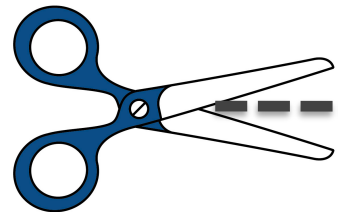


Higher-Order Functions, Declaring Types, Declaring Classes

LECTURE 3

A Thing About My Slides

Things everybody can see



Things people in the back can't see

I feared having ≤ 3 students 🥲

**10 students would have made
me happy 😊**

I dreamed of having 20 students_{zzZ}

**At the first lecture I counted
37 students 😊**

**51 students registered in CIS
and handed in exercises 😊**

That is AWESOME!
(and I am happy 😊)

But ...

25 student presentations is a lot 😵

No presentations!

Instead:

- Write a paper/tutorial (> 2 pages)
- Review another paper/tutorial
- Do this in pairs (and in English)

Grading (1/2)

- Weekly exercises
 - Small set of exercises (nearly) every week
 - (individually, for now)
- Programming project
 - During second half of the course
 - Create a real application
 - In teams (4/5 persons)
- Research/Write
 - Research an FP related topic
 - Write a tutorial/paper
 - Review a paper of fellow students
 - In pairs

Grading 2/2

- Final grade:
 - 40% exercises + 40% project + 20% writing
- For a passing grade:
 - 50% of the points of the weekly exercises
 - ≥ 4 for project
 - ≥ 4 for presentation
- Master/Diploma students
 - One extra question in weekly exercise
 - Is a bonus exercise for bachelor students

Based on slides of Graham Hutton

HIGHER-ORDER FUNCTIONS

In Haskell ...

... functions are values!

Functions are values

So, we can pass functions around as any other value!

```
twice    :: (a -> a) -> a -> a  
twice f x = f (f x)
```

```
>twice (+1) 40  
42
```

Higher-Order Functions

A function is called higher-order if it takes a function as an argument or returns a function as a result.

```
twice    :: (a → a) → a → a  
twice f x = f (f x)
```

twice is higher-order because it takes a function as its first argument.

Why Are They Useful?

- Common programming idioms can be encoded as functions within the language itself.
- Domain specific languages can be defined as collections of higher-order functions.
- Algebraic properties of higher-order functions can be used to reason about programs.

The Map Function

The higher-order library function called map applies a function to every element of a list.

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

For example:

```
> map (+1) [1, 3, 5, 7]  
[2, 4, 6, 8]
```

The Map Function

The map function can be defined in a particularly simple manner using a list comprehension:

```
map f xs = [f x | x ← xs]
```

The Map Function

Alternatively, for the purposes of proofs, the map function can also be defined using recursion:

```
map f [] = []  
map f (x:xs) = f x : map f xs
```

The Filter Function

The higher-order library function `filter` selects every element from a list that satisfies a predicate.

```
filter :: (a -> Bool) -> [a] -> [a]
```

For example:

```
> filter even [1..10]
```

```
[2,4,6,8,10]
```

The Filter Function

Filter can be defined using a list comprehension:

```
filter p xs = [x | x ← xs, p x]
```

The Filter Function

Alternatively, it can be defined using recursion:

```
filter p []           = []  
filter p (x:xs)      =  
  | p x               = x : filter p xs  
  | otherwise         = filter p xs
```

The Foldr Function

A number of functions on lists can be defined using the following simple pattern of recursion:

```
f [] = v
f (x:xs) = x ⊕ f xs
```

f maps the empty list to some value v

f maps any non-empty list to some function \oplus applied to its head and f of its tail

The Foldr Function

```
sum [] = 0  
sum (x:xs) = x + sum xs
```

$v = 0$
 $\oplus = +$

The Foldr Function

```
product [] = 1  
product (x:xs) = x * product xs
```

$v = 1$
 $\oplus = *$

The Foldr Function

The higher-order library function foldr (fold right) encapsulates this simple pattern of recursion, with the function \oplus and the value v as arguments.

For example:

```
sum      = foldr (+) 0
product = foldr (*) 1
```

The Foldr Function

Foldr itself can be defined using recursion:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v []      = v
foldr f v (x:xs) = f x (foldr f v xs)
```

However, it is best to think of foldr non-recursively, as simultaneously replacing each (:) in a list by a given function, and [] by a given value.

The Foldr Function

```
sum [1,2,3]
```

=

```
foldr (+) 0 [1,2,3]
```

=

```
foldr (+) 0 (1:(2:(3:[])))
```

=

```
1+(2+(3+0))
```

Replace each (:) by (+) and [] by 0

=

```
6
```

Why Is Foldr Useful?

- Some recursive functions on lists, such as sum, are simpler to define using foldr.
- Properties of functions defined using foldr can be proved using algebraic properties of foldr, such as fusion and the banana split rule.
- Advanced program optimisations can be simpler if foldr is used in place of explicit recursion.

Quiz

Define `and` in terms of `foldr`

```
and :: [Bool] -> Bool
and []      = True
and (x:xs) = x && and xs
```

Quiz

Define the function `length` in terms of `foldr`

```
length      :: [a] → Int
length []   = 0
length (_:xs) = 1 + length xs
```

Quiz

Define reverse in terms of foldr

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

Function Composition

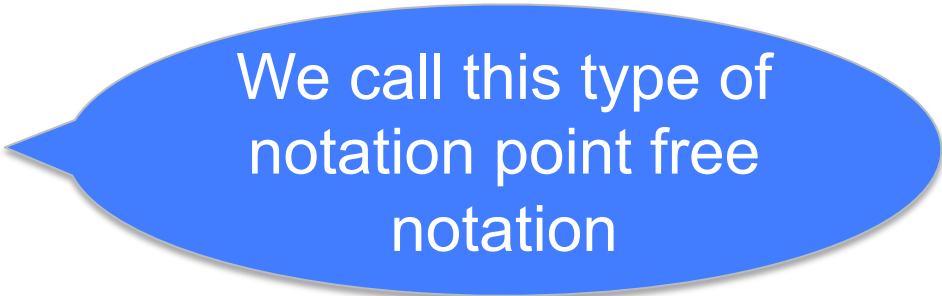
The library function `(.)` returns the composition of two functions as a single function.

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

Function Composition

For example:

```
odd :: Int -> Bool  
odd = not . even
```



We call this type of notation point free notation

Functions composed out of other function that do not mention their actual argument are called point free.

Point free notation has nothing to do with the absence of the (.) operator!

Application Operator

The library function (\$) applies its second argument to its first argument.

$$\begin{aligned} (\$) &:: (a \rightarrow b) \rightarrow a \rightarrow b \\ f \$ x &= f x \end{aligned}$$

Application Operator

The application operator has low, right associative binding precedence. So it can sometimes be used to omit parentheses.

```
f $ g (+1) $ [1,2] ++ [4,5]
```

=

```
f (g (+1) ([1,2] ++ [4,5]))
```

Many More Operators

The GHC comes with a huge library with useful functions

- One module is always in scope the Prelude
- Other modules need to be imported
 - For Example:

```
import Data.List
```

Many More Operators

Documentation for these libraries on the GHC site:
<http://www.haskell.org/ghc/docs/7.0.3/html/libraries/index.html>

Looking for a specific function?

Hoogle

The logo for Hoogle, a Haskell API search engine. It features the word 'Hoogle' in a stylized font where the 'H' is blue, the 'o' is red, the 'l' is yellow, the 'o' is blue, and the 'e' is green. Below the word, the text 'HASKELL API SEARCH' is written in a smaller, black, sans-serif font.

Quiz

- Can we find a function that takes a function that gets two arguments and changes the order they are expected in (using Hoogle)?

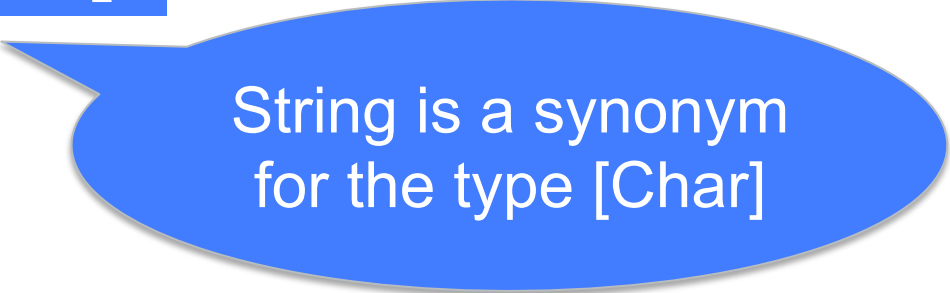
Based on slides of Graham Hutton

DECLARING TYPES

Type Declarations

In Haskell, a new name for an existing type can be defined using a type declaration.

```
type String = [Char]
```



String is a synonym
for the type [Char]

Type Declarations

Type declarations can be used to make other types easier to read. For example:

```
type Pos = (Int, Int)
```

Type Declarations

Now we can define:

```
origin :: Pos
origin = (0,0)

left    :: Pos → Pos
left (x,y) = let x' = x - 1
              in (x',y)
```

Type Declarations

Like function definitions, type declarations can also have parameters. For example:

```
type Pair a = (a, a)
```

Type Declarations

Now we can define:

```
mult      :: Pair Int → Int
mult (m,n) = m*n

copy      :: a → Pair a
copy x    = (x,x)
```

Type Declarations

Type declarations can also have multiple parameters.
For example:

```
type Quadruple a b c d = (a, b, c, d)
```

Type Declarations

Type declarations can be nested:

```
type Pos    = (Int,Int)
type Trans = Pos → Pos
```



Type Declarations

Type declarations cannot be recursive:

```
type Tree = (Int, [Tree])
```



Data Declarations

A completely new type can be defined by specifying its values using a data declaration.

```
data Bool = False | True
```

Bool is a new type,
with two new values
False and True.

Data Declarations

- The two values False and True are called the constructors for the type Bool.
- Type and constructor names must begin with an uppercase letter.

Type and Value Constructors

```
data Bool = False | True
```

Bool is a type
constructor

True and False are
value constructors

Type and Value Constructors

```
data Singleton = Singleton
```

Singleton is a type
constructor

Singleton is a value
constructor

Sum Types

```
data Bool = False | True
```

A type with multiple constructors is a sum type

Like C union-types

Data Declarations

Values of new types can be used in the same ways as those of built in types. For example:

```
data Answer = Yes | No | Unknown
```

Data Declarations

We can now define:

```
answers      :: [Answer]
answers      = [Yes, No, Unknown]

flip         :: Answer → Answer
flip Yes     = No
flip No      = Yes
flip Unknown = Unknown
```

Data Declarations

Or alternatively:

```
answers      :: [Answer]
answers      = [Yes, No, Unknown]
```

```
flip :: Answer → Answer
```

```
flip a = case a of
    No → Yes
    Yes → No
    Unknown → Unknown
```

Data Declarations

The constructors in a data declaration can also have parameters. For example:

```
data Shape = Circle Float  
           | Rect Float Float
```

A type with type constructors with multiple arguments is a product type

Data Declarations

Now we can define:

```
square          :: Float → Shape
square n        = Rect n n

area            :: Shape → Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

Data Declarations

We can consider the two constructors as functions:

```
Circle :: Float → Shape
```

```
Rect   :: Float → Float → Shape
```

Data Declarations

Data declarations can also have parameters.
For example:

```
data Maybe a = Nothing | Just a
```

Data Declarations

We can now define:

```
safediv    :: Int → Int → Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

safehead   :: [a] → Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

Data Declarations

Data declarations can also have multiple parameters.
For example:

```
data Either l r = Left l | Right r
```

Data Declarations

We can now define:

```
safehead    :: [a] → Either String a
safehead [] = Left "Empty list"
safehead xs = Right (head xs)

unsafeHead :: [a] → a
unsafeHead xs = case safehead xs of
    (Left err) → error err
    (Right x)  → x
```

Recursive Types

In Haskell, new types can be declared in terms of themselves. That is, type can be recursive.

```
data Nat = Zero | Succ Nat
```

Recursive Types

Using recursion we can now define:

```
int2nat      :: Int → Nat
int2nat 0    = Zero
int2nat n | n > 0 = Succ (int2nat $ n - 1)
           | otherwise = undefined
```

Recursive Types

Using recursive types we can define lists:

```
data List a = Nil  
            | Cons a (List a)
```

Recursive Types

Using recursion we can now define:

```
length      :: List a → Nat
length Nil  = 0
length (Cons _ xs) = 1 + length xs
```

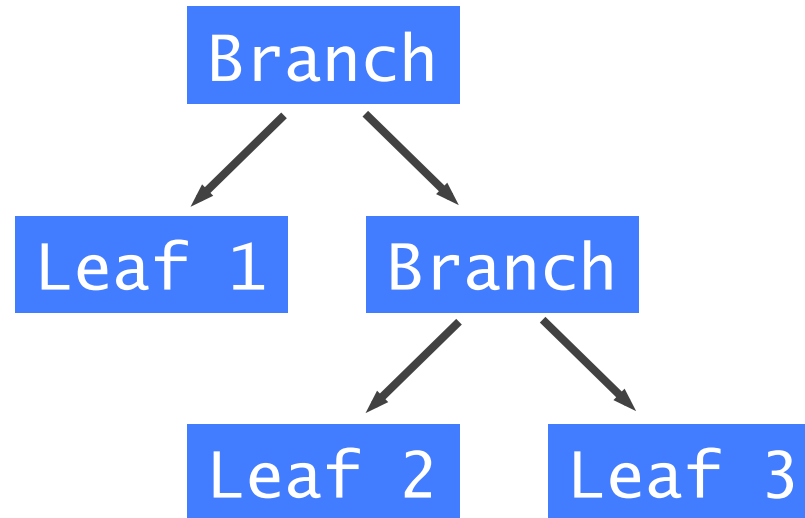
Trees

Binary trees can be represented as follows:

```
data Tree a = Branch (Tree a) (Tree a)
              | Leaf a
```

Trees

Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3))



Trees

Sum all the leaf values of a Tree:

```
sumTree :: Num a => Tree a -> a
sumTree (Leaf a) = a
sumTree (Branch l r) = sumTree l + sumTree r
```

Trees

Flatten a tree into a list

```
flatten :: Tree a -> [a]
flatten (Leaf a) = [a]
flatten (Branch l r) = flatten l ++ flatten r
```

Catamorphisms

There is a pattern in these two functions

```
sumTree :: Num a => Tree a -> a
sumTree (Leaf a)      = a
sumTree (Branch l r) = sumTree l + sumTree r
```

```
flatten :: Tree a -> [a]
flatten (Leaf a)      = [a]
flatten (Branch l r) = flatten l ++ flatten r
```

Catamorphisms

We can generalise this pattern

```
sumTree :: Num a => Tree a -> a
sumTree (Leaf a)      = a
sumTree (Branch l r) = sumTree l + sumTree r
```

f = id

```
foldTree (Leaf a)      = f a
foldTree (Branch l r) = foldTree l ⊕ foldTree r
```

⊕ = +

Catamorphisms

We can generalise this pattern

```
flatten :: Tree a -> [a]
flatten (Leaf a)      = [a]
flatten (Branch l r) = flatten l ++ flatten r
```

$f = (:\![])$

```
foldTree (Leaf a)      = f a
foldTree (Branch l r) = foldTree l  $\oplus$  foldTree r
```

$\oplus = ++$

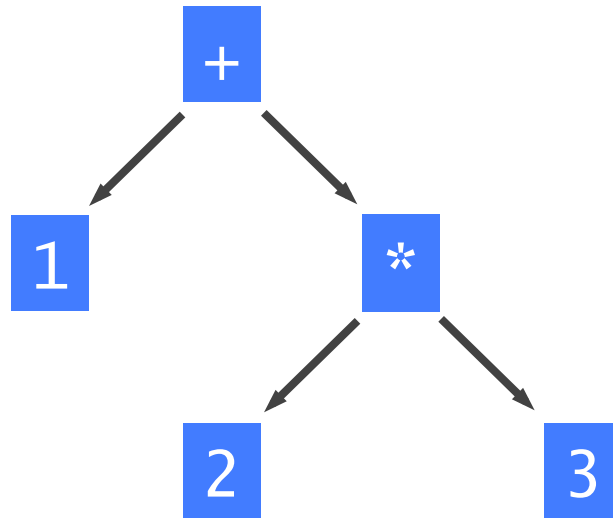
Catamorphisms

```
foldTree :: (a -> b) -> (b -> b -> b) -> Tree a -> b
foldTree f ⊕ (Leaf a)      = f a
foldTree f ⊕ (Branch l r) = foldTree l ⊕
                             foldTree r
```

```
sumTree = foldTree (+) id
flatten = foldTree (++) ([])
```

Arithmetic Expressions

Consider a simple form of expressions built up from integers using addition and multiplication.



Arithmetic Expressions - Quiz

Define a suitable new type to represent arithmetic expressions.

```
data Expr = ???
```

Arithmetic Expressions

Using recursion we can now define:

```
eval      :: Expr → Int
eval (Val n)    = n
eval (Add x y)  = eval x + eval y
eval (Mul x y)  = eval x * eval y
```

Arithmetic Expressions

Using recursion we can now define:

```
size      :: Expr → Int
size (Val n)    = 1
size (Add x y)  = size x + size y
size (Mul x y)  = size x + size y
```

Arithmetic Expressions - Quiz

Many functions can be defined by replacing the constructors by other functions using a suitable fold function.

Define a function `fold` for our `Expr` type such that:

```
eval = fold id (+) (*)
```

Based on slides of Graham Hutton

DECLARING CLASSES

Declaring Classes

In Haskell, a new class can be declared using the class mechanism. For example:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

Declaring Classes

When using these functions their types are actually:

```
(==), (/=) :: Eq a => a -> a -> Bool
```

Declaring Instances

Given this class we can now define an instance for a type. For example:

```
instance Eq Bool where
  False == False = True
  True  == True   = True
  _     == _     = False
```

This only works for types declared using the data mechanism

Declaring Instances

- It is possible to override the default behaviour of functions, simply by defining their behaviour in the instance declaration

Defining Instances

You cannot just define instances for every type!

- No instances for type synonyms
- No instance such `[Int]` or `Maybe Int`
- Write instances for: `SomeClass a => [a]` and
`SomeClass a => Maybe a`

Extending Classes

Classes cannot really be extended, we can however define a sub-class. For example:

```
class Eq a => Ord a where
  (<), (>), (<=), (>=) :: a -> a -> Bool
  min, max           :: a -> a -> a
  min x y | x < y     = x
           | otherwise = y
  ...
```

Extending Classes

An instance `Ord a` can only be defined if an instance `Eq a` is defined!

Deriving Instances

For some classes Haskell can generate instances!
(Show, Eq, Ord, Read, + a few more)

```
data List a = Nil  
           | Cons a (List a)  
deriving (Show, Eq)
```

Deriving Instances

The generated instances restrict the types `a` to be a member of the same class!

```
show :: Show a => [a] -> String
(==) :: Eq a => [a] -> [a] -> Bool
```

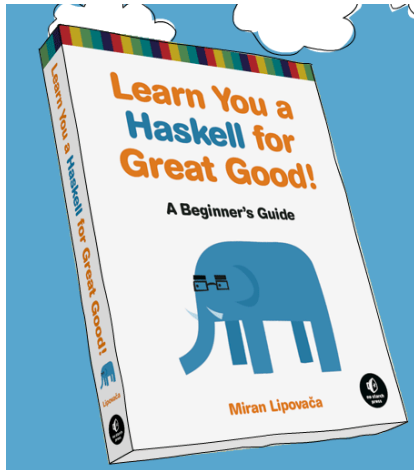
Quiz

- Define an instance of Eq for the type list:

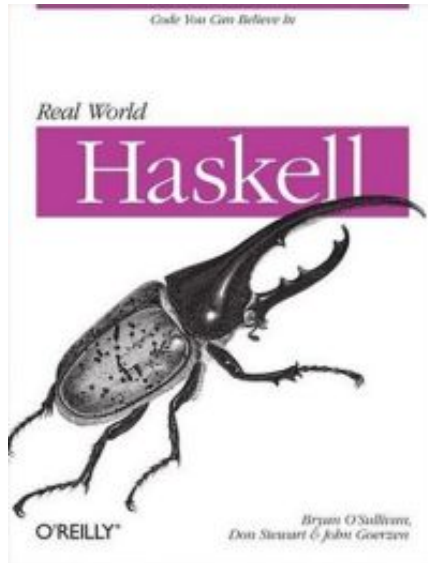
```
data List a = Nil | Cons a (List a)
```

- Define a Class Map that defines a function mmap with type: $\text{mmap} :: \text{Map } t \Rightarrow a \rightarrow b \rightarrow t \ a \rightarrow t \ b$
- Define instances of Map for List a, Maybe a, Either a b, Tree a

Reading material



- Learn you a Haskell:
 - Chapter 6
 - Chapter 8



- Real world Haskell
 - Chapter 3
 - Chapter 6