Informatics 1

Functional Programming Lecture 4

More fun with recursion

Don Sannella

University of Edinburgh

Part I

Counting

Counting

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

`[m..n]` *stands for* `enumFromTo m n`

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]
```

How enumFromTo works (recursion)

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

  enumFromTo 1 0
=
  []
```

Factorial

```
> 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
```

Counting forever!

```
> [0..]
[0,1,2,3,4,5,...
> enumFrom 0
[0,1,2,3,4,5,...
```

`[m..]` *stands for* `enumFrom m`

Recursion

```
enumFrom :: Int -> [Int]
enumFrom m = m : enumFrom (m+1)
```

How enumFrom works (recursion)

```
enumFrom :: Int -> [Int]
enumFrom m  =  m : enumFrom (m+1)

  enumFrom 0
=
  0 : enumFrom 1
=
  0 : (1 : enumFrom 2)
=
  0 : (1 : (2 : enumFrom 3))
=
  ...
=
  [0,1,2,...   -- computation goes on forever!
```

Part II

Zip and search

Zip

```
zip :: [a] -> [b] -> [(a,b)]
zip [] ys           =  []
zip 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 alternative definitions of zip

Laid back

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

Uptight

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

Zip with lists of different lengths

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

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

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

> zipHarsh [0,1,2] "abcde"
[(0,'a'),(1,'b'),(2,'c')*** Exception:
Non-exhaustive patterns in function zipHarsh

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

> zipHarsh [0,1,2,3,4] "abc"
[(0,'a'),(1,'b'),(2,'c')*** Exception:
Non-exhaustive patterns in function zipHarsh
```

More fun with zip

```
> zip [0..] "word"
[(0,'w'),(1,'o'),(2,'r'),(3,'d')]

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

Zip with an infinite list

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

```
   zip [0..] "abc"
=
   (0,'a') : zip [1..] "bc"
=
   (0,'a') : ((1,'b') : zip [2..] "c")
=
   (0,'a') : ((1,'b') : ((2,'c') : zip [3..] ""))
=
   (0,'a') : ((1,'b') : ((2,'c') : zip (3 : [4..]) ""))
=
   (0,'a') : ((1,'b') : ((2,'c') : []))
=
   [(0,'a'),(1,'b'),(2,'c')]
```

Computer can determine `(3 : [4..])` $\neq$ `[]` without computing `[4..]`.

Dot product of two lists

Comprehensions and library functions

```
dot :: Num a => [a] -> [a] -> a
dot xs ys  =  sum [ x*y | (x,y) <- zipHarsh xs ys ]
```

Recursion

```
dotRec :: Num a => [a] -> [a] -> a
dotRec [] []              =  0
dotRec (x:xs) (y:ys)  =  x*y + dotRec xs ys
```

How dot product works (comprehension)

```
dot :: Num a => [a] -> [a] -> a
dot xs ys  =  sum [ x*y | (x,y) <- zip xs ys ]

  dot [2,3,4] [5,6,7]
=
  sum [ x*y | (x,y) <- zip [2,3,4] [5,6,7] ]
=
  sum [ x*y | (x,y) <- [(2,5), (3,6), (4,7)] ]
=
  sum [ 2*5, 3*6, 4*7 ]
=
  sum [ 10, 18, 28 ]
=
  56
```

How dot product works (recursion)

```
dotRec :: Num a => [a] -> [a] -> a
dotRec [] []            =  0
dotRec (x:xs) (y:ys)    =  x*y + dotRec xs ys

  dotRec [2,3,4] [5,6,7]
=
  dotRec (2:(3:(4:[]))) (5:(6:(7:[])))
=
  2*5 + dotRec (3:(4:[])) (6:(7:[]))
=
  2*5 + (3*6 + dotRec (4:[]) (7:[]))
=
  2*5 + (3*6 + (4*7 + dotRec [] []))
=
  2*5 + (3*6 + (4*7 + 0))
=
  10 + (18 + (28 + 0))
=
  56
```

Search

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

Comprehensions and library functions

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

Recursion

```
searchRec :: Eq a => [a] -> a -> [Int]
searchRec xs y  =  srch xs y 0
  where
  srch :: Eq a => [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 :: Eq a => [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 "k" 'o' 3)
=
  1 : (2 : srch "" 'o' 4)
=
  1 : (2 : [])
=
  [1,2]
```