Informatics 1

Functional Programming Lecture 3

Lists and Comprehensions

Don Sannella

University of Edinburgh

Part I

List Comprehensions

LISTS are the most important data structure in functional programming.
It's a COMPOUND data structure, for collecting simpler data together (integers, booleans, etc.).
All of the elements in a given list are of the same type.

Lists — Some examples

```
someNumbers :: [Int]
someNumbers = [1,2,3]
```

Lists are written with square brackets, with commas between items.
Earlier, we gave a function a name. Here we are giving a list a name.
someNumbers is a list of integers.
Lists can be as short (including empty) or as long as you want.

```
someChars :: [Char]
   -- equivalent: someChars :: String
someChars = ['I','n','f','1']
   -- equivalent: someChars = "Inf1"
```

A list of characters is called a STRING.
We have a special notion for such lists.

```
someLists :: [[Int]]
someLists = [[1],[2,4,2],[],[3,5]]
```

someLists is a list of lists of integers.
The lists in someLists have different lengths.
One of them is empty.

```
someFunctions :: [Picture -> Picture]
someFunctions = [invert,flipV]
```

someFunctions is a list of functions.

```
someStuff = [1,"Inf1",[2,3]]          -- type error!
```

```
someMoreNumbers :: [Int]
someMoreNumbers = [1..10]
```

There are various shorthands for writing lists, see the book.
someMoreNumbers = [1,2,3,4,5,6,7,8,9,10]

List comprehensions — Generators

```
Prelude> [ x*x | x <- [1,2,3] ]
[1,4,9]
```

This is read: "for each x DRAWN FROM [1,2,3], return x*x".
So the result is [1*1, 2*2, 3*3].

```
Prelude> [ toLower c | c <- "Hello, World!" ]
"hello, world!"
```

This yields a list of characters, i.e. a string.
toLower :: Char -> Char is a built-in function.

```
Prelude> [ (x, even x) | x <- [1,2,3] ]
[(1,False),(2,True),(3,False)]
```

This is a list of PAIRS.
The items in a pair can have different types.
The items in this list have type (Int,Bool)
so the list has type [(Int,Bool)].
even :: Int -> Bool is a built-in function.

`x <- [1,2,3]` is called a *generator*

`<-` is pronounced *drawn from*

List comprehensions are for doing "whoosh"-style programming - operating on all of the items in a list at once.
It's supposed to resemble set notation in mathematics, for instance { x | 2 < x < 20 }.
Notice that the expression giving the result (normally) mentions the element that is drawn from the starting list.

List comprehensions — Guards

```
Prelude> [ x | x <- [1,2,3], odd x ]
[1,3]

Prelude> [ x*x | x <- [1,2,3], odd x ]
[1,9]

Prelude> [ x | x <- [42,-5,24,0,-3], x > 0 ]
[42,24]

Prelude> [ toLower c | c <- "Hello, World!", isAlpha c ]
"helloworld"
```

isAlpha :: Char -> Bool is a built-in function.
isAlpha c is True if c is a letter, false otherwise.

`odd x` is called a *guard*

A GUARD is an expression whose value is True or False.
It mentions the element being tested and is used for filtering - if True we keep the element, if False we leave it out.

Sum, Product

```
Prelude> sum [1,2,3]
6
```

sum :: [Int] -> Int is a built-in function.
It computes the sum of the numbers in a list.

```
Prelude> sum []
0
```

sum [] = 0 because 0 is the IDENTITY for +: 0 + x = x = x + 0

```
Prelude> sum [ x*x | x <- [1,2,3], odd x ]
10
```

```
Prelude> product [1,2,3,4]
24
```

product :: [Int] -> Int computes the product of the numbers in a list.

```
Prelude> product []
1
```

product [] = 1 because 1 is the identity for *: 1 * x = x = x * 1

```
Prelude> let factorial n = product [1..n]
Prelude> factorial 4
24
```

sum and product are called ACCUMULATORS.

Example uses of comprehensions

```
squares :: [Int] -> [Int]
squares xs  =  [ x*x | x <- xs ]

odds :: [Int] -> [Int]
odds xs  =  [ x | x <- xs, odd x ]

sumSqOdd :: [Int] -> Int
sumSqOdd xs  =  sum [ x*x | x <- xs, odd x ]
```

We can define functions using comprehension notation.

QuickCheck

```
-- sumSqOdd.hs

import Test.QuickCheck

squares :: [Int] -> [Int]
squares xs  =  [ x*x | x <- xs ]

odds :: [Int] -> [Int]
odds xs  =  [ x | x <- xs, odd x ]

sumSqOdd :: [Int] -> Int
sumSqOdd xs  =  sum [ x*x | x <- xs, odd x ]

prop_sumSqOdd :: [Int] -> Bool
prop_sumSqOdd xs  =  sum (squares (odds xs)) == sumSqOdd xs
```

Here are two ways of defining the sum of squares of the odd numbers in a list.
prop_sumSqOdd tests, for a list xs, whether both ways give the same answer.

Running QuickCheck

```
[melchior]dts: ghci sumSqOdd.hs
GHCi, version 7.6.3: http://www.haskell.org/ghc/ :? for help
Loading package base ... linking ... done.
[1 of 1] Compiling Main             ( sumSqOdd.hs, interpreted )
*Main> quickCheck prop_sumSqOdd
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Loading package random-1.0.0.0 ... linking ... done.
Loading package mtl-1.1.0.1 ... linking ... done.
Loading package QuickCheck-2.1 ... linking ... done.
+++ OK, passed 100 tests.
*Main>
```

You can use quickCheck (imported with "import Test.QuickCheck" earlier) to test whether prop_sumSqOdd yields True for 100 randomly-chosen lists of integers.