Informatics 1

Functional Programming Lectures 5 and 6

Monday 13 and Tuesday 14 October 2008

More fun with recursion

Philip Wadler

University of Edinburgh

Tutorials

Tutorial 2 due 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

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.
Thompson, Chapters 8–9 (pp. 135–166): by Mon 20 Oct 2008.

Part I

Booleans and characters

Boolean operators

```
not :: Bool -> Bool
(&&), (||) :: Bool -> Bool -> Bool

not False  =  True
not True   =  False

False && False  =  False
False && True   =  False
True  && False  =  False
True  && True   =  True

False || False  =  False
False || True   =  True
True  || False  =  True
True  || True   =  True
```

Defining operations on characters

```
isLower :: Char -> Bool
isLower x  =  'a' <= x && x <= 'z'

isUpper :: Char -> Bool
isUpper x  =  'A' <= x && x <= 'Z'

isDigit :: Char -> Bool
isDigit x  =  '0' <= x && x <= '9'

isAlpha :: Char -> Bool
isAlpha x  =  isLower x || isUpper x
```

Defining operations on characters

```
digitToInt :: Char -> Int
digitToInt c | isDigit c  =  ord c - ord '0'

intToDigit :: Int -> Char
intToDigit d | 0 <= d && d <= 9  =  chr (ord '0' + d)

toLower :: Char -> Char
toLower c | isUpper c  =  chr (ord c - ord 'A' + ord 'a')
          | otherwise  =  c

toUpper :: Char -> Char
toUpper c | isLower c  =  chr (ord c - ord 'a' + ord 'A')
          | otherwise  =  c
```

Part II

Conditionals

Conditional equations

```
max :: Integer -> Integer -> Integer
max x y | x >= y    =  x
        | y >= x    =  y

max3 :: Integer -> Integer -> Integer -> Integer
max3 x y z | x >= y && x >= z  =  x
           | y >= x && y >= z  =  y
           | z >= x && z >= y  =  z
```

Conditional equations with otherwise

```
max :: Integer -> Integer -> Integer
max x y | x >= y      =  x
        | otherwise   =  y

max3 :: Integer -> Integer -> Integer -> Integer
max3 x y z | x >= y && x >= z  =  x
           | y >= x && y >= z  =  y
           | otherwise         =  z
```

Conditional equations with otherwise

```
max :: Integer -> Integer -> Integer
max x y | x >= y      =  x
        | otherwise   =  y

max3 :: Integer -> Integer -> Integer -> Integer
max3 x y z | x >= y && x >= z  =  x
           | y >= x && y >= z  =  y
           | otherwise         =  z

otherwise :: Bool
otherwise =  True
```

Conditional expressions

```
max :: Integer -> Integer -> Integer
max x y  =  if x >= y then x else y

max3 :: Integer -> Integer -> Integer -> Integer
max3 x y z  =  if x >= y && x >= z then x
               else if y >= x && y >= z then y
               else z
```

Another way to define max3

```
max3 :: Integer -> Integer -> Integer -> Integer
max3 x y z  =  if x >= y then
                  if x >= z then x else z
               else
                  if y >= z then y else z
```

A better way to define max3

```
max3 :: Integer -> Integer -> Integer -> Integer
max3 x y z  =  max (max x y) z
```

An even better way to define max3

```
max3 :: Integer -> Integer -> Integer -> Integer
max3 x y z  =  x `max` y `max` z
```

An even better way to define max3

```
max3 :: Integer -> Integer -> Integer -> Integer
max3 x y z  =  x `max` y `max` z

max :: Integer -> Integer -> Integer
x `max` y | x >= y     =  x
          | otherwise  =  y
```

`x + y` *stands for* `(+) x y`

`x >= y` *stands for* `(>=) x y`

`` x `max` y `` *stands for* `max x y`

Associativity

```
prop_max_assoc :: Integer -> Integer -> Integer -> Bool
prop_max_assoc x y z  =
  (x `max` y) `max` z  ==  x `max` (y `max` z)
```

When using an infix function that is not associative, best to write in all the parentheses!

Why we use infix notation

```
prop_max_assoc :: Integer -> Integer -> Integer -> Bool
prop_max_assoc x y z  =
  max (max x y) z  ==  max x (max y z)
```

This is much harder to read than infix notation!

Part III

Sum and Product

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
```

Odd squares

```
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]
=
   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]
```

Sum odd squares

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

Comprehension and library function

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

Recursion

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

How recursion works—sumOddSquares

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

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

Upto

```
Prelude upto 1 3
[1,2,3]
```

Library function

```
upto :: Integer -> Integer -> [Integer]
upto m n   =   [m..n]
```

Recursion

```
upto :: Integer -> Integer -> [Integer]
upto m n | m > n     =   []
         | m <= n    =   m : upto (m+1) n
```

How recursion works—upto

```
upto :: Integer -> Integer -> [Integer]
upto m n | m > n     =  []
         | m <= n    =  m : upto (m+1) n

   upto 1 3
=
   1 : upto 2 3
=
   1 : (2 : upto 3 3)
=
   1 : (2 : (3 : upto 4 3))
=
   1 : (2 : (3 : []))
=
   [1,2,3]
```

Factorial

```
Main*> factorial 3
```

Library functions

```
factorial :: Integer -> Integer
factorial n  =  product [1..n]
```

Recursion

```
factorial :: Integer -> Integer
factorial n  =  fact 1 n
  where
  fact :: Integer -> Integer -> Integer
  fact m n | m > n     =  1
           | m <= n    =  m * fact (m+1) n
```

How recursion works—factorial

```
factorial :: Integer -> Integer
factorial n  =  fact 1 n
  where
  fact :: Integer -> Integer -> Integer
  fact m n | m > n     =  1
           | m <= n    =  m * fact (m+1) n

   factorial 3
=
   fact 1 3
=
   1 * fact 2 3
=
   1 * (2 * fact 3 3)
=
   1 * (2 * (3 * fact 4 3))
=
   1 * (2 * (3 * 1))
=
   6
```

Part IV

Append, zip, search

Append

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

   "abc" ++ "de"
=
   ('a' : ('b' : ('c' : []))) ++ ('d' : ('e' : []))
=
   'a' : (('b' : ('c' : [])) ++ ('d' : ('e' : [])))
=
   'a' : ('b' : (('c' : []) ++ ('d' : ('e' : []))))
=
   'a' : ('b' : ('c' : ([] ++ ('d' : ('e' : [])))))
=
   'a' : ('b' : ('c' : ('d' : ('e' : []))))
=
   "abcde"
```

Zip

```
zip :: [a] -> [b] -> [(a,b)]
zip [] ys             =  []
zip (x:xs) []         =  []
zip (x:xs) (y:ys)     =  (x,y) : zip xs ys

    zip [0,1,2] "abc"
=
    zip (0 : (1 : (2: []))) ('a' : ('b' : ('c' : [])))
=
    (0,'a') : zip (1 : (2 : [])) ('b' : ('c' : []))
=
    (0,'a') : ((1,'b') : zip (2 : []) ('c' : []))
=
    (0,'a') : ((1,'b') : ((2,'c') : zip [] []))
=
    (0,'a') : ((1,'b') : ((2,'c') : []))
=
    [(0,'a'),(1,'b'),(2,'c')]
```

Two definitions of zip

Shorter

```
zip :: [a] -> [b] -> [(a,b)]
zip [] ys           =  []
zip (x:xs) []       =  []
zip (x:xs) (y:ys)   =  (x,y) : zip xs ys
```

Longer

```
zip :: [a] -> [b] -> [(a,b)]
zip [] []           =  []
zip [] (y:ys)       =  []
zip (x:xs) []       =  []
zip (x:xs) (y:ys)   =  (x,y) : zip xs ys
```

More fun with zip

```
Prelude> zip [0,1,2] "abc"
[(0,'a'),(1,'b'),(2,'c')]

Prelude> zip [0,1,2] "abcde"
[(0,'a'),(1,'b'),(2,'c')]

Prelude> zip [0,1,2,3,4] "abc"
[(0,'a'),(1,'b'),(2,'c')]

Prelude> zip [0..] "words"
[(0,'w'),(1,'o'),(2,'r'),(3,'d'),(4,'s')]

Prelude> let pairs xs = zip xs (tail xs)
Prelude> pairs "words"
[('w','o'),('o','r'),('r','d'),('d','s')]
```

Zip with an infinite list

```
zip :: [a] -> [b] -> [(a,b)]
zip [] ys           =  []
zip (x:xs) []       =  []
zip (x:xs) (y:ys)   =  (x,y) : zip xs ys

   zip [0..] "abc"
=
   zip [0..] ('a' : ('b' : ('c' : [])))
=
   (0,'a') : zip [1..] ('b' : ('c' : []))
=
   (0,'a') : ((1,'b') : zip [2..] ('c' : []))
=
   (0,'a') : ((1,'b') : ((2,'c') : zip [3..] []))
=
   (0,'a') : ((1,'b') : ((2,'c') : []))
=
   [(0,'a'),(1,'b'),(2,'c')]
```

Search

```
Main*> search "bookshop" 'o'
[1,2,6]
```

Comprehensions and library functions

```
search :: [a] -> a -> [Int]
search xs y = [ i | (i,x) <- zip [0..] xs, x==y ]
```

Recursion

```
search :: [a] -> a -> [Int]
search xs y  =  search' xs y 0
  where
  search' :: [a] -> a -> Int -> [Int]
  search' [] y i        =  []
  search' (x:xs) y i
    | x == y            =  i : search' xs y (i+1)
    | otherwise         =  search' xs y (i+1)
```

How search works

```
search :: [a] -> a -> [Int]
search xs y  =  search' xs y 0
  where
  search' :: [a] -> a -> Int -> [Int]
  search' [] y i  =  []
  search' (x:xs) y i
    | x == y       =  i : search' xs y (i+1)
    | otherwise    =  search' xs y (i+1)
```

```
   search "book" 'o'
=
   search' ('b': ('o' : ('o' : ('k' : []))))  'o'  0
=
   search' ('o' : ('o' : ('k' : []))) 'o' 1
=
   1 : search' ('o' : ('k' : [])) 'o' 2
=
   1 : (2 : search' ('k' : []) 'o' 3)
=
   1 : (2 : search' [] 'o' 4)
=
   1 : (2 : [])
=
   [1,2]
```

Part V

Select, take, and drop

Select, take, and drop

```
Prelude> "words" !! 3
'd'

Prelude> take 3 "words"
"wor"

Prelude> drop 3 "words"
"ds"
```

Select

```
Prelude> "words" !! 3
'd'
```

Comprehension

```
(!!) :: [a] -> Int -> a
xs !! n  =  the [ x | (i,x) <- zip [0..] xs, i == n ]
            where
            the [x]  =  x
```

Recursion

```
(x:xs) !! 0          =  x
(x:xs) !! n | n > 0  =  xs !! (n-1)
```

Take

```
Prelude> take 3 "words"
"wor"
```

Comprehension

```
take :: Int -> [a] -> [a]
take n xs   =   [ x | (i,x) <- zip [0..] xs, i < n ]
```

Recursion

```
take :: Int -> [a] -> [a]
take 0 xs                  =   []
take n []       | n > 0    =   []
take n (x:xs) | n > 0    =   x : take (n-1) xs
```

How take works

```
take :: Int -> [a] -> [a]
take 0 xs                 =  []
take n []      | n > 0    =  []
take n (x:xs)  | n > 0    =  x : take (n-1) xs

   take 3 "word"
=
   take 3 ('w':('o':('r':('d':('s':[])))))
=
   'w' : take 2 ('o':('r':('d':('s':[]))))
=
   'w' : ('o' : take 1 ('r':('d':('s':[]))))
=
   'w' : ('o' : ('r' : (take 0 ('d':('s':[])))))
=
   'w' : ('o' : ('r' : []))
=
   "wor"
```

Natural numbers

A natural number is

- Zero, written `0`, or
- The successor of a natural number, written `n+1`, where `n` is a natural number

```
= ((0+1)+1)+1

= 2+1
= 1+1
= 0+1
```

Select and take, two ways

Guards

```
(x:xs) !! 0               =  0
(x:xs) !! n    | n > 0    =  xs !! (n-1)

take 0 xs                 =  []
take n []      | n > 0    =  []
take n (x:xs)  | n > 0    =  x : take (n-1) xs
```

$n + 1$ patterns

```
(x:xs) !! 0          =  x
(x:xs) !! (n+1)      =  xs !! n

take 0 xs            =  []
take (n+1) []        =  []
take (n+1) (x:xs)    =  x : take n xs
```

How take works, reprise

```
take :: Int -> [a] -> [a]
take 0 xs           =  []
take (n+1) []       =  []
take (n+1) (x:xs)   =  x : take n xs

   take 3 "words"
=
   take (((0+1)+1)+1) ('w':('o':('r':('d':('s':[])))))
=
   'w' : take ((0+1)+1) ('o':('r':('d':('s':[]))))
=
   'w' : ('o' : take (0+1) ('r':('d':('s':[]))))
=
   'w' : ('o' : ('r' : (take 0 ('d':('s':[])))))
=
   'w' : ('o' : ('r' : []))
=
   "wor"
```

Arithmetic

```
(+) :: Integer -> Integer -> Integer
m + 0       =  m
m + (n+1)   =  (m + n) + 1

(*) :: Integer -> Integer -> Integer
m * 0       =  0
m * (n+1)   =  (m * n) + m

(^) :: Integer -> Integer -> Integer
m ^ 0       =  1
m ^ (n+1)   =  (m ^ n) * m
```