Informatics 1

Functional Programming Lectures 5 and 6

Monday 12 and Tuesday 13 October 2009

More fun with 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!

Laboratories

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

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.

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.

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 and Associativity

Conditional equations

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

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

Conditional equations with otherwise

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

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

Conditional equations with otherwise

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

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

otherwise :: Bool
otherwise =  True
```

Conditional expressions

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

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

Key points about conditionals

- As always: write your program in a form that is easy to read. Don't worry (yet) about efficiency: premature optimization is the root of much evil.
- Conditionals are your friend: without them, programs could do very little that is interesting.
- Conditionals are your enemy: each conditional doubles the number of test cases you must consider. A program with five two-way conditionals requires $2^5 = 32$ test cases to try every path through the program. A program with ten two-way conditionals requires $2^{10} = 1024$ test cases.

A better way to define max3

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

An even better way to define max3

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

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

An even better way to define max3

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

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

It doesn't matter where the parentheses go with an associative operator, so we often omit them.

Associativity

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

It doesn't matter where the parentheses go with an associative operator, so we often omit them.

Why we use infix notation

```
prop_max_assoc :: Int -> Int -> Int -> 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!

Key points about associativity

- There are a few key properties about operators: *associativity*, *identity*, *commutativity*, *distributivity*, *zero*, *idempotence*. You should know and understand these properties.
- When you meet a new operator, the first question you should ask is “Is it associative?” (The second is “Does it have an identity?”)
- Associativity is our friend, because it means we don’t need to worry about parentheses. The program is easier to read.
- Associativity is our friend, because it is key to writing programs that run twice as fast on dual-core machines, and a thousand times as fast on machines with a thousand cores. We will study this later in the course.

Part III

Append

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"
```

Append

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

  "abc" ++ "de"
=
  'a' : ("bc" ++ "de")
=
  'a' : ('b' : ("c" ++ "de"))
=
  'a' : ('b' : ('c' : ("" ++ "de")))
=
  'a' : ('b' : ('c' : "de"))
=
  "abcde"
```

Properties of append

```
prop_append_assoc :: [Int] -> [Int] -> [Int] -> Bool
prop_append_assoc xs ys zs  =
  (xs ++ ys) ++ zs  ==  xs ++ (ys ++ zs)

prop_append_ident :: [Int] -> Bool
prop_append_ident xs  =
  xs ++ [] == xs  &&  xs == [] ++ xs

prop_append_cons :: Int -> [Int] -> Bool
prop_append_cons x xs  =
  [x] ++ xs  ==  x : xs
```

Part IV

Counting

Counting

```
Prelude [1..3]
[1,2,3]
Prelude enumFromTo 1 3
[1,2,3]
```

Recursion

```
enumFromTo :: Int -> Int -> [Int]
enumFromTo m n | m > n     =  []
               | m <= n    =  m : enumFromTo (m+1) n
```

How enumFromTo works (recursion)

```
enumFromTo :: Int -> Int -> [Int]
enumFromTo m n | m > n     =  []
               | m <= n    =  m : enumFromTo (m+1) n

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

Factorial

```
Main*> factorial 3
```

Library functions

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

Recursion

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

How factorial works (recursion)

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

```
   factorialRec 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 V

Zip and search

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"
=
  (0,'a') : zip [1,2] "bc"
=
  (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 equivalent 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
```

Two alternative definitions of zip

Liberal

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

Conservative

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

Lists of different lengths

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

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

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

Prelude> zipHarsh [0,1,2] "abcde"
error

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

Prelude> zipHarsh [0,1,2,3,4] "abc"
error
```

More fun with zip

```
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

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

How search works (comprehension)

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

  search "book" 'o'
=
  [ i | (i,x) <- zip [0..] "book", x=='o' ]
=
  [ i | (i,x) <- [(0,'b'),(1,'o'),(2,'o'),(3,'k')], x=='o' ]
=
  [0|'b'=='o']++[1|'o'=='o']++[2|'o'=='o']++[3|'k'=='o']
=
  []++[1]++[2]++[]
=
  [1,2]
```

How search works (recursion)

```
searchRec xs y  =  srch xs y 0
  where
  srch [] y i                        =  []
  srch (x:xs) y i  | x == y          =  i : srch xs y (i+1)
                   | otherwise       =  srch xs y (i+1)

  searchRec "book" 'o'
=
  srch "book" 'o' 0
=
  srch "ook" 'o' 1
=
  1 : srch "ok" 'o' 2
=
  1 : (2 : srch "ok" 'o' 3)
=
  1 : (2 : srch "" 'o' 4)
=
  1 : (2 : [])
=
  [1,2]
```

Part VI

Select, take, and drop

Select, take, and drop

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

Prelude> take 3 "words"
"wor"

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

Select, take, and drop (comprehensions)

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

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

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

Select, take, and drop (recursion)

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

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

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

How take works (comprehension)

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

   take 3 "words"
=
   [ x | (j,x) <- zip [0..] "words", j < 3 ]
=
   [ x | (j,x) <- [(0,'w'),(1,'o'),(2,'r'),(3,'d'),(4,'s')],
         j < 3 ]
=
   ['w'|0<3]++['o'|1<3]++['r'|2<3]++['d'|3<3]++['s'|4<3]
=
   ['w']++['o']++['r']++[]++[]
=
   "wor"
```

How take works (recursion)

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

  take 3 "words"
=
  'w' : take 2 "ords"
=
  'w' : ('o' : take 1 "rds")
=
  'w' : ('o' : ('r' : take 0 "rds"))
=
  'w' : ('o' : ('r' : []))
=
  "wor"
```

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).

Natural numbers

Every natural number can be written using only `(+1)` and `0`.

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

A *recursive* definition: A *natural number* is either

- *zero*, written `0`, or
- *successor*, written `n+1`
  with *predecessor* `n` (a natural number).

Select, take, and drop (recursion)

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

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

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

Select, take, and drop ($n + 1$ patterns)

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

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

drop :: Int -> [a] -> [a]
drop 0 xs             =  xs
drop (i+1) (x:xs)     =  drop i 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

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

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

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