**Joost 2 `addMebe` Joost3**

**-> Joost 5.0**

**:t** is type

Eg: **:t liftMebe**

**-> liftMebe :: (Double -> Double -> Double) -> Mebe -> Mebe -> Mebe**

(this takes in 3 Double variables and outputs them as Mebe)

Note: we already used the code:

**data Mebe = Joost Double | Nada**

**deriving Show**

to define Mebe as a data type that accepts Joost Double or Nada

If a Double at input = Nada:

the function will return **-> Nada**

**Mulitplication liftMebe(\*) (Joost 2) (Joost 3)**

-> **Joost 6**

Nada in this function acts like an error

(Haskell is lazy and doesn’t like returning errors)

recipMebe. function

recipMebe gives Nada if Joost 0 is given on input

Prints Joost1/n given input (Joost n) (if n/= 0)

Use **/=** for not equals instead of **!=**

**liftMebe1**

**chainToMebe**‘**:: (Double ->Mebe) -> Mebe -> Mebe**

**chainToMebe**‘ **f Nada = Nada**

(‘ is an identifier)

You can change:

**liftMebe2:: (Double -> Double -> Double ) ->Mebe a-> Mebe b -> Mebe c**

**liftMebe2:: (a->b->c) -> Mebe a Mebe b Mebe c**

Note you will then need to create data types for a, b, & c

Data Mebe changes to:

**dataMebe a = Joost a | Nada**

**deriving Show**

**>==** is a monad class that combines two monadic values in the form of

**:: m a -> (a -> m b) -> m b**

where **m** is **a** monadic value holding **a** (a monad instance).

**a** produces **m b,** anew monadic instance then returns **m b**

So **(>>=) :: Maybe a -> (a->Maybe b) -> Maybe b**

Means: what could be an error will result in a **Double b** or a **Nada b**

**[1,2,3] >>= (\x -> [5\*x, 7\*x])**

**-> [5,7,10,14,15,21]** printing possibilities from binding list to value \x

List of possible values from (1,2,3)(5,7)

Where “possible a” is [a], list of a’s

**Return x >>=f** is the same as writing f(x)

*( f x) in Haskell*

IO Haskell: [data IO a note]

Takes a character from old world into new function. [Take word, give world + value]

Output character:: Char -> (World -> World, ( , ,) *void value*

Input character :: World -> (World, Char)

So openfile String -> (World -> (World, ()) (*Input/output action)*

So chaining?

**:t getChar >>= putChar**

**-> getChar >>= putChat :: IO (IO action)**

Haskell yields to input/output actions, like java (wait on user for input to give output)

Haskell also allows you to modify IO actions, unlike Python where you can only call it. (eg: you can output 3 a character times from 1 input)

: I/O systems in Haskell : purely functional & has power in conventional programming languages

**:t getChar**

**-> get Char:: IO Char ->** *action to return character*

**:t putChar**

**->:putChar :: Char -> IO()**  returns no interesting values