Informatics 1

Functional Programming Lecture 6

Monday 29 September 2014

Even more fun with recursion

Don Sannella

University of Edinburgh

Part I

Counting

Counting

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

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
```

Counting forever!

```
Prelude [0..]
[0,1,2,3,4,5,...
Prelude 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

Liberal

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

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

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

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 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

```
Main*> 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]
```

Part III

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)

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

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

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

How take works (comprehension)

```
takeComp :: Int -> [a] -> [a]
takeComp 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"
```

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`.

```
3  =  ((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)

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

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

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

Pattern matching and conditionals (squares)

Pattern matching

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

Conditionals with binding

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

Pattern matching and conditionals (take)

Pattern matching

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

Conditionals with binding

```
take :: Int -> [a] -> [a]
take i ws
  if i == 0 || null ws then
    []
  else
    let
      x  = head ws
      xs = tail ws
    in
      x : take (i-1) xs
```

Pattern matching and guards (take)

Pattern matching

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

Guards

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

How take works (recursion)

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

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

The infinite case

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

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

Prelude> take 3 [10..]
[10,11,12]

Prelude> takeComp 3 [10..]
[10,11,12   -- computation goes on forever!
```

The infinite case explained

Function `takeComp` is equivalent to `takeCompRec`.

```
takeCompRec :: Int -> [a] -> [a]
takeCompRec i xs  =  helper 0 i xs
  where
  helper j i []                        = []
  helper j i (x:xs) | j > i            =  x : helper (j+1) i xs
                    | otherwise        =  helper (j+1) i xs

  takeCompRec 3 [10..]
=
  helper 0 3 [10..]
=
  10 : helper 1 3 [11..]
=
  10 : (11 : helper 2 3 [12..])
=
  10 : (11 : (12 : helper 3 3 [13..]))
=
  10 : (11 : (12 : helper 4 3 [14..]))
= ...
```

Part IV

Arithmetic

Arithmetic (recursion)

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

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

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

How arithmetic works (recursion)

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

   2 + 3
=
   (2 + 2) + 1
=
   ((2 + 1) + 1) + 1
=
   (((2 + 0) + 1) + 1) + 1
=
   ((2 + 1) + 1) + 1
=
   5
```

Giuseppe Peano (1858–1932)

The definition of the natural numbers is named the *Peano axioms* in his honour.
Made key contributions to the modern treatment of mathematical induction.