial

Laboratories

Drop-in laboratories:

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

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 Mon 29 Sep 2008.
Thompson, Chapters 4–5 (pp. 53–95): by Mon 6 Oct 2008.
Thompson, Chapters 6–7 (pp. 96–134): by Mon 13 Oct 2008.

Blackwells has confirmed they will take back textbooks.

Part I

List comprehensions

List comprehensions — Generators

```
Prelude> [ x*x | x <- [0..5] ]
[0,1,4,9,16,25]

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

Prelude [ (x, even x) | x <- [0..5] ]
[(0,True),(1,False),(2,True),(3,False),(4,True),(5,False)]
```

`x <- [0..5]` is called a *generator*

`<-` is pronounced *drawn from*

List comprehensions — Guards

```
Prelude> [ x*x | x <- [0..5], even x ]
[0,4,16]

Prelude> [ x*x | x <- [0..5], x*x < 10 ]
[0,1,4,9]

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

`even x` is called a *guard*

List comprehensions — Multiple generators

```
Prelude> [ (x,y) | x <- [0..2], y <- [0..3] ]
[(0,0),(0,1),(0,2),(0,3),
 (1,0),(1,1),(1,2),(1,3),
 (2,0),(2,1),(2,2),(2,3)]

Prelude> [ (x,y) | x <- [0..2], y <- [0..3], x < y ]
[(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)]

Prelude> [ (x,y) | x <- [0..2], y <- [x+1..3] ]
[(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)]

Prelude> [ (x,y,z) | x <- [1..10],
                     y <- [x+1..10],
                     z <- [1..10],
                     x*x + y*y == z*z]
[(3,4,5),(6,8,10)]
```

Sum, Product

```
Prelude> sum [1,2,3,4]
10

Prelude> sum []
0

Prelude> sum [ x*x | x <- [1,2,3,4] ]
30

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

Prelude> product []
1

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

Part II

Lists and Recursion

Lists

A list is either

- *null*, written `[]`, or
- *constructed*, written `x:xs`, with *head* `x` (an element), and *tail* `xs` (a list).

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

[1,2,3]   =   1 : [2,3]      -- head is 1, tail is [2,3]
[2,3]     =   2 : [3]        -- head is 2, tail is [3]
[3]       =   3 : []         -- head is 3, tail is []
[]                           -- null

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

"list"    =   'l' : "ist"    -- head is 'l', tail is "ist"
"ist"     =   'i' : "st"     -- head is 'i', tail is "st"
"st"      =   's' : "t"      -- head is 's', tail is "t"
"t"       =   't' : ""       -- head is 't', tail is ""
""                           -- null
```

Cons and append

Operator `(:)` is pronounced *cons*, for *construct*.

Operator `(++)` is pronounced *append*.

```
(:)   :: 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"
"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!
```

Cons takes an element and a list.

Append takes two lists.

Two styles of definition—squares

```
Main*> squares [1,2,3]
[1,4,9]
```

Comprehension

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

Recursion

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

How recursion works—squares

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

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

How recursion works—squares

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

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

How recursion works—squares

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

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

How recursion works—squares

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

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

How recursion works—squares

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

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

How recursion works—squares

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

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

How recursion works—squares

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

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

How recursion works—squares

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

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

How recursion works—squares

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

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

Two styles of definition—odds

```
Main*> odds [1,2,3]
[1,3]
```

Comprehension

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

Recursion

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

How recursion works—odds

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

```
   odds [1,2,3]
```

How recursion works—odds

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

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

How recursion works—odds

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

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

How recursion works—odds

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

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

How recursion works—odds

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

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

How recursion works—odds

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

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

How recursion works—odds

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

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

How recursion works—odds

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

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

Two styles of definition—oddSquares

```
Main*> oddSquares [1,2,3]
[1,9]
```

Comprehension

Recursion

Two styles of definition—oddSquares

```
Main*> oddSquares [1,2,3]
[1,9]
```

Comprehension

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

Recursion

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

How recursion works—oddSquares

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

```
   oddSquares [1,2,3]
```

How recursion works—oddSquares

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

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

How recursion works—oddSquares

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

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

How recursion works—oddSquares

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

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

How recursion works—oddSquares

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

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

How recursion works—oddSquares

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

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

How recursion works—oddSquares

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

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

How recursion works—oddSquares

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

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

How recursion works—oddSquares

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

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

Definition by pattern matching

```
null :: [a] -> Bool
null []        =  True
null (x:xs)    =  False

head :: [a] -> a
head (x:xs)    =  x

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

How definition by pattern matching works—null

```
null :: [a] -> Bool
null []        =  True
null (x:xs)    =  False
```

```
   null []
```

How definition by pattern matching works—null

```
null :: [a] -> Bool
null []        =  True
null (x:xs)    =  False

   null []
=
   True
```

How definition by pattern matching works—null

```
null :: [a] -> Bool
null []        =  True
null (x:xs)    =  False

   null []
=
   True
```

How definition by pattern matching works—null

```
null :: [a] -> Bool
null []         =  True
null (x:xs)     =  False
```

```
   null [1,2,3]
```

How definition by pattern matching works—null

```
null :: [a] -> Bool
null []         =  True
null (x:xs)     =  False

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

How definition by pattern matching works—null

```
null :: [a] -> Bool
null []         =  True
null (x:xs)     =  False

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

How definition by pattern matching works—null

```
null :: [a] -> Bool
null []          =  True
null (x:xs)      =  False

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

How definition by pattern matching works—head

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

   head [1,2,3]
```

How definition by pattern matching works—head

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

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

How definition by pattern matching works—head

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

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

How definition by pattern matching works—head

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

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