## Outline of Lecture 8

- Higher order functions (folding, continued)
- Function composition and application
- Lambda expressions in Haskell revisited
- Partial function application
- Curried functions

## Higher-order functions: Folding (reminder)

- (Slightly) generalised folding to the right:

```
foldr ::  (a->a->a) -> a -> [a] -> a
```

- The first argument: a binary function over an arbitrary type a
- The second argument: the starting (default) value of the type a
- The third argument: the list to be folded
- The definition of `foldr`:

```
foldr f s [] = s
foldr f s (x:xs) = f x (foldr f xs)
```

## Higher-order functions: Folding (reminder)

Using `foldr`, we can define now some standard functions of Haskell, like concatenating a list of lists or calculating conjunction over a list of boolean values:

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

and ::  [Bool] -> Bool
and bs = foldr (&&) True bs
```

`foldr1` can be then defined in terms of `foldr`:

```
foldr1 f xs = foldr f (last xs) (init xs)
```

where `last` gives the last element of a list, and `init` removes that element

## Primitive recursion and folding

Often, we can replace primitive recursion over a list with folding. For example, sorting by insertion is defined (see the Lecture 5 slides)

```
iSort [] = []
iSort (x:xs) = ins x (iSort xs)
```

We could rewrite the second equation as

```
iSort (x:xs) = x `ins` (iSort xs)
```

and the whole definition as

```
iSort xs = foldr ins [] xs
```

**The problem**: the type of `ins` is `::a -> [a] -> [a]`, not `::a -> a -> a`, as required by `foldr`

## Higher-order functions: Folding (cont.)

- No problem, since the actual type of `foldr` is even more general
- Most general folding to the right:

```
foldr ::  (a->b->b) -> b -> [a] -> b
```

- The resulting folding type can be different! That means that the above definition of `iSort` via `foldr` is correct (the type `a` stays `a`, while the type `b` becomes `[a]`)

## Higher-order functions: Folding (cont.)

Another example: redefining list reversion via folding

```
rev ::  [a] -> [a]
rev xs = foldr snoc [] xs

snoc ::  a -> [a] -> [a]
snoc x xs = xs ++ [x]
```

The function `snoc` creates a list by always adding an element to the list end, i.e., does the opposite of `':'` operator

The functions `foldr1` and `foldr` have their dual versions `foldl1` and `foldl` that define folding from the left, i.e., from the list beginning towards its end

## Functional composition

- Functional composition is one of the simplest ways of structuring a functional program by composing together separate functions
- Functional composition has an effect of wiring two functions together by passing the output of one to the input of the other. As a result, a new composite function is automatically created
- It is defined as the operator `(.)`

```
(.)  ::  (b->c) -> (a->b) -> (a->c)
(f . g) x = f (g x)
```

- First execute `g::a->b`, then pass its output to `f::b->c`, and execute `f`. The resulting function `(f . g)` is of the type `::a->c`

## Forward function composition

- The execution order of `(f . g)` might be confusing: the execution order is opposite to the order of function appearance
- Another version of functional composition – forward functional composition. It can be defined as follows:

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

- Note how the definition directly composes two functions into the resulting function, no extra arguments are needed. The execution effect is as follows:

```
(f >.> g) x = (g . f) x = f (g x)
```

## Function composition: examples

Applying to arguments:

```
> (negate . abs) 5
-5
> (negate . abs) (-7)
-7
> map (negate . sum . tail) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

Defining new functions:

```
makeNegatives ::  [Integer] -> [Integer]
makeNegatives = map (negate . abs)

fun ::  Integer -> Integer
fun = sum . (replicate 8) . negate
```

## Function application

- The typical use of function application: `f x`
- In Haskell, there is also the explicit operator of function application:

```
($) ::  (a->b) -> a -> b
f $ e = f e
```

- Why is it needed?
  - Many Haskell programmers use $ as alternative to parentheses, e.g., instead of writing

    ```
    flipV (flipH (rotate horseFig))
    ```

    it is possible to write

    ```
    flipV $ flipH $ rotate horseFig
    ```

  - Sometimes we need the application operator as a function that, e.g. we pass as an argument

    ```
    zipWith ($) [sum, product] [[1,2],[3,4]]
    ```

## Associativities of function application and definition

Different associativities:

- Function application is **left associative** so that `f x y` means `(f x) y`, not `f (x y)`
- Function type symbol `->` is **right associative** so that `a -> b -> c` means `a -> (b -> c)`, not `(a -> b) -> c`

## Expressions for functions: lambda abstractions (reminder)

- Haskell definitions give us a way of defining and then naming our own functions:

```
addOne x = x + 1
```

- We can also write down a Haskell expression that directly results in a function that takes `x` to `x+1`, without giving it a name:

```
\x -> x+1
```

- Such expressions (called lambda abstractions) can be used in any places where Haskell expressions are used and evaluated, e.g, as parameters

```
map (\x -> x+1) [2,3,4]
```

  or function results

```
addNum n = (\m -> n+m)
```

## Lambda abstractions (cont.)

Equivalent ways to define a function:

```haskell
mThree x y z = x * y * z

mThree' x y = \z -> x * y * z

mThree'' x = \y z -> x * y * z

mThree''' = \x y z -> x * y * z
```

where

```haskell
\x y -> z
```

is just a shorthand for

```haskell
\x -> (\y -> z)
```

## Lambda abstractions (example)

- Suppose we want to take a list of functions and apply them all to a particular argument. One possible solution:

```
mapFuns ::  [a->b] -> a -> [b]
mapFuns [] x = []
mapFuns (f:fs) x = f x :  mapFuns fs x
```

- We can also to directly use `map` in the definition, by applying each list element to `x`:

```
mapFuns' ::  [a->b] -> a -> [b]
mapFuns' fs x = map (\f -> f x) fs
```

The function `(\f -> f x) fs` depends on the value of `x`, so it cannot as such be defined at the top level.

## Complex function applications with lambda abstractions

- Lambda abstractions are also good for composing functions together (when the standard function composition and/or application mechanisms are not so easily applicable)
- For instance, we need to "plumb" together the functions `f` and `g` into a function, which (1) takes two arguments `x` and `y`, (2) applies `f` separately to `x` and `y`, and then (3) forwards the results as arguments to the function `g`
- Using lambda abstraction, such a function is easily defined as, e.g.

```
comp2 f g = (\x y -> g (f x) (f y))
```

## Partial function application

- Let us consider a binary function, e.g.

```
multiply x y = x*y
```

- As we know, it actually stands for

```
multiply = (\x y -> x*y)
```

  which is a shorthand for

```
multiply = (\x -> (\y -> x*y))
```

- Thus, each Haskell function takes **one parameter value at a time**. If a function has multiple parameters, they are applied gradually (partially) one by one

## Partial function application (cont.)

- What happens if we apply `multiple` to single value 3?
- According to the lambda calculus rules (beta conversion), it would simplify the right hand side expression and substitute `x` with 3:

```
multiply 3 = (\y -> 3*y)
```

  The result of such partial application is a function, which multiplies any given number by 3
- Very convenient technique to adapt a function to our needs

## Partial function application (cont.)

- We can now define a function for doubling all list elements simply as:

```
doubleAll ::  [Integer] -> [Integer]
doubleAll = map (multiply 2)
```

- This solves the problem when `map` requires a single argument transformation function (of general type `a->a`) as its first parameter, while we have a binary function (of general type `b->a->a`)
- Partial application adjusts the function and makes it applicable

## Partial applied operators

Haskell operators can be also partially applied. Moreover, depending on the side we put an argument, the operator is automatically partially applied to either the first or second argument.

Examples:

- `(+2)` – a function which adds to its argument;
- `(2+)` – a function which adds to its argument;
- `(>2)` – a function which returns whether a number is greater than 2;
- `(2>)` – a function which returns whether a number is smaller than 2;
- `(3:)` – a function which adds 3 to the list beginning;
- `(++"!!!")` – a function which adds exclamations to the string end;
- `($ 3)` – a function which applies a given function to integer 3

## Partial applied operators (cont.)

- When combined with with higher-order functions like `map`, `filter` and composition, the notation becomes both powerful and elegant
- For instance,

```
ff = filter (>0) . map (+100)

doubleAll = map (*2)

(%%) ::  Eq a => a -> [a] -> Bool
(%%) = elem

is_whitespace = (%% whitespace)
```

## Partial function application (summary)

- Partial function application allows us to create a new function by applying an existing function to a part of its arguments
- It relies of the internal multi-parameter function representation as an embedded lambda expression, taking one parameter at a time
- Very convenient technique to adapt a function to our needs, especially for consequent composition with other functions

## Partial function application (cont.)

What if we want to partially apply a function, but not to its first argument?

For instance, we want to specialise

```
elem ::  Char -> [Char] -> Bool
```

to be used to check whether a character belongs to `whitespaces`

```
whitespaces = " \n\t"
```

How to partially apply `elem` for its second argument (supplying the value `whitespaces`)?

## Partial function application (cont.)

Several possible solutions:

- Redefine `elem` by switching its arguments:

```
member xs x = elem x xs


is_whitespace = member whitespace
```

- Apply some argument switching higher-order function (like `flip` from Prelude):

```
is_whitespace = (flip elem) whitespace
```

- Use lambda abstraction:

```
is_whitespace = (\ch -> elem ch whitespace)
```

## Curried functions

- As mentioned before, it is preferable to define Haskell functions in the curried form, i.e, of general type `:: T1 -> T2 -> ... Tn -> T` rather than `:: (T1,T2,...,Tn) -> T`
- Functions defined in such a way are called *curried*
- The reason: such a form makes it easy to support lambda abstraction and partial application:

```
multiply :: Int -> Int -> Int

multiply 2 :: Int -> Int

(multiply 2) 5 :: Int
```

## Currying and uncurrying

Special functions for moving between two forms of functions:

```
curry ::  ((a,b) -> c) -> (a -> b -> c)
curry g x y = g (x,y)

uncurry ::  (a -> b -> c) -> ((a,b) -> c)
uncurry f (x,y) = f x y
```

Example with `zip` and `unzip`:

```
> zip (unzip [(1,True),(99,False)])
error
> (uncurry zip) (unzip [(1,True),(99,False)])
[(1,True),(99,False)]
```

## Summary: various ways to define a function

Suppose we want to write a function that negates all the elements in the integer list and then multiplies them together. How many different ways we can come up with to define such a function? Several of them are below.

```
prodNegated ::  [Integer] -> Integer
prodNegated xs = foldr (*) 1 (map negate xs)

prodNegated_2 xs = (foldr (*) 1 . map negate) xs

prodNegated_3 xs = foldr (*) 1 $ map negate $ xs

prodNegated_4 = \xs -> (foldr (*) 1 . map negate) xs

prodNegated_5 = foldr (*) 1 . map negate

prodNegated_6 = foldr (\x y -> negate x * y) 1
```

## Datatype constructors

Datatype constructors are functions too ⇒ they can be partially applied, passed as arguments or returned as results

Example:

```
data People = Person String Int deriving (Show)

somePeople = zipWith Person ["Bernie Stauskas",
"Bob Dyllan"] [25,71]

> print somePeople
[Person "Bernie Stauskas" 25,
Person "Bob Dyllan" 71]
```

## Some properties of higher-order functions

- `f . (g . h) = (f . g) . h`
- `map (f . g) = map f . map g`
- `map f (xs ++ ys) = map f xs ++ map f ys`
- `filter p (xs ++ ys) = filter p xs ++ filter p ys`
- `filter p . map f = map f . filter (p . f)`
- `foldr f st (xs ++ ys) =`
  `f (foldr f st xs) (foldr f st ys)`
- `foldr f st . map g = foldr (\x y -> f (g x) y) st`