Informatics 1

Functional Programming Lecture 6

More fun with recursion

Don Sannella

University of Edinburgh

Part I

Counting

Now we'll look at functional programming techniques used to solve certain kinds of problems that arise frequently. On the way we'll see some new patterns of recursion.

Counting

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

Recall this notation for the list of numbers from 1 to 3.

Everything in Haskell is a function, and this is the function that is used to give the result of [m..n]

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

Recursion

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

Recursion again, but on numbers this time - counting up from m to 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]
```

Here's how it works.
Before, the list was getting smaller on each recursive call.
Now, the number is getting larger - what's going on?
The point is that the DIFFERENCE between m and n is getting smaller.
It's essential that something gets smaller - otherwise the recursion will never terminate.

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
```

We use a HELPER FUNCTION
to compute the product of [m..n]

This is the same pattern as for enumFromTo,
using * instead of :
(There are simpler definitions of factorial, without
a helper function - try it!)

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
```

Here's how it works.k

Counting forever!

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

Haskell represents the head and tail of a list as unevaluated EXPRESSIONS.
Evaluation happens when you need the value, not before.
This is called LAZY EVALUATION.
The alternative is called EAGER EVALUATION.

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

Recursion

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

Lazy evaluation makes it possible to operate on INFINITE DATA STRUCTURES provided you never need all of the values - just enough to compute some result.

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: like a zipper, but not interleaving the "teeth".
Defined using simultaneous recursion on both lists.

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

If the two lists are different lengths,
it truncates the longer one.

```
  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')]
```

Here's how it works.
Processing two lists, possibly of different types, in lock step, returning a list of pairs.

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
```

The "conservative" version only works if both lists are the same length.
Otherwise it gives an error.
The liberal version is the one that is built into Haskell.

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')]
```

The way that zip treats lists of different lengths is very convenient - it allows us to play tricks like these.

The first example pairs characters with their positions in the list, counting from 0.
It treats [0..] as if it is [0..(length "words" - 1)] without needing to produce that list explicitly.

The second example is a useful trick for when you want to relate successive elements of a list.
For example, counting the number of doubled letters in a string could be done this way.

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..]) ≠ []` 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
```

This example is from linear algebra - dot product is also known as "scalar product".
Given two vectors of equal length, it gives the sum of the product of corresponding components.

"Num a" in these types means that these functions only work when a is a numerical type - will be explained later.

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]
```

Return a list of all of the positions that a character occurs in a string.

Comprehensions and library functions

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

Recursion

"Eq a" in the type means that equality needs to work on the type a.

```
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)
```

Search is easy to define with comprehension using zip: return every i where (i,x) is drawn from zip [0..] xs and x==y.
Recursively: use a helper function. i is the index of the position of the start of the list we're searching through.

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]
```