Collections of things: Tuples and Lists

A first taste!

Tuples

```
examplePair :: (Double, Bool)
examplePair = (3.14 , False)

exampleTriple :: (Bool, Int, String)
exampleTriple = (False, 42, "Answer")

exampleFunction :: (Bool, Int, String) -> Bool
exampleFunction (b, i, s) = not b && length s < i
```

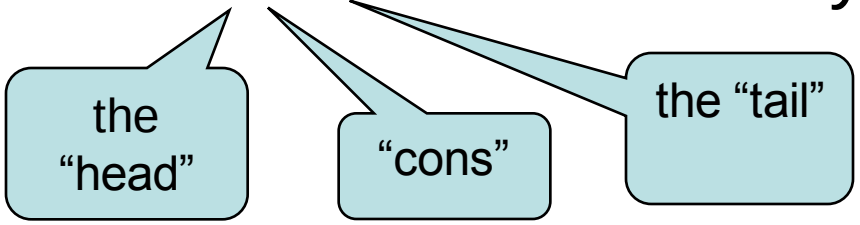
Lists

- The “duct tape” of functional programming
- Collections of things of the same type.
  - Two lists of the same type may have different number of elements

- For any type x, [x] is the type of lists of x’s
  - e.g. [Bool] is the type of lists of Bool

Lists

- The values in [A] are either of the form
  - [ ] , the empty list (also called *nil*)
  - x:xs where x has type A and xs has type [A].



- Which of these are in [Bool] ?

True : [ ] True:False False:(False:[ ])

List shorthands

- The following are all equivalent ways of writing the list 1:(2:(3:[ ]))

  1:2:3:[ ]

  [1,2,3]

  [1..3]

- The third is a bit special – it is really a shorthand for an expression which builds the list. Other examples: [‘a’..’z’] and [1..]

Functions over lists

- Functions over lists can be defined using pattern matching. E.g.,

```
summerize :: [String] -> String
summerize [ ]  = “None”
summerize [x]  = “Only ” ++ x
summerize  _   = “Several things.”
```

The “don’t care” pattern

Functions over lists

- Primitive recursion is the most common form:

```
doubles :: [Integer] -> [Integer]
  -- doubles [3,6,12] = [6,12,24]
doubles [ ]    = ...
doubles (x:xs) = ...
```

Functions over lists

- Primitive recursion is the most common form:

```
doubles :: [Integer] -> [Integer]
  -- doubles [3,6,12] = [6,12,24]
doubles []     = []
doubles (x:xs) = (2*x) : doubles xs
```

- Would not write it in this way – it is such a common pattern that we define a general function

map

```
-- map f [x1,x2,…,xn] = [f x1,f x2,…,f xn]
map f [ ]     = ...
map f (x:xs) = ...
```

map

```
-- map f [x1,x2,…,xn] = [f x1,f x2,…,f xn]
map f [ ]     = [ ]
map f (x:xs) = f x : map f xs
```

Note: map is part of the standard Prelude - does not need to be defined

filter

Produce a list by removing all elements which do not have a certain property from a given list:

e.g. filter even [1..9] gives [2,4,6,8]

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
          | p x         = x : (filter p xs)
          | otherwise = filter p xs
```

List comprehensions

- An alternative notation with the power of map and filter is list comprehensions

```
Prelude> [ 2*n | n<- [10..12] ]
[20,22,24]
Prelude>
```

the list of “2*n”,
where n is taken from
the list of integers
in the range 10 to 12

Based on set-theory notation;used in earlier functional languages (Hope, KRC). Popularised by Python.

List comprehensions

- [3*n | n<- [10..12], even n]

  *"the list of all 3*n, where n is taken from the list of integers from 10 to 12, and n is even".*

  – equivalent to:

  filter even [3*n | n <- [10..12]]

  map (3*) [n | n <- [10..12], even n]

  map (3*) (filter even [10..12])

Further example

- This example has multiple “generators”

```
pythag :: Int -> [(Int, Int, Int)]
pythag  n = [(x,y,z) |
                x <- [1..n],
                y <- [x..n],
                z <- [y..n],
                    x^2 + y^2 == z^2]
```

- Note that a generator can be any list-producing expression (of appropriate type), not just [a..b]-expressions.