# Haskell working group

session 2

Rémi and Dhananjay

# Recap

# Types

```haskell
42 :: Int
42.0 :: Float
'a' :: Char
[1, 2] :: [Int] -- List of Ints
"Hello" :: [Char] -- :'(
length :: [a] -> Int
```

# Functions

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs

map (+1) [1, 2]
-- evaluates to [2, 3]
```

# Some more functions

```haskell
-- fold
foldl :: (b -> a -> b) -> b -> [a] -> b

sumList :: [Int] -> Int
sumList l = foldl (+) 0 l
```

# Some lists

```
Prelude> []
Prelude> [1]
Prelude> [1, 2]
Prelude> [1..5]
Prelude> 1:2:[3, 4, 5]
```

# Creating New Data types

### Algebraic Data types

Create new from current..

### Start with caps!!

# Enumeration Types

```
data Fruit = Banana | Apple | Orange
             deriving (Show, Eq)
```

# No more null BS

```
Prelude> Apple :: Fruit
Prelude> NoFruit :: Fruit

<interactive>:19:1: Not in scope: data constructor ‘NoFruit

<interactive>:19:12:
    Not in scope: type constructor or class ‘Fruit’
```

# Just like Haskell primitive types

```haskell
data Bool = True | False
data List a = [] | a : List a
data Int = 1 | 2 | 3 | 4 | 5 ...
-- Just conceptually...
```

# Embedding Results

```
data Result = Failure | OK Double
                       deriving (Show)
```

# Making in more generic

```
data Result a = Failure | OK a
```

# SafeDiv

```haskell
safeDiv :: Double -> Double -> Result Double

-- safeDiv 1 0 == Failure
-- safeDiv 4 2 == Ok 2
```

# SafeDiv

```
safeDiv :: Double -> Double -> Result Double

-- safeDiv 1 0 == Failure
-- safeDiv 4 2 == Ok 2
```

`Result` is called `Maybe` in Haskell:

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

# DataTypes - Recap

General syntax:

```
data Name = Constructor1 type11 type12 ...
        | Constructor2 type21 ..
        | Constructor3
```

# DataTypes in Functions

```
showResult :: Result -> String
```

# Pattern matching

We already saw function arguments destructuring:

```
fun [] = ...
fun (x:xs) = ...
```

# case statements

We can do it in function body:

```
fun lst =
    case lst of
        [] -> ...
        [x] -> ...
        (x:xs) -> ...
```

# case statements

We can do it in function body:

```
fun lst =
    case lst of
        [] -> ...
        [x] -> ...
        (x:xs) -> ...
```

General structure:

```
case something of
  case1 -> result1
  case2 -> result2
  ...
  _ -> resultN
```

# Log file parsing

Let's practice!