Lecture 1 — Functional Programming

Neil Ghani

Dept. of Computer and Information Sciences
University of Strathclyde

November 3, 2014

Overview of Lecture 1

- **From Imperative to Functional Programming:**
  - What is imperative programming?
  - What is functional programming?

- **Key Ideas in Functional Programming:**
  - **Types:** Which model the data in our programs
  - **Functions:** Which are our programs
  - **Evaluating Expressions:** Which executes our programs

How not to Fail CS316!

- **Coursework:** An easy way to pick up marks. Therefore
  - Some coursework is assessed in the labs and hence you should prepare it before the labs on Tuesday
  - Always hand some coursework in since there will be some simple questions on every practical.
- **Plagiarism:** Evidence suggests those who plagiarise will fail
  - Departmental capping catches many who plagiarise.
  - Penalties can be stiff, eg deduction of 10% of module mark/year mark or termination of course
- **Reading:** The lecture notes!

What is Imperative Program — Adding up square numbers

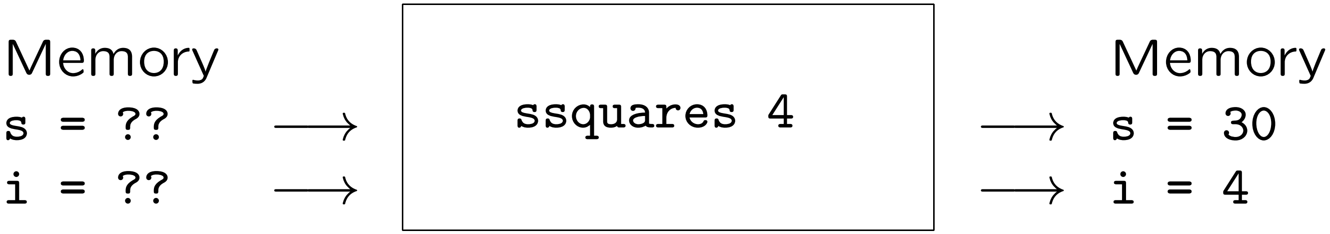
- **Problem:** Add up the first `n` square numbers

$$\texttt{ssquares n} = 0^2 + 1^2 + \ldots + n^2$$

- **Program:** We could write the following in Java

```
public int ssquares(int n){
private int s,i;
s=0; i=0;
      while (i<n) {i:=i+1;s:=s+i*i;}
}
```

- **Execution:** We may visualize running the program as follows



Memory
s = ?? ⟶
i = ?? ⟶

ssquares 4

⟶ Memory
⟶ s = 30
⟶ i = 4

- **Key Idea:** Imperative programs transform the memory

The Two Aspects of Imperative Programs

- **Functional Content:** What the program does
  - Programs take some input values and returns an output value
  - `ssquares` takes a number and returns the sum of the squares upto that number

- **Implementational Content:** How the program does it
  - Imperative programs transform the memory using assignment etc
  - `ssquares` uses variables `i` and `s` to represent locations in memory. The program transforms the memory until `s` contains the correct number.

What is Functional Programming?

- **Motivation:** Problems arise as programs contain two aspects:
  - High-level algorithms and low-level implementational features
  - Humans are good at the former but not the latter

- **Idea:** The idea of functional programming is to
  - Concentrate on the functional behaviour of programs
  - Leave memory management to the language implementation

- **Summary:** Functional languages are more abstract and avoid low level detail

A Functional Program — Summing squares in Haskell

- **Types:** First we give the type of summing-squares

```
hssquares ::  Int -> Int
```

- **Functions:** Our program is a function

```
hssquares 0 = 0
hssquares n = n*n + hssquares(n-1)
```

- **Evalutation:** Run the program by applying the function

$$
\begin{aligned}
\texttt{hssquares 2} &\Rightarrow \texttt{2*2 + hssquares 1}\\
&\Rightarrow \texttt{4 + 1*1 + hssquares 0}\\
&\Rightarrow \texttt{4 + 1 + 0}\\
&\Rightarrow \texttt{5}
\end{aligned}
$$

Key Ideas in Functional Programming I — Types

- **Motivation:** Recall that types model the data in our programs

- **Integers**: `Int` is the Haskell type $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$

- **Built in Operations:**
  - Arithmentic Operations: `+ * - div mod abs`
  - Ordering Operations: `> >= == /= <= <`

- **Expressions:** Some expressions using integers

```
5 * 4        (*) 5 4     mod 13 4       13 `mod` 4
5-(3*4)      (5-3)*4     7 >= (3*3)     5 * (-1)
```

- **Precedence:** The rules about precedence and bracketing apply

Key Ideas in Functional Programming II — Functions

- **Intuition:** Recall that a function associates to every input-value a unique output-value

$$x \in A \longrightarrow \boxed{\text{Function } f} \xrightarrow{?} y \in B$$

- **Example 1:** The *square* and *cube* functions are written

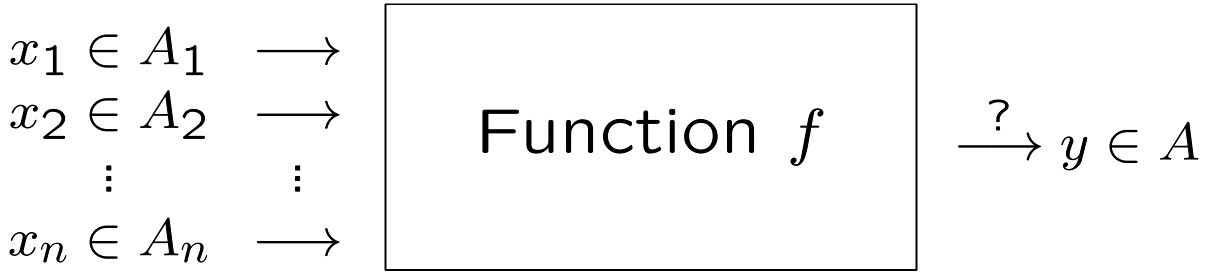
```
square ::  Int -> Int        cube ::  Int -> Int
square x = x * x             cube x = x * square x
```

- **In General:** In Haskell, functions are defined as follows

```
          ⟨function-name⟩  ::  ⟨input type⟩-> ⟨output type⟩
⟨function-name⟩ ⟨variable⟩  =  ⟨expression⟩
```

Functions with Multiple Arguments

- **Intuition:** A function f with n inputs is written `f::a1->...-> an-> a`



$$\begin{array}{ll} x_1 \in A_1 & \longrightarrow \\ x_2 \in A_2 & \longrightarrow \\ \vdots & \vdots \\ x_n \in A_n & \longrightarrow \end{array} \quad \boxed{\text{Function } f} \quad \overset{?}{\longrightarrow} y \in A$$

- **Examples:** The difference between two integers

```
diff ::  Int -> Int -> Int
diff x y = abs (x - y)
```

- **In General:**

⟨`function-name`⟩ :: ⟨`type 1`⟩`->` ...`->` ⟨`type n`⟩`->` ⟨`output-type`⟩

⟨`function-name`⟩ ⟨`variable 1`⟩ ... ⟨`variable n`⟩ = ⟨`expression`⟩

Applying Functions to Expressions

- **Motivation:** Get the result of a function by *applying* it
  - Write the function name followed by the input

- **Examples:** Here are some examples

`square` 4	`square` (3 + 1)	`square` 3 + 1
`cube (square` 2)	`difference` 6 7	`square` 2.2

- **In General:** Application is governed by the typing rule
  - If `f` is a function of type `a->b`
  - And, `u` is an expression of type `a`
  - Then `f u` is the result of applying `f` to `u` and has type `b`

Key Ideas in Functional Programming III — Evaluating Expressions

- **Procedure:**
  - Find application of a function to an expression, eg `square` 5
  - Substituted expression into function definition, eg $5 * 5$
  - Repeat as often as possible

- **Example:**

$$\begin{aligned}
\texttt{cube (square 3)} &\Rightarrow \texttt{(square 3) * square (square 3)} \\
&\Rightarrow \texttt{(3*3) * ((square 3) * (square 3))} \\
&\Rightarrow \texttt{9 * ((3*3) * (3*3))} \\
&\Rightarrow \texttt{(9 * (9*9)} \\
&\Rightarrow \texttt{729}
\end{aligned}$$

Summary — Comparing Functional and Imperative Programs

- **Difference 1:** Level of Abstraction
  - Imperative Programs include low level memory details
  - Functional Programs describe only high-level algorithms

- **Difference 2:** How exectution works
  - Imperative Programming based upon memory transformation
  - Functional Programming based upon expression evaluation

- **Difference 3:** Type systems
  - Type systems play a key role in functional programming

Advantages of Functional Programming

- **Advantage 1:** Functional Programs are easier to write
  - The algorithm we concieve of is easier to write down in a functional style. This is because functional programs are more abstract

- **Advantage 2:** Functional Programs are easier to read
  - Because they are shorter and not cluttered by implementational details, eg there is no `public static blah blah blah`!

- **Advantage 3:** Functional Programs are easier to prove correct,
  - Becuase they are based on the mathematical theory of functions, This is increasingly important in safety critical applications.

Summary — Key ideas in Haskell

- **Types:** A type is a collection of data values
  - Every expression has a type describing its nature
- **Functions:** Transform inputs to outputs
  - We build complex expressions by defining functions and applying them to other expressions
- **Evaluation:** Calculates the result of applying a function to an input
  - Expressions can be evaluated by hand or by HUGS
- **Now:** Go and look at the first practical!

Lecture 2 — More Types and Functions

Neil Ghani

Dept. of Computer and Information Sciences
University of Strathclyde

November 3, 2014

Recall ....

- **Basic Idea:** Functional Programming is about
  - Writing *expressions* — these are our programs
  - Evaluating *expressions* — this gives the result of programs
- **Building Expressions:** Expressions are built from
  - Types provide basic expressions: `0`, `True`, `‘‘hello’’`
  - Functions allow us to build new expressions: `square 3`, `4+6`
- **Haskell Types:** There are two kinds of types in Haskell
  - Basic Types: `Int`, `Float`, `Bool`, `Char`, `String`
  - Compound Types: Function types, Pair types, List types

Overview of Lecture 2.1

- **New Types:** Today we shall learn about the following types
  - The type of booleans: `Bool`
  - The type of characters: `Char`
  - The type of strings: `String`
  - The type of fractions: `Float`

- **New Functions:** And also about the following functions
  - Conditional expressions and Guarded functions
  - Error Handling and Local Declarations

Booleans and Logical Operators

- **Values of `Bool` :** Contains two values — `True, False`
- **Logical Operations:** Various built in functions

```
&&    ::  Bool -> Bool -> Bool
||    ::  Bool -> Bool -> Bool
not   ::  Bool -> Bool
```

- **Functions:** Booleans can be used in expressions and functions

```
exOr ::  Bool -> Bool -> Bool
exOr x y = (x || y) && not (x && y)
```

- **Evaluation:** As before substitute arguments for variables

```
exOr True False  ⇒  (True || False) && not (True && False)
                 ⇒  True && not False
                 ⇒  True && True  ⇒  True
```

Conditionals — If statements

- **Conditionals:** A *conditional expression* has the form

```
if b then e1 else e2
```

  where

  - `b` is an expression of type `Bool`
  - `e1` and `e2` are expressions with the <u>same</u> type

- **Example:** Maximum of two numbers

```
maxi ::  Int -> Int -> Int
maxi n m = if n>=m then n else m
```

- **Example:** Testing if an integer is `0`

```
isZero x ::  Int -> Bool
isZero x = if (x == 0) then True else False
```

Guarded functions — An alternative to `if`-statements

- **Example:** `doubleMax` returns double the maximum of its inputs

```
doubleMax ::  Int -> Int -> Int
doubleMax x y
   | x >= y   =   2*x
   | x < y    =   2*y
```

- **Definition:** A guarded function is of the form

```
⟨function-name⟩ :: ⟨type 1⟩ -> ⟨type n⟩ -> ⟨output type⟩

⟨function-name⟩ ⟨var 1⟩...⟨var n⟩
      | ⟨guard 1⟩  =   ⟨expression 1⟩
      | ...        =   ...
      | ⟨guard n⟩  =   ⟨expression n⟩
```

where `guard 1, ..., guard n ::  Bool`

The `Char` *type*

- **Elements of `Char` :** Letters, digits and special characters

- **Forming elements of `Char` :** Single quotes form characters:

```
'd' :: Char     '3' :: Char
```

- **Functions:** Characters have codes and conversion functions

```
chr :: Int -> Char    ord :: Char -> Int
```

- **Examples:** Expressions using these functions

```
offset :: Int
offset = ord 'A' - ord 'a'

capitalize :: Char -> Char
capitalize ch = chr (ord ch + offset)
```

The String type

- **Elements of `String` :** Contains lists of characters

- **Forming elements of `String` :** Double quotes form strings

  `‘‘Newcastle Utd’’`   `‘‘1a’’`

- **Special Strings:** Newline and Tab characters

  `‘‘cat\ndog’’` `‘‘1\t2\t3’’`

- **Combining Strings:** Strings can be combined by `++`

  `‘‘cat’’ ++ ‘‘n’’ ++ ‘‘fish’’ = ‘‘catnfish’’`

- **Strings and Lists:** All list operations work as `String = [Char]`

The type of Fractions `Float`

- **Elements of `Float` :** Contains decimals, eg `-21.3`, `23.1e-2`

- **Built in Functions:** Arithmetic, Ordering, Trigonometric

- **Conversions:** Functions between `Int` and `String`

```
ceiling, floor, round   ::  Float -> Int
fromInt                 ::  Int -> Float
show                    ::  Float -> String
read                    ::  String -> Float
```

- **Overloading:** Overloading is when values/functions belong to several types

```
2 ::  Int       show ::  Int -> String
2 ::  Float     show ::  Float -> String
```

Examples of some functions

- **Example 1:** `isLower` checks if a character is lower-case

```
isLower ::  Char -> Bool
isLower x = ('a' <= x) && (x <= 'z')
```

- **Example 2:** `toUpper` capitalizes only lower case letters

- **Example 3:** `threeLines` prints 3 strings on successive lines

- **Example 4:** `isDigit` checks if a character is a digit

- **Example 5:** `duplicate` gives two copies of a string

- **Example 6:** Formatting pence

Error-Handling

- **Motivation:** Informative error messages for run-time errors
- **Example:** Dividing by zero will cause a run-time error

```
myDiv ::  Float -> Float -> Float
myDiv x y = x/y
```

- **Solution:** Use an `error` message in a guarded definition

```
myDiv ::  Float -> Float -> Float
myDiv x y
    |y /= 0      = x/y
    |otherwise   = error ‘‘Attempt to divide by 0’’
```

- **Execution:** If we try to divide by `0` we get

```
Prelude> mydiv 5 0
Program error:  Attempt to divide by 0
```

Local Declarations — `where`

- **Motivation:** Functions will often depend on other functions
- **Example :** Summing the squares of two numbers

```
sq ::  Int -> Int
sq x = x * x

sumSquares ::  Int -> Int -> Int
sumSquares x y = sq x + sq y
```

- **Problem:** Such definitions clutter the top-level environment
- **Answer:** Local definitions allow auxilluary functions

```
sumSquares2 ::  Int -> Int -> Int
 sumSquares2 x y =   sq x + sq y
                     where sq z = z * z
```

Extended Example

- **Quadratic Equations:** The solutions of $ax^2 + bx + c = 0$ are

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- **Types:** Our program will have type

```
roots ::  Float -> Float -> Float -> String
```

- **Guards:** There are 3 cases to check so use a guarded definition

```
roots a b c
   | a == 0           =   ....
   | b*b-4*a*c == 0   =   ....
   | otherwise        =   ....
```

The function `roots` — *Stage II*

- **Code:** Now we can add in the answers

```
roots a b c
  | a == 0           =  error ``Not a quadratic eqn''
  | b*b-4*a*c == 0   =  ``One root:  ''  ++ show (-b/2*a)
  | otherwise        =  ``Two roots:  ''  ++
                        show ((-b + sqrt (b*b-4*a*c))/2*a) ++
                         ``and'' ++
                        show ((-b - sqrt (b*b-4*a*c))/2*a)
```

- **Problem:** This program uses several expressions repeatedly
  - Being cluttered, the program is hard to read
  - Similarly the program is hard to understand
  - Repeated evaluation of the same expression is inefficient

The final version of `roots`

- **Local decs:** Expressions used repeatedly are made local

```
roots a b c
   | a == 0                                =   error ``Not a quadratic eqn''
   | disc == 0                             =   ``One root:  ''  ++ show centre
   | otherwise                             =   ``Two roots:  ''  ++
                                               show (centre + offset) ++
                                                ``and'' ++
                                               show (centre - offset)
     where
     disc = b*b-4*a*c
     offset = (sqrt disc) / 2*a
     centre = -b/2*a
```

Summary — Basic Types in Haskell

- We have learnt about Haskell's basic types.

- For each type we learnt
  - Its basic values
  - Its built in functions

- We learnt how to write expressions involving
  - Conditional expressions and Guarded functions
  - Error Handling and Local Declarations

Lecture 4 — New Types from Old

Neil Ghani

Dept. of Computer and Information Sciences
University of Strathclyde

November 3, 2014

Recall ....

- **Basic Idea:** Functional Programming is about
  - Writing *expressions* — these are our programs
  - Evaluating *expressions* — this gives the result of programs
- **Building Expressions:** Expressions are built from
  - Types provide basic expressions, eg `0`, `True`, ``‘‘hello’’``
  - Functions allow us to build new expressions
- **Haskell Types:** There are two kinds of types in Haskell
  - Basic Types: `Int`, `Float`, `Bool`, `Char`, `String`
  - Compound Types: Types built from other types

Overview of Lecture 2.3

- **Building New Types:** Today we will learn about the following compound types
  - Pairs
  - Tuples
  - Type Synonyms

- **Describing Types:** As with basic types, for each type we want to know
  - What are the values of the type
  - What expressions can we write and how to evaluate them

From simple data values to complex data values

- **Motivation:** Data for programs modelled by values of a type
- **Problem:** Single values in basic types too simple for real data
- **Example:** A point on a plane can be specified by
  - A number for the x-coordinate and another for the y-coordinate
- **Example:** A name could be specified by
  - A string for the first name and another for the second name
- **Example:** The performance of a football team could be
  - A string for the team and a number for the points

New Types from Old I — Pair Types

- **Key Idea:** Pair types consist of two values.
- **In Pascal:** We write the following to model points

```
record
       xcoord :  integer;
       ycoord :  integer;
end
```

- **In Haskell:** We have the simpler notation
  - If `s` is a type and `t` is a type, then `(s,t)` is a type
- **Examples:** For instance
  - A point could have type `(Int, Int)`
  - A name could have type `(String, String)`
  - The performance of a team could have type `(String,Int)`

New Types from Old I — Pair Expressions

- **Question:** What are the values of a pair type?

- **Answer:** A pair type contains pairs of values, ie
  - If `e1` has type `s` and `e2` has type `t`
  - Then `(e1,e2)` has type `(s,t)`

- **Examples:** For instance
  - The point `(5,3)` has type `(Int, Int)`
  - The name `(‘‘Alan’’,‘‘Shearer’’)` has type `(String, String)`
  - The performance `(‘‘Newcastle’’, 22)` has type `(String,Int)`

New Types from Old I — Functions using Pairs

- **Types:** Pair types can be used as input and/or output types
- **Key Idea:** If input is a pair-type, use `(x,y)` in definition
- **Key Idea:** If output is a pair-type, result is often $(\langle\texttt{exp}\rangle, \langle\texttt{exp}\rangle)$
- **Examples:** The functions `fst` and `snd` are vital

```
fst ::  (a,b) -> a
fst (x,y) = x

winUpdate ::  (String,Int) -> (String,Int)
winUpdate (x,y)  = (x,y+3)

movePoint ::  Int -> Int -> (Int,Int) -> (Int,Int)
movePoint m n (x,y)  = (x+m,y+n)
```

New Types from Old II — Tuples

- **Motivation:** Some data consists of more than two parts
- **Example:** People on a mailing list
  - Specified by name, telephone number, and age
  - A person on the list can have type `(String, Int, Int)`
- **Idea:** Generalise pairs of types to collections of types
- **Type Rule:** Given types `a1,...,an` , then `(a1,...,an)` is a type
- **Expression Formation:** Given expressions `e1::a1, ..., en::an` , then `(e1,...,en)` is an expression of type `(a1,...,an)`

Functions using Tuples

- **Key Idea:** As before, if input/output is a tuple use `(...)`

```
isAdult :: (String,Int,Int) -> Bool
isAdult (x,y,z) = if z>=18 then True else False

updateMove :: (String,Int,Int) -> Int -> (String,Int,Int)
updateMove (x,y,z) w = (x,w,z)

updateAge :: (String,Int,Int) -> (String,Int,Int)
updateAge (x,y,z) = (x,y,z+1)
```

- **Calendar Dates:** Represented by a triple of integers `(Int,Int,Int)`

```
isSummer :: (Int,Int,Int) -> Bool
isSummer (x, y, z) = (6<=y) && (y<=8)
```

Pattern Matching

- **Simple Functions:** We started with functions of the form

$$\langle\texttt{function-name}\rangle\ \langle\texttt{variable}\rangle\ \texttt{=}\ \ \langle\texttt{expression}\rangle$$

- **Generalisation:** Then we allowed
  - Multiple arguments
  - Guarded definitions
  - Local declarations

- **Pattern Matching:** Now we also replace *variables* by *patterns*

General Definition of a Function

- **Definition:** Functions now have the form

```
<function-name> ::  <type 1> -> ...  -> <type n> -> <out-type>

<function-name> <pat 1> ...  <pat n> = <exp n>
```

- **Patterns:** Patterns are
  - Variables `x` : Use for any type
  - Constants `0`, `True`, `‘‘cherry’’` : Definition by cases
  - Tuples `(x,..,z)` : If the argument has a tuple-type
  - Wildcards _: If the output doesnt use the input

- **In general:** Use several lines and mix patterns.

More Examples

- **Example:** Using values and wildcards

```
isZero ::  Int -> Bool
isZero 0 = True
isZero _ = False
```

- **Example:** Using tuples and multiple arguments

```
expand ::  Int -> (Int,Int) -> (Int,Int)
expand n (x,y) = (n*x,n*y)
```

- **Example:** Days in the month

```
days ::  String -> Int -> Int
 days ``January''    x    = 31
 days ``February''   x    = if isLeap x then 29 else 28
 days ``March''      x    = 31
  .....
```

New Types from Old III — Type Synonyms

- **Motivation:** More descriptive names for particular types.

- **Definition:** Type synonyms are declared with the keyword `type` .

```
type Team = String
type Goals = Int
type Result = String
type Match = ((Team,Goals), (Team,Goals))

nusw ::  Match
nusw = ((‘‘Newcastle", 8), (‘‘Sheffield’’, 0))
```

- **Functions:** Types of functions are more descriptive, same code

```
homeTeam ::  Match -> Team
totalGoals ::  Match -> Goals
result ::  Match -> Result
```

Summary

- **Tuples:** Collections of data from other types

- **Pairs:** Pairs, triples etc are examples of tuples

- **Type synonyms:** Make programs easier to understand

- **Pattern Matching:** General description of functions covering definition by cases, tuples etc.

- **Pitfall!** What is the difference between

```
addPair ::  (Int,Int) -> Int
addPair (x,y) = x + y

addTwo ::  Int -> Int -> Int
addTwo x y = x + y
```

Lecture 4 — List Types

Neil Ghani

Dept. of Computer and Information Sciences
University of Strathclyde

November 3, 2014

Recall ....

- **Basic Idea:** Functional Programming is about
  - Writing *expressions* — these are our programs
  - Evaluating *expressions* — this gives the result of programs
- **Building Expressions:** Expressions are built from
  - Types provide basic expressions
  - Functions allow us to build new expressions
- **Haskell Types:** There are two kinds of types in Haskell
  - Basic Types: `Int`, `Float`, `Bool`, `Char`, `String`
  - Compound Types: Types built from other types

Overview of Lecture 4 — List Types

- **Lists:** What are lists?
  - Forming list types
  - Forming elements of list types
- **Functions over lists:** Some old freinds, some new friends
  - Functions: `cons`, `append`, `head`, `tail`
  - Some new functions: `map`, `filter`
- **Clarity:** Unlike Java, Haskell treatment of lists is clear
  - No list iterators!

The List Type Constructor

- **Motivation:** A key data-type in functional programming

- **Type Formation:** If `a` is <u>any</u> type, then `[a]` is a type

- **Example 1:** Lists of characters: `[Char]`

- **Example 2:** Lists of lists of integers: `[[Int]]`

- **Example 3:** Lists of functions on integers: `[Int -> Int]`

- **Example 4:** Lists of points: `[Point]`

Building Lists

- **List Expressions:** Lists are written using square brackets `[...]`
  - If `e1` ,...,`en` are expressions of type `a`
  - Then `[e1, ..., en]` is an expression of type `[a]`
- **Example 1:** `[3, 5, 14] :: [Int]`
- **Example 2:** `[3, 4+1, double 7] :: [Int]`
- **Example 3:** `[['a'], ['a','b'], ['a','b','c']] :: [[Char]]`
- **Example 4:** `[double, square, cube] :: [Int -> Int]`
- **Empty List:** The empty list is `[]` and belongs to all list types

Some built in functions - Two infix operators

- **Cons:** The cons function `:` adds an element to a list

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

 a = Int        1         :  [2,3,4]      = [1,2,3,4]
 a = Int->Int   addone    :  [square]     = [addone, square]
 a = Char       'a'       :  ['b', 'z']   = ['a', 'b', 'z']
```

- **Append:** Append joins two lists together

```
++ ::  [a] -> [a] -> [a]

 a = Bool    [True, True] ++ [False]   = [True, True, False]
 a = Int     [1,2] ++ ([3] ++ [4,5])   = [1,2,3,4,5]
 a = Int     ([1,2] ++ [3]) ++ [4,5]   = [1,2,3,4,5]
 a = Float   [] ++ [54.6, 67.5]        = [54.6, 67.5]
 a = Int     [6,5] ++ (4 :  [7,3])     = [6,5,4,7,3]
```

More Built in Functions

- **Head and Tail:** Head gives the first element of a list, tail the remainder

```
a = Int->Int    head [double, square]   = double
a = Int         head ([5,6]++[6,7])     = 5

a = Int->Int    tail [double, square]   = [square]
a = Int         tail ([5,6]++[6,7])     = [6,6,7]
```

- **Length and Sum:** The length of a list and the sum of a list of integers

```
length (tail [1,2,3]) = 2
sum [1+4,8,45] = 58
```

- **Sequences:** The list of integers from `1` to `n` is written

```
[1 ..  n]
```

Two New Functions — Map And Filter

- **Map:** Map is a function which has two inputs.
  - The first input is a function of type `Int -> Int`
  - The second is a list of integers

  The output is the list obtained by applying the function to every element of the input list

- **Filter:** Filter is a function which has two inputs.
  - The first input is a function of type `Int -> Bool`
  - The second is a list of integers

  The output is the list of those elements of the input list which the function maps to `True`

Using Map and Filter

- **Even Numbers:** The even numbers less than or equal to `n`
    - `evens::Int->[Int]`

- **Solution 1** — Using `map`.

- **Solution 2** — Using `filter`

More Examples

- **Methodology:** Develop algorithm by asking
  - Can we apply a funciton to every member of a list
  - Can we delete all members of a list not satisfying a property
- **Example 1:** `factors` calculate the factors of an integer
- **Example 2:** `isPrime` tests if an integer is prime
- **Example 3:** `primesUpto` calculates primes upto an integer

Summary

- **Types:** We have looked at list types
  - What list types and list exressions looks like
  - What built in functions are availiable
- **New Functions:** Map and filter
  - Apply a function to every member of a list
  - Delete those that dont satisfy a properties
- **Algorithms:** Develop an algorithm by asking
  - Can I solve this problem by applying a function to every kmember of a list or by deleting certain elements.

Lecture 5 — List Comprehensions

Neil Ghani

Dept. of Computer and Information Sciences
University of Strathclyde

November 3, 2014

Recall ....

- **Basic Idea:** Functional Programming is about
  - Writing *expressions* — these are our programs
  - Evaluating *expressions* — this gives the result of programs
- **Building Expressions:** Expressions are built from
  - Types provide basic expressions
  - Functions allow us to build new expressions
- **Haskell Types:** There are two kinds of types in Haskell
  - Basic Types: `Int`, `Float`, `Bool`, `Char`, `String`
  - Compound Types: We are studying lists

Overview of Lecture 5

- **Revision:** What are lists
  - A reminder about `map` and `filter`
- **List comprehension:** An alternative way of writing lists
  - Definition of list comprehension
  - Comparison with `map` and `filter`
- **Examples:** Which allow you to start practical 2

Revision

- **Type Formation:** If `a` is any type, then `[a]` is a type
- **List Expressions:** Lists are written using square brackets `[...]`
  - If `e1` ,...,`en` are expressions of type `a`
  - Then `[e1, ..., en]` is an expression of type `[a]`
- **Functions:** Some useful built in functions
  - **Cons:** Attaches an element to the front of a list `:` `:: a -> [a] -> [a]`
  - **Append:** Append joins two lists together `++ :: [a] -> [a] -> [a]`
  - **Head:** Returns the first element of a list `head :: [a] -> a`
  - **Tail:** Deletes the first element of a list `tail :: [a] -> [a]`

Map And Filter

- **Map:** Map is a function which has two inputs.
  - The first input is a function
  - The second is a list of integers

  The output is the list obtained by applying the function to every element of the input list

- **Filter:** Filter is a function which has two inputs.
  - The first input is a function returning a boolean
  - The second is a list of integers

  The output is the list of those elements of the input list which the function maps to `True`

List Comprehension — An alternative to `map` *and* `filter`

- **Example 1:** If `ex = [2,4,7]` then

```
[ 2*a | a <- ex ] = [4,8,14]
```

- **Example 2:** If `isEven :: Int->Bool` tests for even-ness

```
[ isEven a | a <- ex ] = [True,True,False]
```

- **In General:** List comprehensions are

  [⟨`exp`⟩ | ⟨`variable`⟩ `<-` ⟨`list-exp`⟩]

- **Evaluation:** The meaning of a list comprehension is
  - Take each element of `list-exp` and evaluate the expression `exp`

Using List Comprehensions Instead of `map`

- **Example 1:** A function which doubles a list's elements

```
double ::  [Int] -> [Int]
double l = [ 2*x | x <- l]
```

- **Example 2:** A function to tell if list elements are even

```
isEvenList ::  [Int] -> [(Int,Bool)]
isEvenList l = [ (a, isEven a) | a <- l]
```

- **Example 3:** A function to add pairs of numbers

```
addpairs ::  [(Int,Int)] -> [Int]
addpairs l = [ a+b | (a,b) <- l]
```

- **In general:** `map f l = [f x | x <- l]`

Using List Comprehensions Instead of Filter

- **Intuition:** List Comprehension also selects elements from a list

- **Example:** We can select the even numbers in a list

```
[ a | a <- l, isEven a]
```

- **Example:** Selecting names beginning with `A`

```
names ::  [String] -> [String]
names l ::  [ a | a <- l , head a = 'A' ]
```

- **Example:** Combining selection and applying functions

```
doubleEven ::  [Int] -> [Int]
doubleEven l ::  [ 2*a | a <- l , isEven a ]
```

General Form of List Comprehension

- **In General:** These list comprehensions are of the form

  `[ ⟨exp⟩ | ⟨variable⟩ <- ⟨list-exp⟩ , ⟨test⟩ ]`

- **Example:** We can also use several tests — if `l = [2,5,8,10]`

  ```
  [ 2*a | a <- l , isEven a , a>3 ] = [16,20]
  ```

- **Key Example:** Cartesian product is the list of pairs, the first component of which comes from the first list and the second component from the second list. Use two generators

  ```
  [ (x,y) | x<-[1,2,3], y<-['a','b','c'] ] = [(1,'a'), (1,'b') ...  ]

  league ::  [Team]
  fixtures = [ ??  | ??  ]
  toonGames = [??  | ??  ]
  ```

Erothosthenes Sieve — The algorithm

- **Motivation:** A more efficient way to calculate prime numbers

- **Algorithm:** Given a list of numbers
  - Keep the first element and delete all multiples of the first element from the tail.
  - Repeat this procedure on the tail

- **Example:** Thus,

```
seive [2,3,4,5,6,7,8,9,10,11,12]  =  2 :  seive [3,5,7,9,11]
                                  =  2 :  3 :  seive [5,7,11]
                                  =  2 :  3 :  5 :  seive [7,11]
```

Erothosthenes Sieve — The code

- **Strategy:** We implement the algorithm as follows
  - *Keep the first element* — use `head` and `:`
  - *Delete all multiples of the first element* — use list comprehension and a test
  - *Repeat this procedure* — apply the function again
- **Code:** Here is the code
- **Primes:** Can then be calculated

```
priomes n = seive [1 ..  n]
```

Removing Duplicates

- **Problem:** Given a list remove all duplicate entries

- **Algorithm:** Given a list,
  - Keep first element
  - Delete all occurrences of the first element
  - Repeat the process on the tail

- **Code:**

Summary

- We have looked at list types
  - What list types and list expressions looks like
  - What built in functions are available

- List comprehensions are like `filter` and `map`. They allow us to
  - Select elements of a list
  - Delete those that dont satisfy certain properties
  - Apply a function to each element of the remainder

3.2 — Recursion over Natural Numbers

Neil Ghani

Dept. of Computer and Information Sciences
University of Strathclyde

November 3, 2014

Recall ....

- **Basic Idea:** Functional Programming is about
  - Writing *expressions* — these are our programs
  - Evaluating *expressions* — this gives the result of programs
- **Building Expressions:** Expressions are built from
  - Types provide basic expressions: `0`, `True`, `‘‘hello’’`
  - Functions allow us to build new expressions
- **Haskell Functions:** Haskell funtions we have seen
  - Simple definitions, Multiple Arguments, Local Declarations
  - Guarded functions, Pattern matching

Overview of Lecture 3.2 — Recursion over Natural Numbers

- **Recursion:** General features of recursion
  - What is a recursive function
  - How do we write recursive functions
  - How do we evaluate recursive functions

- **Recursion over Natural Numbers:** Special features
  - How can we guarantee evaluation works
  - Recursion using patterns
  - Avoiding negative input

- **Example:** Adding up the first n sqaures

$$\texttt{hssquares n} = 0^2 + 1^2 + \ldots + \texttt{n}^2$$

- **Types:** First we give the type of summing-squares

```
hssquares ::  Int -> Int
```

- **Definitions:** Our program is a function

```
hssquares 0 = 0
hssquares n = n*n + hssquares(n-1)
```

- **Key Idea:** `hssquares` is recursive as its definition contains `hssquares` in the right-hand side

General Definitions

- **Definition:** A function is *recursive* if it occurs in its definition

- **Intuition:** You will have seen recursion in action before
  - Imperative procedures which call themselves
  - Divide-and-conquer algorithms

- **Why Recursion:** Recursive definitions tend to be
  - Shorter, more understandable and easier to prove correct
  - Compare with a non-recursive solution

```
nrssquares n = n * (n+0.5) * (n+1)/3
```

Evaluating Recursive Functions

- **Key Idea:** Two cases when applying a recursive function
  - Non-recursive call: Doesn't mention the recursive function
  - Recursive call: Does mention the recursive function
- **Procedure:** If a recursive function is applied to an argument
  - As before, substitute the input into the function's definition
  - But, recursive calls re-introduce the function name
  - Hence, carry-on until there are no more recursive calls
- **Question:** Will evaluation stop?

Examples of evaluation

- **Example 1:** Lets calculate `Hssquares 4`

```
hssquares 4  ⇒  4*4 + hssquares 3
             ⇒  16 + (3*3 + hssquares 2)
                ...
             ⇒  16 + (9 + ..  (1 + hssquares 0))
             ⇒  16 + (9 + ...  (1 + 0))        ⇒ 30
```

- **Example 2:** Here is a non-terminating function

```
mydouble n   =   n + mydouble (n/2)

mydouble 4  ⇒  4 + mydouble 2
            ⇒  4 + 2 + mydouble 1
            ⇒  4 + 2 + 1 + mydouble 0.5
            ⇒  ......
```

Problems with Recursion

- **Questions:** There are some outstanding problems
  - Is `hssquares` defined for every number
  - Does evaluation of recursive functions terminate
  - What happens if `hssquares` is applied to a negative number?
  - Are these recursive definitions sensible: `f n = f n` , `g n = g (n+1)`
- **Answers:** Here are the answers
  - Yes: The variable pattern matches every input
  - Not always: See example
  - Trouble: Evaluation doesnt terminate

Primitive Recursion over Natural Numbers

- **Motivation:** Restrict definitions to get better behaviour

- **Idea:** Many functions defined by three cases
  - A non-recursive call selected by the pattern `0`
  - A recursive call selected by `n`

- **Example** Our program now looks like

```
hssquares2 0   =   0
hssquares2 n   =   n*n + hssquares (n-1)
```

Examples of recursive functions

- **Example 1:** `star` uses recursion over `Int` to return a string

```
star     ::  Int -> String
star 0   =   []
star n   =   '*' :  star (n-1)
```

- **Example 2:** `power` is recursive in its second argument

```
power       ::    Float -> Int -> Float
power x 0   =     1
power x n   =     x * power x (n-1)
```

- **In General:** Use the following style of definition

```
⟨function-name⟩ 0   =   ⟨exp 1⟩
⟨function-name⟩ n   =   ⟨exp 2⟩
```

where

`⟨expression 1⟩`	does not contain	`⟨function-name⟩`
`⟨expression 2⟩`	may contain	`⟨function-name⟩` applied to `n-1`

- **Evaluation:** Termination guaranteed!
  - If the input evaluates to `0` , evaluate `⟨exp 1⟩`
  - If not, if the input is greater than `0` , evaluate `⟨exp 2⟩`

Larger Example

- **Problem:** Produce a table for `perf :: Int -> (String, Int)`

- **Stage 1:** We need the headings and then the actual table

```
table :: Int -> String
table n = header ++ printTable n
```

- **Stage 2:** The heading is just a string

```
header = ‘‘Team \ t Points \ n’’
```

- **Stage 3:** Printing the table is a recursive function

```
printTable     ::    Int -> String
printTable 0   =     .....
printTable n   =     .....
```

The Function `printTable`

- **Base Case:** If we want no entries, then just return `[]`

```
printTable 0 = []
```

- **Recursive Case:** Print n-entries by
  - Print the first `n-1` -entries
  - Add on the `n` -th entry

- **Code:** Code for the recursive call

```
printTable n =   printTable (n-1) ++
                 fst (perf n) ++ ‘‘\ t’’ ++
                 show (snd (perf n)) ++ ‘‘\ n’’
```

The Final Version

- **Code:** Heres the final version

```
table ::  Int -> String
table n = header ++ printTable n

header = ‘‘Team \ t Points \ n’’

 printTable    ::   Int -> String
 printTable 0  =    []
 printTable n  =     printTable (n-1) ++
                     fst (perf n) ++ ‘‘\ t’’ ++
                     show (snd (perf n)) ++ ‘‘\ n’’
```

Summary

- Recursion allows new functions to be written.
  - Advantages: Clarity, brevity, tractability
  - Disadvantages: Evaluation may not stop

- Recursive functions on natural numbers avoid this by
  - The values at `0` is non-recursive
  - Each recursive call uses a smaller input
  - An error-clause catches negative inputs

3.3 — Recursion over lists

Neil Ghani

Dept. of Computer and Information Sciences
University of Strathclyde

November 3, 2014

Recall ....

- **Basic Idea:** Functional Programming is about
  - Writing *expressions* — these are our programs
  - Evaluating *expressions* — this gives the result of programs
- **Building Expressions:** Expressions are built from
  - Types provide basic expressions: `0`, `True`, `‘‘hello’’`
  - Functions allow us to build new expressions
- **Haskell Functions:** Haskell functions we have seen
  - Simple definitions, Guarded functions, Pattern matching
  - Recursion over integers and natural numbers

Overview of Lecture 3.3

- **Lists:** Another look at lists
  - Lists are a recursive structure
  - Every list can be formed by `[]` and `:`
- **List Recursion:** Primitive recursion for Lists
  - How do we write recursive functions
  - Examples — `++`, `length`, `head`, `tail`, `take`, `drop`, `zip`
- **Avoiding Recursion?:** List comprehensions revisited

- **Question:** This lecture is about the following question
  - We know what a recursive function over `Int` is
  - What is a recursive function over lists

- **Answer:** In general, the answer is the same as before
  - A recursive function mentions itself in its definition
  - Evaluating the function may reintroduce the function
  - Hopefully this will stop at the answer

- **Question:** Is there an analogue of primitive recursion for lists

Another Look at Lists

- **Recall:** The two basic operations concerning lists
  - The empty list `[]`
  - The cons operator `(:)  ::  a -> [a] -> [a]`
- **Key Idea:** Every list is either empty, or of the form `a:xs`

  `[2,3,7] = 2:3:7:[]   [True, False] = True:False:[]`

- **Recursion:** Define recursive functions using the scheme
  - Non-recursive call: Define the function on the empty list `[]`
  - Recursive call: Define the function on `(x:xs)` using the function on `xs`

The General Pattern

- **Definition:** *Primitive Recursive List Functions* are given by

```
⟨function-name⟩ []       =   ⟨expression 1⟩
⟨function-name⟩ (x:xs)   =   ⟨expression 2⟩
```

where

⟨`expression 1`⟩	does not contain	⟨`function-name`⟩
⟨`expression 2`⟩	may contain expressions	⟨`function-name`⟩ `xs`

- **Compare:** Very similar to recursion over `Int`

```
⟨function-name⟩ 0       =   ⟨expression 1⟩
⟨function-name⟩ (n+1)   =   ⟨expression 2⟩
```

where

⟨`expression 1`⟩	does not contain	⟨`function-name`⟩
⟨`expression 2`⟩	may contain expressions	⟨`function-name`⟩ `n`

Examples of Recursive Functions

- **Example 1:** Doubling every element of an integer list

```
double ::  [Int] -> Int
double [] = []
double (x:xs) = (2*x) :  double xs
```

- **Example 2:** Selecting the even members of a list

```
onlyEvens ::  [Int] -> [Int]
 onlyEvens []        =   []
 onlyEvens (a:xs)    =   if isEven a then a:rest else rest
                         where rest = onlyEvens xs
```

- **Example 3:** Flattening some lists

```
flatten ::  [[a]] -> [a]
flatten [] = []
flatten (a:xs) = a ++ flatten xs
```

More Examples:

- **Example 4:** Reversing a list

```
reverse ::  [a] -> [a]
reverse [] = []
reverse (a:xs) = reverse xs ++ [a]
```

- **Example 5:** Append is defined recursively

```
append ::  [a] -> [a] -> [a]
append [] ys = ys
append (a:xs) ys = a :  (append xs ys)
```

- **Example 6:** Testing if an integer is an element of a list

```
member ::  Int -> [Int] -> Bool
member n [] = FALSE
member n (x:xs) = (x==n) || member n xs
```

Evaluation of Recursive Functions over Lists

- **Procedure** Same procedure as for recursive functions over `Int` .
  - Evaluate the input and check which expression to evaluate
  - Substitute input in definition. This can reintroduce function
  - Being primitive recursive, this process will eventually stop

- **Example:** To evaluate `member [4,3,6] 3`

$$
\begin{array}{lll}
\texttt{member [4,3,6] 3} & \Rightarrow & \texttt{member (4:[3,6]) 3} \\
 & \Rightarrow & \texttt{(4==3) || member [3,6] 3} \\
 & \Rightarrow & \texttt{False || member [3,6] 3} \\
 & \Rightarrow & \texttt{member [3,6] 3} \\
 & \Rightarrow & \texttt{(3==3) || member [6] 3} \\
 & \Rightarrow & \texttt{True || member [6] 3} \quad \Rightarrow \quad \texttt{True}
\end{array}
$$

What can we do with a list?

- **Folding:** Combining the elements of the list

```
flatten [[2], [3,72], []]  =  [2] ++ [3,72] ++ [] = [2,3,72]
      sumList [2,3,7,2,1]  =  2 + 3 + 7 + 2 + 1
```

- **Mapping:** Applying a function to every member of the list

```
double [2,3,72,1]  =  [2*2, 2*3, 2*72, 2*1]
isEven [2,3,72,1]  =  [True, False, True, False]
```

- **Filtering:** Selecting particular elements

```
onlyEvens [2,3,72,1] = [2,72]
```

- **Other types:** Breaking lists up, combining lists

```
head, tail, take, drop, zip
```

List Comprehension Revisited

- **Recall:** List comprehensions look like

  [⟨`exp`⟩ | ⟨`variable`⟩ `<-` ⟨`list-exp`⟩ , ⟨`test`⟩]

- **Intuition:** Roughly speaking this means
  - Take each element of the list ⟨`list-exp`⟩
  - Check they satisfy ⟨`test`⟩
  - Form a list by applying ⟨`exp`⟩ to those that do

- **Idea:** Equivalent to a bit of filtering and then mapping

Summary

- List are a recursive data-structure
- Hence, functions over lists tend to be recursive
- Primitive recursion over lists is similar to natural numbers
  - A non-recursive call using the pattern `[]`
  - A recursive call using the pattern `(a:xs)`
- List comprehension is an alternative way of doing some recursion

Lecture 8 — More Complex Recursion

Neil Ghani

Dept. of Computer and Information Sciences
University of Strathclyde

November 3, 2014

Recall ....

- **Basic Idea:** Functional Programming is about
  - Writing *expressions* — these are our programs
  - Evaluating *expressions* — this gives the result of programs
- **Building Expressions:** Expressions are built from
  - Types provide basic expressions: `0`, `True`, `‘‘hello’’`
  - Functions allow us to build new expressions
- **Haskell Functions:** Haskell functions we have seen
  - Simple definitions, Guarded functions, Pattern matching
  - Primitive recursion over natural numbers and lists

Overview

- **Problem:** Our restrictions on recursive functions are too severe
- **Solution:** New definitional formats which keep termination
  - Using new patterns
  - Generalising the recursion scheme
- **Examples:** Applications to integers and lists
- **Sorting Algorithms:** What is a sorting algorithm?
  - Insertion Sort
  - Quicksort

More general forms of primitive recursion

- **Recall:** Our primitive recursive functions follow the pattern
  - **Base Case:** Defines the function non-recursively at `0`
  - **Inductive Case:** Defines the function at `n` in terms of the function at `n-1`

$$\begin{array}{lll} \langle\texttt{function-name}\rangle\, 0 & \texttt{=} & \langle\texttt{exp 1}\rangle \\ \langle\texttt{function-name}\rangle\, \texttt{n} & \texttt{=} & \langle\texttt{exp 2}\rangle \end{array}$$

where

⟨`expression 1`⟩	does not contain	⟨`function-name`⟩
⟨`expression 2`⟩	may contain	⟨`function-name`⟩ applied to `n-1`

- **Motivation:** But some functions do not fit this shape

Fibionacci Numbers

- **Example:** The first Fibionacci numbers are `0,1` . For subsequent Fibionacci numbers, add the previous two together

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

- **Problem:** Using the following gives possible non-termination

```
fib n = fib (n-1) + fib (n-2)
```

- **Solution:** Use another base case

```
fib ::  Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

- **In General:** Use as many base cases as you need.

A Second Idea

- **Definition:** We can use the more general scheme
  - **Base Case:** Defines the function at `0` non-recursively
  - **Inductive Case:** Defines the function at `n` in terms of the function at SMALLER numbers, ie `n-1, n-2, ..., 0`

- **Example:** Calculating the highest common factor

```
hcf ::  Int -> Int -> Int
hcf n m
    |m==n        =   n
    |m>n         =   hcf m n
    |otherwise   =   hcf (n-m) m
```

- **Key Idea:** Evaluation still stops as eventually we always reach the base case which is non-recursive.

More general recursion on lists

- **Recall:** Our primitive recursive functions follow the pattern
  - **Base Case:** Defines the function at `[]` non-recursively
  - **Inductive Case:** Defines the function at `(a:xs)` in terms of the function at `xs`

```
⟨function-name⟩ []      =  ⟨exp 1⟩
⟨function-name⟩ (a:xs)  =  ⟨exp 2⟩
```

where

⟨`expression 1`⟩	does not contain	⟨`function-name`⟩
⟨`expression 2`⟩	may contain	⟨`function-name`⟩ applied to `xs`

- **Motivation:** As with integers, some functions don't fit this shape

More General Patterns for Lists

- **Recall:** With integers, we used more general patterns.
- **Idea:** Use `(a:(b:xs))` pattern to access first two elements
- **Example:** We want a function to delete every second element

```
delete [2,3,5,7,9,5,7] = [2,5,9,7]
```

- **Solution:** Here is the code

```
delete ::  [a] -> [a]
delete [] = []
delete [x] = [x]
delete (a:(b:xs)) = a :  delete xs
```

- **Example:** To delete every third element use pattern `(a:(b:(c:xs)))`

Choosing your pattern?

- **Patterns:** In a function definition, every input receives a pattern
  - If the input type is a pair, use `(x,y)` pattern
  - If the input type is a list, use `[]` and `(a:xs)` patterns
  - If different inputs have different code, use constant patterns
  - If we use the same code for every input use variable
- **Mixing patterns:** Patterns can contain patterns

```
((x,y),z)   (a:(b:xs))   ((x,y):zs)   (0:xs)
```

- **Recursion:** The non-recursive call and recursive call use different code. Hence recursive functions always use patterns

Examples of Recursion and patterns — See how the typing helps

- **Example 1:** Summing pairs

- **Example 2:** Unzipping lists

Some More Examples

- **Example 3:** Defining equality over lists

- **Example 4:** Checking if a list is a palindrome

Sorting Algorithms on Lists

- **Problem:** Elements in a list can come in any order. A sorting algorithm rearranges them in order

```
sort [2,7,13,5,0,4] = [0,2,4,5,7,13]
```

- **Recursion:** Sorting algorithms usually recursively sort a smaller list

- **Example:** To sort a list, sort the tail recursively

```
inssort ::  [Int] -> [Int]
inssort [] = []
inssort (a:xs) = insert a (inssort xs)
```

where `insert` puts the number `a` in the correct place

The function `insert`

- **Patterns:** Insert takes two arguments
  - The code for `insert` doesn't depend on the number — use a variable pattern
  - The code for `insert` depends on whether the list is empty or not — use the `[]` and `(a:xs)` patterns
- **Code:** Here is the final code

```
insert ::  Int -> [Int] -> [Int]
insert n [] = [n]
insert n (a:xs)
    |n <= a      = n:a:xs
    |otherwise   = a:(insert n xs)
```

Sorting Algorithms 2: Quicksort

- **Idea:** Given a list `l` and a number `n`

```
sort l   =   sort those elements less than n ++
             number of occurrences of n ++
             sort those elements greater than n
```

- **Stage 1:** The algorithm may be coded

```
qsort ::  [Int] -> [Int]
qsort [] = []
 qsort (a:xs)   =   qsort (less a xs) ++
                    occs a (a:xs) ++
                    qsort (more a xs)
```

where `less`, `occs`, `more` are auxilluary functions

Defining the Auxiliary Functions

- **Problem:** The auxiliary functions can be specified
  - `less` takes a number and a list and returns those elements of the list less than the number
  - `occs` takes a number and a list and returns the occurrences of the number in the list
  - `more` takes a number and a list and returns those elements of the list more than the number
- **Code:** Using list comprehensions shorten code

```
less, occs, more ::  Int -> [Int] -> [Int]
less n xs = [x | x <- xs, x < n]
occs n xs = [x | x <- xs, x == n]
more n xs = [x | x <- xs, x > n]
```

Sorting Algorithms 3: Mergesort

- **Idea:** Chop a list in half, sort each half recursively, and then merge the results together

- **Implementation:** As done in class

```
msort ::  [Int] -> [Int]
msort [] = []
msort [x] = [x]
msort xs      =   merge (msort first) (msort second)
                  where frist = take n xs
                        second = drop n xs
                        n = length xs ‘div‘ 2
merge [] ys   =  ys
merge xs []   =  xs
merge (x:xs) (y:ys) =
   if x<y then x :  merge xs (y:ys) else y :  merge (x:xs) ys
```

Summary

- **Recursion Schemes:** We've generalised the recursion schemes to allow more functions to be written
  - More general patterns
  - Recursive calls to ANY smaller value
- **Examples:** Applied to recursion over integers and lists
- **Sorting Algorithms:** We've put these ideas into practice by defining three sorting algorithms
  - Insertion Sort
  - QuickSort
  - Mergesort

Lecture 9 — Higher Order Functions

Neil Ghani

Dept. of Computer and Information Sciences
University of Strathclyde

November 3, 2014

Recall ....

- **Basic Idea:** Functional Programming is about
  - Writing *expressions* — these are our programs
  - Evaluating *expressions* — this gives the result of programs
- **Building Expressions:** Expressions are built from
  - Types provide basic expressions: `0`, `True`, `‘‘hello’’`
  - Functions allow us to build new expressions
- **Haskell Functions:** Haskell functions we have seen
  - Simple definitions, Pattern matching, Recursion
  - Today — Higher Order Functions

Overview of Lecture 9

- **Motivation:** Why do we want higher order functions
- **Definition:** What is a higher order function
- **Examples:** Three examples concerning lists
  - Mapping: Applying a function to every memebr of a list
  - Filtering: Selecting elements of a list satisfying a property
  - Folding: Combining the elements of a list

- **Example 1:** A function to double the elements of a list

```
doubleList :: [Int] -> [Int]
doubleList [] = []
doubleList (x:xs) = (2*x) : doubleList xs
```

- **Example 2:** A function to square the elements of a list

```
squareList :: [Int] -> [Int]
squareList [] = []
squareList (x:xs) = (x*x) : squareList xs
```

- **Example 3:** A function to increment the elements of a list

```
incList :: [Int] -> [Int]
incList [] = []
incList (x:xs) = (x+1) : incList xs
```

A Previous Slide — Advantages of Functional Programming

- **Advantage 1:** Functional Programs can be easier to write
  - Functional programs are more abstract
  - Functional programs reflect the algorithmic content
- **Advantage 2:** Functional Programs can be easier to read
  - Functional programs have shorter
  - Functional programs facilitate code-reuse
- **Advantage 3:** Functional programs can be easier to understand
  - Usual mathematical laws apply to functional programs

The Common Pattern

- **Problem:** Three separate definitions despite the clear pattern
- **Intuition:** Examples apply a function to each member of a list

```
function :: Int -> Int

functionList :: [Int] -> [Int]
functionList [] = []
functionList (x:xs) = (function x) : functionList xs
```

where in our previous examples `function` is

`double`   `square`   `inc`

- **Key Idea:** Make `function` an input to a higher order function

A Higher Order Function — `mapInt`

- **Idea:** Make the auxilluary function an argument

```
mapInt f [] = []
mapInt f (x:xs) = (fx) :  mapInt f xs
```

- **Advantages:** There are several advantages
  - Shortens code as previous examples are given by

```
doubleList xs   =   mapInt double xs
squareList xs   =   mapInt square xs
   incList xs   =   mapInt inc xs
```

  - Captures the algorithmic content and is easier to understand
  - Easier code-modification and code re-use

A Definition of Higher Order Functions

- **Types:** What is the type of `mapInt`
  - First argument is a function with type `Int -> Int`
  - Second argument is a list with type `[Int]`
  - Result is a list with type `[Int]`
- **Answer:** So overall type is

```
mapInt ::  (Int -> Int) -> [Int] -> [Int]
```

- **Definition:** A function is higher-order if an input is a function.
- **Imperatively:** Imperative programs cant do this

Another Example — Filtering

- **Recall:** List comprehensions or recursion can be used to select those elements of a list satisfying a certain property
- **Example:** Here are some examples

```
evens, odds, primes ::  [Int] -> [Int]

 evens l    =   [x | x <- l, isEven x]
 odds l     =   [x | x <- l, isOdd x]
 primes l   =   [x | x <- l, isPrime x]
```

- **Idea:** Each function satisfies the pattern

```
test ::  Int -> Bool
testList ::  [Int] -> [Int]
testList l = [x | x <- l, test x]
```

where `test` is `isEven, isOdd, isPrime`

Filtering Via Higher Order Functions

- **Question:** Can we make `test` into an argument of a HOF

```
filterInt test xs = [x | x <- xs, test x]
```

- **Types:** What is the type of `filterInt`
  - First argument is a function with type `Int -> Bool`
  - Second argument is a list with type `[Int]`
  - Result type is a list with type `[Int]`

- **Answer:** So overall type of `filterInt` is

```
filterInt :: (Int -> Bool) -> [Int] -> [Int]
```

Summary

- Higher Order functions are an area where functional programs are more general than their imperative counterparts

- Higher Order functions allow
  - More concise code and also code reuse
  - More abstract code, ie code closer to abstract algorithm

- Higher Order functions express algorithmic content more abstractly
  - Hence code is easier to understand

Lecture 11 — Higher Order Sorting

Neil Ghani

Dept. of Computer and Information Sciences
University of Strathclyde

November 3, 2014

Recall ....

- **Basic Idea:** Functional Programming is about
  - Writing *expressions* — these are our programs
  - Evaluating *expressions* — this gives the result of programs
- **Building Expressions:** Expressions are built from
  - Types provide basic expressions: `0`, **`True`**, ‘‘**`hello`**’’
  - Functions allow us to build new expressions
- **Haskell Functions:** Haskell functions we have seen
  - Simple definitions, Recursion, Higher Order Functions
  - Today — Higher order sorting, folding

Overview of Lecture 11

- **Folding:** What can we do with a list?
  - Mapping: Applying a function to every member of a list
  - Filtering: Selecting elements of a list satisfying a property
  - Folding: Combining the elements of a list

- **HO Sorting:** A more powerful form of sorting
  - What are the limitations of current sorting algorithms
  - How can these limitations be overcome
  - Examples from football

Three Things to do with a List

- **Mapping:** Applying a function to every member of the list

```
map double [2,3,72,1]   =   [2*2, 2*3, 2*72, 2*1]
map isEven [2,3,72,1]   =   [True, False, True, False]
```

- **Filtering:** Selecting particular elements

```
filter isEven [2,3,72,1] = [2,72]
filter isOdd [2,3,72,1] = [3,1]
```

- **Folding:** Combining the elements of the list

```
sumList [2,3,7,2,1]            =   2 + 3 + 7 + 2 + 1
allTrue [True, False, True]    =   True && False && True
flatten [[2], [3,72], []]      =   [2] ++ [3,72] ++ [] = [2,3,72]
```

- **Question:** Is folding a higher order function?

Folding as a Higher Order Function - A First Stab

- **Types:** Lets restrict ourselves to lists of integers
  - First argument takes two integers and returns an integer
  - Second argument gives a value if the list is empty
  - Third argument takes a list of integers
  - Result type is an integers

- **Answer:** `foldl` is defined as follows

```
foldl ::  (Int -> Int -> Int) -> Int -> [Int] -> Int
foldl f n [] = n
foldl f n (a:xs) = f a (foldl f xs n)
```

Some Examples

- **Usage:** To use `foldl`, ask yourself
  - What is the result of the function if the list is empty
  - What is the function which is placed in between elements

- **Examples:** Here are some examples

```
length xs     =

sumList xs    =

prodList xs   =
```

- **Warning:** There are two folds - see the book

Quicksort Revisited

- **Idea:** Recall our implementation of *quicksort*

```
qsort ::  Ord a => [a] -> [a]
qsort [] = []
 qsort (a:xs)  =  qsort less ++ occs ++ qsort more
                  where
                  less = [x | x<-xs, x<a]
                  occs = a :  [x | x<-xs, x==a]
                  more = [x | x<-xs, x>a]
```

- **Polymorphism:** Quicksort requires an order on the elements
  - So the resulting list depends upon the order on the elements
  - This requirement is reflected in type class information `Ord a`
  - Don't worry about type classes as they are beyond this course

Limitations of Quicksort

- **Example:** Football tables have type `[(Team,Points,Goals,Played)]`
- **Problem:** We might get something like

```
Arsenal    16   15   8
AVilla      8   10   8
Bradford    4    1   9
...
```

because order on `(Team,Points,Goals,Played)` is *lexicographic*

```
(x1,x2) < (y1,y2) iff x1<y1 or x1=y1 and x2<y2
```

- **Solution:** Write a new function for this problem

```
tSort [] = []
tSort (a:xs) = tSort less ++ [a] ++ tSort more
               where more = [x| x<-xs, sec x =< sec a]
                     less = [x| x<-xs, sec x > sec a]
                     sec (t,p,g,pl) = p
```

Higher Order Sorting

- **Motivation:** But what if we want different orders, eg
  - If two teams have the same points, compare goals
  - If two teams have the same points, compare goals per game
  - Sort teams in order of goals scored, not points

- **Key Idea:** Make the comparison a parameter of quicksort

```
qsortBy ::  Ord b => (a -> b) -> [a] -> [a]
qsortBy f [] = []
qsortBy f (x:xs) = qsortBy f less ++ occs ++ qsortBy f more
                 where less = [ y | y <- xs, f y < f x]
                       occs = x :  [ y | y <- xs, f y == f x]
                       more = [ y | y <- xs, f x < f y]
```

Higher Order Sorting Insertion Sort

- **Key Idea:** Only thing to remember: recursive calls and comparisons use the comparison function!

- **Implementation:** As done in class

```
msortBy ::  Ord b => (a -> b) -> [a] -> [a]
msortBy f [] = []
msortBy f [x] = [x]
msortBy f xs      =   mergeBy f (msortBy f first) (msortBy f second)
                      where first = take n xs
                            second = drop n xs
                            n = length xs ‘div‘ 2
mergeBy f [] ys   =   ys
mergeBy f xs []   =   xs
mergeBy f (x:xs) (y:ys) =
   | f x < f y = x :  mergeBy f xs (y:ys)
   | otherwise = y :  mergeBy f (x:xs) ys
```

Higher Order Sorting - Insertion Sort

- **Key Idea:** Only thing to remember: recursive calls and comparisons use the comparison function!

```
inssortBy ::  Ord b => (a -> b) -> [a] -> [b]
inssortBy f [] = []
inssortBy f (a:xs) = insertBy f a (inssortBy f xs)

insertBy ::  Ord b => (a -> b) -> a -> [a] -> [a]
insertBy f n [] = [n]
insertBy f n (a:xs)
      | f n <= f a   = n:a:xs
      | otherwise    = a:(insertBy f n xs)
```

Examples

- **Key Idea:** To use a higher order sorting algorithm, use the required order to define the function to *sort by*

- **Example 1:** To sort by points and then goals scored

```
sort1 league =
```

- **Example 2:** To sort by points and then goals per game

```
sort2 league =
```

- **Example 1:** To sort by goals scored

```
sort3 league =
```

Summary

- **Folding:** A new higher order function
  - Use to combine elements of a list
  - Many algorithms are either `map`,`filter` or `fold`
- **HO Sorting:** An application of higher order functions to sorting
  - Produces more powerful sorting
  - Order of resulting list determined by a function
  - Lexicographic order allows us to try one order and then another

5.1 — Finishing off Haskell (... Almost)

Neil Ghani

Dept. of Computer and Information Sciences
University of Strathclyde

November 3, 2014

Recall ....

- **Basic Idea:** Functional Programming is about
  - Writing *expressions* — these are our programs
  - Evaluating *expressions* — this gives the result of programs

- **Building Expressions:** Expressions are built from
  - Types provide basic expressions: `0`, **`True`**, ‘‘**`hello`**’’
  - Functions allow us to build new expressions

- **Haskell Functions:** Haskell functions we have seen
  - Simple definitions, Pattern Matching, Recursion
  - Higher Order Functions and Polymorphism

Overview of Lecture 5.1

- **Topics Covered:** Today we (almost) finish our survey of Haskell
  - Partial Application: Not giving all the inputs required
  - Lambda Notation: Expressions of function type
  - Composing Funtions: Sequential composition (functionally)
  - Auxilluary Functions: Adding a bit of memory

- **Reference:** You can find out more on the net

- **Recall 1:** In Lecture 1, we defined functions with one input

```
           ⟨function⟩ ::   ⟨input type⟩ -> ⟨output type⟩
⟨function⟩ ⟨variable⟩  =   ⟨expression⟩
```

- **Application:** (Monomorphic) Functions applied using rule

```
If     ⟨function⟩          ::   a -> b
And    ⟨expr⟩              ::   a
Then   ⟨function⟩ ⟨expr⟩   ::   b
```

- **Recall 2:** Functions with several inputs are given by

```
⟨function⟩ ::   ⟨type 1⟩  -> ...->  ⟨type n⟩  ->  ⟨out-type⟩
⟨function⟩      ⟨var 1⟩      ...    ⟨var n⟩   =   ⟨expr⟩
```

- **Confession:** There are no functions with more than one input!

But What About `times` *?*

- **Key Idea:** Functions with many inputs are actually functions with one input and whose output is itself a function.

- **Example:** The `times` function has type

```
times ::  Int -> (Int -> Int)
times x y = x * y
```

- **Application:** To multiply numbers, use application repeatedly
  - Since `5 ::  Int` , `times 5 ::  Int -> Int`
  - Next, `7 ::  Int` , and so `times 5 7 ::  Int`

- **Summary:** We have all the expressions we used to have. But we also have some new ones.

Partial Application of Functions

- **Code Re-use:** As always we want to reduce effort

- **Before:** Defining the following functions is repetative

```
times2 :: Int -> Int        times3 :: Int -> Int
times2 x = x + 2            times3 x = x + 3
```

- **Now:** Define `times2` and `times3` using code for `times`

```
times :: Int -> Int -> Int
times x y = x + y

times2 = times 2
times3 = times 3
```

- **Key Idea:** Partial application supports code-reuse

Part II — Composing Functions

- **Motivation:** Some algorithms say “Do this, then do that.”

- **Key Idea:** *Function composition* implements such algorithms

- **Intuition:** The function `g.f` does the following
  - Takes an input and applies `f` to it.
  - Then applies `g` to the result

- **Typing Rule:** If `f::a->b` and `g::b->c` are functions:

  `(g.f) :: a->c`

- **Condition:** The output of `f` and input of `g` have same type.

Examples of Composing Functions

- **Example 1:**

```
length                         ::    [a] -> Int
mysucc                         ::    Int -> Int
mysucc x                       =     x + 1

(mysucc .  length)             ::    [a] -> Int
(mysucc .  length) [2,3,4]     ⇒     mysucc (length [2,3,4])
                               ⇒     mysucc 3
                               ⇒     3+1  ⇒  4
```

- **Recall:** Expressions of list or pair type are written

  $$(\langle \texttt{expr1} \rangle , \langle \texttt{expr2} \rangle) \quad [\langle \texttt{expr1} \rangle , \ldots , \langle \texttt{exprn} \rangle]$$

- **Motivation:** How do we write expressions with function type

- **Answer 1:** Use local declarations to define the function

```
timesnum ::  Int -> (Int -> Int)
 timesnum n =  timesn where timesn m = n*m
```

- **Problem:** This expression says `timesnum n` is the function `timesn` and then `timesn` is defined. Too verbose!

- **Solution:** We want code for

  "The function which takes a number `m` and multiplies it by `n` "

Writing down functions without names

- **Answer 2:** Use lambda notation

```
timesnum n = \ m -> n * m
```

- **Definition:** The expression

$$\backslash\ \langle\texttt{variable-name}\rangle\ \texttt{->}\ \langle\texttt{expression}\rangle$$

  is shorthand for the expression

```
<function-name>
where <function-name> <variable-name> = <expression>
```

- **Defining Functions:** This gives a new way to define functions

```
double = \ x -> 2*x
atZero = \ f -> f 0
```

Evaluating Lambda-Expressions

- **Evaluation:** How do we calculate with lambda-expressions?
  - Again, substitute argument for variable after the \

- **Examples:** Here are some examples

```
          timesnum 3  ⇒   \ m->3 * m
        timesnum 3 5  ⇒  (\ m->3 * m) 5
                      ⇒  3 * 5   ⇒ 15

      atZero square   ⇒  (\ f->f 0) square
                      ⇒  square 0  ⇒ 0*0  ⇒ 0

map (\ x->2*x) [4,5]  ⇒  (\ x->2*x) 4 :  map (\ x->2*x) [5]
                      ⇒  8:  (\ x->2*x) 5 :  map (\ x->2*x) []
                      ⇒  [8,10]
```

- **For Loops:** The sum of the first `n` numbers

```
total := 0; count:= 0;
while count <= n do
          total := total + count; count := count + 1
```

- **Functonally:** Functionally we write a recursive program

```
sum 0 = 0
sum (n+1) = (n+1) + sum n
```

- **Differences:** The algorithms are different
  - The imperative program uses the memory to store the result
  - Functionally, we calculate the result directly

Adding State/Memory to Functional Programs

- **State Model:** Imperative programs transform the memory
- **Key Idea:** Memory is mimicked functionally as extra arguments

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

newsum n = sumaux n 0
```

- **Example:** Length of a list

```
lengthaux ::  [a] -> Int -> Int
lengthaux [] n = n
lengthaux (a:xs) n = length xs (n+1)

newlength xs = lengthaux xs 0
```

Summary of Todays Lecture

- **Partial Application:** Functions with many arguments are a convenient explanation. Actually:
  - They are really functions whose output is another function.
  - Such functions can be applied to some of their arguments
- **Lambda Expressions:** Used when we want
  - Expressions of function type
  - An alternate way to define functions
- **State:** Memory is mimicked functionally by extra arguments
- **Composition:** Functional composition corresponds to ;