Outline of Lecture 8

- Lambda expressions (lambda abstractions) in Haskell
- Partial application
- Currying and uncurrying
- Some properties of higher-order functions

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 (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.

Lambda abstractions (summary)

- Lambda abstractions allow us to create anonymous functions on-the-fly, e.g.,

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

- They can be used to define function arguments for higher order functions, e.g.

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

  or function results

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

- Essentially, lambda abstractions are basic constructors for functions

Lambda abstractions (cont.)

- Lambda abstractions can have multiple parameters, e.g.,

```
\x y -> x^y + 10*x
```

- Such a notation for multiple parameters is just a shorthand for

```
\x -> (\y -> x^y + 10*x)
```

- Any function definition can be rewritten as giving a name to the corresponding lambda expression, e.g.

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

  Thus, lambda abstraction is basic (primary) function constructor

Lambda abstractions (cont.)

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

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
```

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 `a->a->a`)
- Partial application adjusts the function and makes it applicable

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 the Homework 4!):

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

- Use lambda abstraction:

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

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 add to its argument;
- (2+) – a function which add 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)

(%%) = elem
is_whitespace = (%% 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`
- 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)]
```

Associativity

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`

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`