Prog & Alg I (COMS10002)
Week 10 – Tuples and Lists

Dr. Oliver Ray
Department of Computer Science
University of Bristol

Monday 1st December, 2014

Some (not so) Basic Notions

- Tuples: A tuple is denoted `(a,b,c)` and the functions `fst` and `snd` return the 1st and 2nd elements
- Lists: A list is denoted `[a,b,c]` or `[]` if empty. A comprehension of the form `[f e| e<-L]` forms a new list by applying function f to each element e of an old list L
- Identity function: this simply maps each input to itself. In Haskell it is denoted `id`
- Function composition: in maths f∘g denotes the composition of two functions f and g such that (f∘g)(x) = f(g(x)). In Haskell this is represented `f.g`
- Lambda notation: in maths λx.f(x) is an anonymous function that maps input x to output f(x). Thus λx.2x represents a function that doubles its input. In Haskell this is represented `\x->2*x`

Example: Scaling a Point

```
--  define a Point as a pair of integers
type Point = (Int,Int)


--  define the point we call the origin
origin :: Point
origin = (0,0)


-- define a function for scaling a point
-- (horizontally by h and vertically by v)
scale :: Int -> Int -> Point -> Point
scale h v (x,y) = (h*x,v*y)
```

```
*Main> scale 2 3 (2,1)
(4,3)
```

Example: Flipping a Point

```
-- flip a point about the  horizontal axis (y=0)
flipH :: Point -> Point
flipH (x,y) = (x,-y)

-- flip a point about the  vertical axis (x=0)
flipV :: Point -> Point
flipV (x,y) = (-x,y)

-- flip a point about the diagonal line (x=y)
flipD :: Point -> Point
flipD (x,y) = (y,x)
```

```
-- alternatively (using partial evaluation):
flipH = scale 1 (-1)
flipV = scale (-1) 1
```

Example: Transforming an Image

```
-- quarter turn clockwise
turnC :: Point -> Point
turnC (x,y) = (y,-x)

-- half turn
turnB :: Point -> Point
turnB (x,y) = (-x,-y)

-- quarter turn anticlockwise
turnA :: Point -> Point
turnA (x,y) = (-y,x)
```

```
-- alternatively (using function composition):
turnB = turnC.turnC = flipV.flipH
turnA = turnC.turnB = turnB.turnC = flipV.flipD
```

Example: Transforming an Image

```
-- define an Image as a list of points
type Image = [Point]


-- define a T-shaped image
t :: Image
t=[(0,1),(1,0),(1,1),(2,1)]


-- transform an Image point by point
pointwise :: (Point->Point) -> (Image->Image)
pointwise f = \ps -> [f p | p <- ps]
```

```
*Main> pointwise turnC t
[(1,0),(0,-1),(1,-1),(1,-2)]
```

Example: Overlaying Images

```
-- overlay two images
overlay :: Image -> Image -> Image
overlay i j = i ++ j
```

```
*Main> overlay [origin] t
[(0,0),(0,1),(1,0),(1,1),(2,1)]
```

Lists (p.77,91)

- As we saw with our images, lists are easily used by enclosing types or data within square brackets [...]

  e.g. [Int] denotes a list of integers such as [1,2,3]
- Lists have many useful built-in and library functions

e.g.	[]	denotes the empty list	[1,2,3] ≠ []
	:	takes the head item	[1,2,3] = (1:[2,3])
	!!	gives the n(+1)'th item	[1,2,3]!!2=3
	++	concatenates two lists	[1,2,3] = [1]++[2,3]

- There is a convenient notation for lists of ordered types

e.g.	[1,3..8] = [1,3,5,7]	step can be negative
	['a'..'d'] = "abcd"	a String has type [Char]

- Note that order and repetition of items is important!

Lists Comprehensions (p.79-81, 344-6)

- A list comprehension is a convenient way to build a new list from one or more existing lists
- It consists of an expression that combines the results of one or more generators (each of which takes successive values from a given list) and zero or more tests (each of which lets through values with a particular property)

  [2*x | x<-[1..3]] = [2,4,6] which is like $\{2x|x\in\{1,2,3\}\}$

  [(x,y) | x<-[1..3], y<-[1..3], x<y] = [(1,2),(1,3),(2,3)]
- Note that later generators can refer to earlier values

  [(x,y) | x<-[1..3], y<-[(x+1)..3]] = [(1,2),(1,3),(2,3)]

Prog & Alg I (COMS10002)
Week 10 – Polymorphism and Sorting

Dr. Oliver Ray
Department of Computer Science
University of Bristol

Wednesday 3rd December, 2014

(Polymorphic) Functions on Lists (p.87)

- Many functions can operate over lists of any type. For example the code for returning the first item of a list is same no matter what type of items are in the list

```
head (x:_) = x
```

- We give such functions a “polymorphic” type by including “type variables” in their declaration

```
head :: [a] -> a
```

- Similarly for computing the length of a list

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

More Functions on Lists (p.91)

- Testing membership of a list

```
elem :: a-> [a] -> Bool
elem y [] = False
elem y (x:xs)
  | x==y       = True
  | otherwise = elem y xs
```

- Many more built-in functions are similarly defined (p.91) concat, length, head, last, tail, init, replicate, take, drop, splitAt, reverse, zip, unzip, and, or, sum, product
- Many more are available as library functions using

```
import Data.List
```

Example in ghci

```
Prelude> let xs = [1..5]
Prelude> xs
[1,2,3,4,5]
Prelude> head xs
1
Prelude> tail xs
[2,3,4,5]
Prelude> length xs
5
Prelude> xs !! 2
3
Prelude> xs ++ xs
[1,2,3,4,5,1,2,3,4,5]
Prelude> splitAt 2 xs
([1,2],[3,4,5])
Prelude> reverse xs
[5,4,3,2,1]
Prelude> :q
```

Insertion Sort Concept

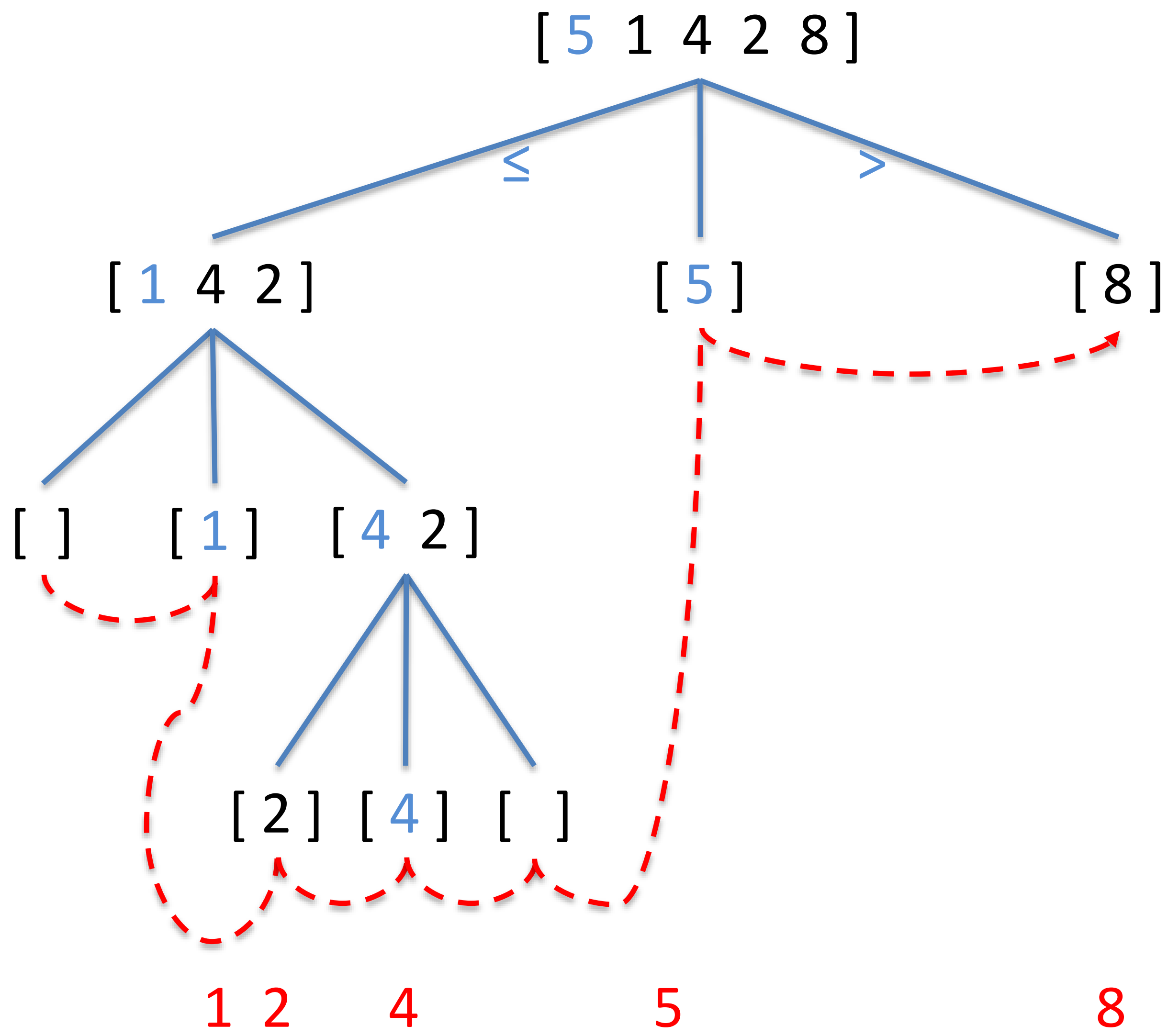
[5 1 4 2 8]	[ ]
[5 1 4 2]	[8]
[5 1 4]	[2 8]
[5 1]	[2 4 8]
[5]	[1 2 4 8]
[ ]	[1 2 4 5 8]

iSort (p.123-4)

```
iSort :: [Int] -> [Int]
iSort [] = []
iSort (x:xs) = ins x (iSort xs)

ins :: Int -> [Int] -> [Int]
ins y [] = [y]
ins y (z:zs)
      | y <= z      =   (y : z : zs)
      | otherwise   =   (z : ins y zs)
```

Quick Sort Concept



qSort (p.127)

```
qSort :: [Int] -> [Int]
qSort [] = []
qSort (x:ys) = (qSort ls) ++ [x] ++ (qSort gs)
    where
    ls = [y | y<-ys, y<=x]
    gs = [y | y<-ys, y>x]
```