Feel and See Haskell Basics

This presentation is inspired by Bartosz Milewski and Alexander Grothendieck for who seeing and understanding is the same thing. I am trying to develop a very basic feeling of what Haskell has to show to every programmer. Apart basic functions, lambda constructs, or schematics, I am using a very terse expression on purpose, because each concept has already a presentation on internet, just follow the links. Here, the path is as important as the purpose.

Read–Eval–Print Loop

On Windows : https://www.haskell.org/platform/windows.html

It is better to follow the script with ghci.

Foldr/Foldl (on lists)

Reducing a list, for example a reduction of the sum of the elements using foldr.

```
Prelude> foldr (+) 0 [1..10]
55
Prelude>
```

This is the sum of the n first numbers.

```
Prelude> let f x y = x + y
Prelude> foldr f 0 [1..10]
55
Prelude>
```

Generalization using a function f.

```
Prelude> let f = (+)
Prelude> foldr f 0 [1..10]
55
Prelude>
```

Defining a function without specifying parameters.

```
Prelude> foldr (\x y -> x + y) 0 [1..10]
55
Prelude>
```

Or using a lambda, the anonymous function.

Identity of a list

```
Prelude> foldr (\x y -> x:y) [] [1..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude>
```

Identity is a very useful concept.

```
Prelude> foldl (\y x -> x:y) [] [1..10]
[10,9,8,7,6,5,4,3,2,1]
Prelude>
```

foldl is opposed to foldr, and gives a reversed list.

Mapping of a function on a list

```
Prelude> foldr (\x y -> (x+1):y) [] [1..10]
[2,3,4,5,6,7,8,9,10,11]
Prelude>
```

The identity form helps us to map a function, in this case (+1).

```
Prelude> let f = (+1)
Prelude> foldr (\x y -> (f x):y) [] [1..10]
[2,3,4,5,6,7,8,9,10,11]
Prelude>
```

Generalization with the function f.

```
Prelude> let jpmap f l = foldr (\x y -> (f x):y) [] l
Prelude> jpmap (+1) [1..10]
[2,3,4,5,6,7,8,9,10,11]
Prelude>
```

Finally jmap could be a usefull function, taking a function as a first argument, why not ?

Functorial mapping from Prelude

```
Prelude> fmap (+1) [1..10]

[2,3,4,5,6,7,8,9,10,11]
Prelude>
```

It seems, after all, to be a pretty big concept.

First useful example ?

```
Prelude> let evens = foldr (\x y -> if (x `mod` 2 == 0) then x:y else y) [] [1..10]
Prelude> evens
[2,4,6,8,10]
Prelude> let lis = fmap (\x -> "<li>"++(show x)++"</li>") evens
Prelude> lis
["<li>2</li>","<li>4</li>","<li>6</li>","<li>8</li>","<li>10</li>"]
Prelude> foldr (++) "" lis
"<li>2</li><li>4</li><li>6</li><li>8</li><li>10</li>"
Prelude>
```

Could be usefull in some place.

Fun

```
Prelude> foldr (\x y -> x) 0 [11..21]
11
Prelude> foldl (\y x -> x) 0 [11..21]
21
Prelude>
```

Is this the head and tail of a list ?

```
Prelude> foldr (\x _ -> x) 0 [11..21]
11
Prelude> foldl (\_ x -> x) 0 [11..21]
21
Prelude>
```

Actually, better to tell everybody, if you don't care about me use undescore _.

```
Prelude> foldr (\_ y -> y+1) 0 [11..21]
11
Prelude> foldl (\y _ -> y+1) 0 [11..21]
11
Prelude>
```

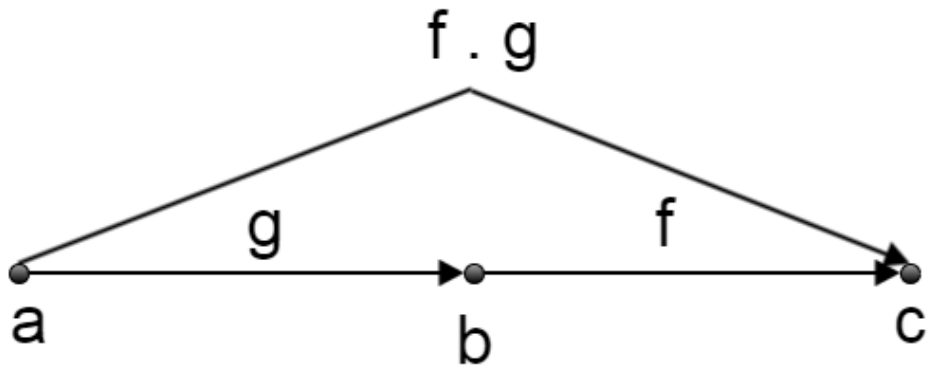
The length of the list.

```
Prelude> foldl (\y x -> [x]++y) [] [11..21]
[21,20,19,18,17,16,15,14,13,12,11]
Prelude>

Prelude> foldr (\x y -> y++[x]) [] [11..21]
[21,20,19,18,17,16,15,14,13,12,11]
Prelude>
```

Once again the reverse function.

Composition of function




```
Prelude> :{

Prelude| let compose:: (b -> c) -> (a -> b) -> a -> c

Prelude|     f `compose` g = \x -> f (g x)

Prelude| :}

Prelude> ((+1) `compose` (*2)) 3

7
Prelude>

Prelude> let after = compose
Prelude> ((+1) `after` (*2)) 3
7
Prelude>

Prelude> let before = flip after
Prelude> ((*2) `before` (+1)) 3
7
Prelude>
```

Function composition is a powerfull concept that is easy to implement.

(b -> c) is function f and (a -> b) is function g, the function composition returns a function from (a -> c).

```
Prelude> ((+1) . (*2)) 3
7
Prelude>
```

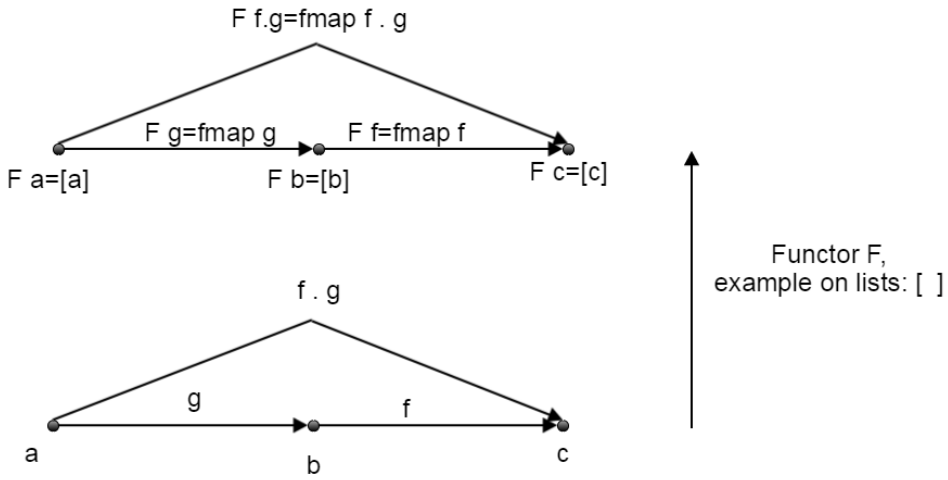
The Prelude dot **.** represents composition of functions.

Functors

```
Prelude> fmap ((+1) . (*2)) [1..3]
[3,5,7]
Prelude>

Prelude> ((fmap (+1)) . (fmap (*2))) [1..3]
[3,5,7]
Prelude>
```

fmap on lists, which complies to the functoriality requirement of preseving function compostion.



In short, functors preserve composition.

Endless lists

```
Prelude> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,5
5,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100,101,102,103,104,1
05,106,107,108,109,110,111,112,113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,128,129
```

Endless lists of integers.

```
Prelude> let infinity = [1..]
Prelude> let infinity2 = infinity++infinity
Prelude>
```

Let's define one endless list after another.

Lazyness

```
Prelude> foldr (\x y -> y+1) 0 [[1..],[1..],[1..]]

3
Prelude>
```

In this example, endless loops are not evluated to be counted, better to be lazy with infinity...

In Haskell, most Functors are also Monads

```
Prelude> :info []
data [] a = [] | a : [a] -- Defined in ‘GHC.Types’
instance Applicative [] -- Defined in ‘GHC.Base’
instance Eq a => Eq [a] -- Defined in ‘GHC.Classes’
instance Functor [] -- Defined in ‘GHC.Base’
instance Monad [] -- Defined in ‘GHC.Base’
instance Monoid [a] -- Defined in ‘GHC.Base’
instance Ord a => Ord [a] -- Defined in ‘GHC.Classes’
instance Show a => Show [a] -- Defined in ‘GHC.Show’
instance Read a => Read [a] -- Defined in ‘GHC.Read’
instance Foldable [] -- Defined in ‘Data.Foldable’
instance Traversable [] -- Defined in ‘Data.Traversable’
Prelude> :info Maybe
data Maybe a = Nothing | Just a -- Defined in ‘GHC.Base’
instance Applicative Maybe -- Defined in ‘GHC.Base’
instance Eq a => Eq (Maybe a) -- Defined in ‘GHC.Base’
instance Functor Maybe -- Defined in ‘GHC.Base’
instance Monad Maybe -- Defined in ‘GHC.Base’
instance Monoid a => Monoid (Maybe a) -- Defined in ‘GHC.Base’
instance Ord a => Ord (Maybe a) -- Defined in ‘GHC.Base’
```

```
instance Show a => Show (Maybe a) -- Defined in ‘GHC.Show’
instance Read a => Read (Maybe a) -- Defined in ‘GHC.Read’
instance Foldable Maybe -- Defined in ‘Data.Foldable’
instance Traversable Maybe -- Defined in ‘Data.Traversable’
Prelude>
```

```
[] is a Functor and a Monad
```

Maybe is a Functor and a Monad

```
Prelude> let uplift = return;
Prelude> let f :: Int -> [Int];f = (uplift . (*2));
Prelude> let g :: Int -> [Int];g = (uplift . (+1))
Prelude> [3] >>= f >>= g
[7]
Prelude>
```

```
Prelude Control.Monad> [3] >>= (uplift . (*2)) >>= (uplift . (+1))
[7]
Prelude Control.Monad>
```

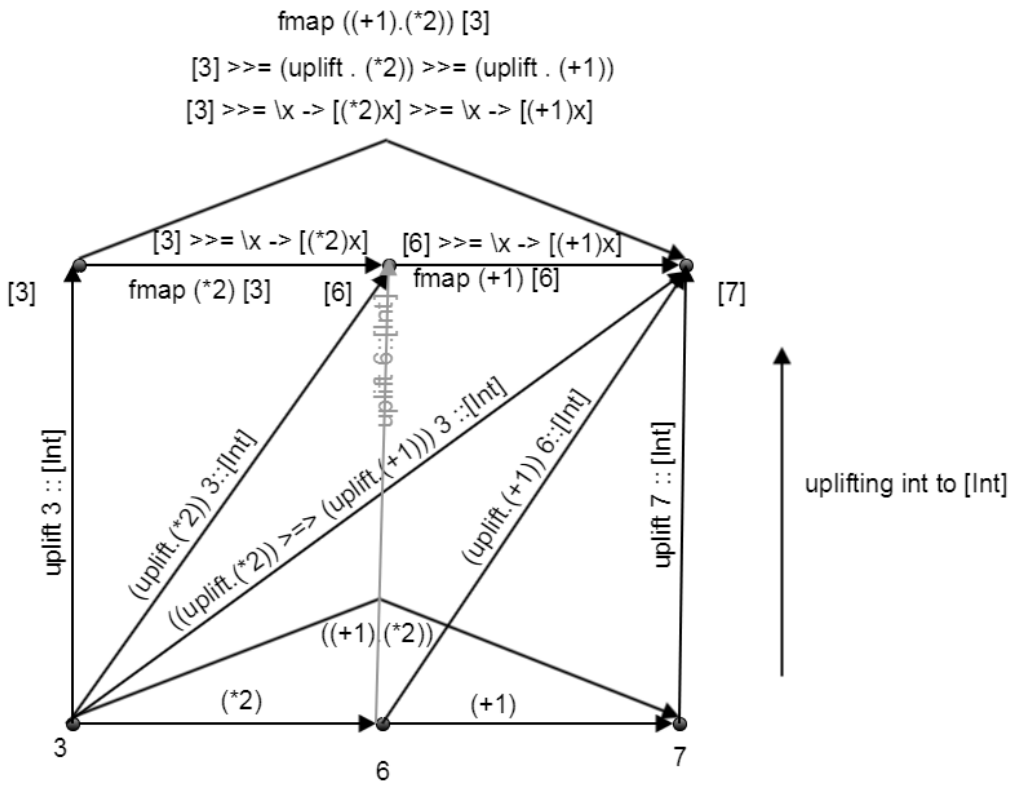
Monads are usefull to work in a context, in this case a list.

```
Prelude> import Control.Monad
```

```
Prelude Control.Monad> let h = f >=> g
```

```
Prelude Control.Monad> [3] >>= h
[7]
Prelude Control.Monad>
```

Functions to be done in the context can be composed.

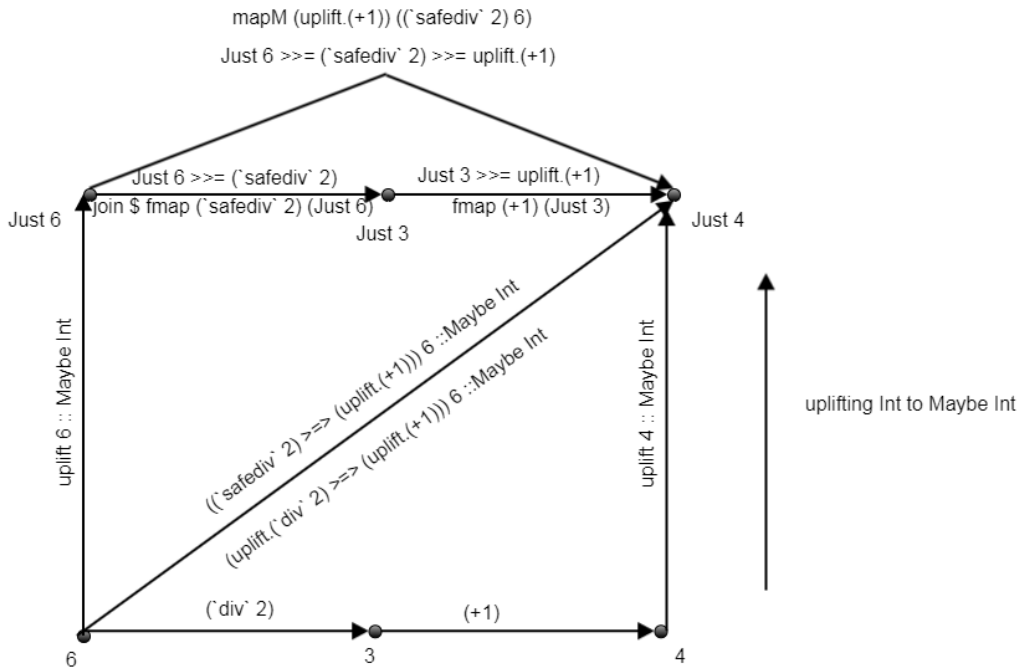


Composition using monads.

Monads allow to chain partial functions:

```
Prelude Control.Monad> 12 `div` 2
6
Prelude Control.Monad> 12 `div` 0
*** Exception: divide by zero
Prelude Control.Monad> let safediv::Int->Int->Maybe Int;safediv _ 0 = Nothing;safediv x y = Just (x `div`y);
Prelude Control.Monad> 12 `safediv` 2
Just 6
Prelude Control.Monad> 12 `safediv` 0
Nothing
Prelude Control.Monad> (12 `safediv` 0) >>= (uplift.(+1))
Nothing
Prelude Control.Monad> (+1) (12 `div` 0)
*** Exception: divide by zero
Prelude Control.Monad> (12 `safediv` 2) >>= (uplift.(+1))
Just 7
Prelude Control.Monad>
```

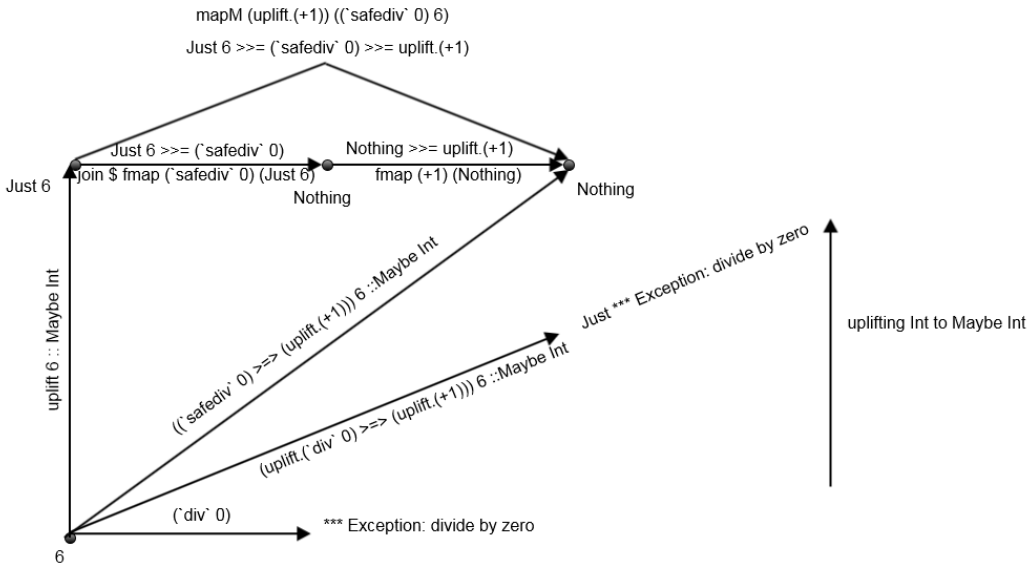
In a program, it is better to say Nothing than trying to divide by zero.



Clearly uplift means going up in the program.

```
Prelude System.Environment Control.Monad> 6 `div` 0
*** Exception: divide by zero
Prelude System.Environment Control.Monad>

Prelude System.Environment Control.Monad> (uplift.(`div` 0) >=> (uplift.(+1))) 6 ::Maybe Int
Just *** Exception: divide by zero
Prelude System.Environment Control.Monad>
```



Once again, it is better to say Nothing than throwing an Exceptions because they break composition.

Point free style

```
Prelude Control.Monad> let h = (+1).(*2)

Prelude Control.Monad> h 3
7
Prelude Control.Monad>
```

This programming style is based on function and functor composition.

```
Prelude Control.Monad> let h' = ((uplift.(*2)) >=> (uplift.(+1)))
Prelude Control.Monad> h' 3::[Int]
[7]
Prelude Control.Monad>
```

Only the starting point is used. The intermediate points are anonymous.

From bottom to top with recursion and corecursion

```
Prelude> let bottom = bottom
Prelude> bottom
Interrupted.
Prelude>
```

The only thing that does this program is looping, and consume cpu power. It must be killed from outside to stop.

```
Prelude> :m + Debug.Trace

Prelude Debug.Trace> let f = (trace ".") f

Prelude Debug.Trace> f
.
.

.
Interrupted.
Prelude Debug.Trace>
```

Same as above, but with trace.

```
Prelude Debug.Trace> let f x = (trace (show x)) f (x+1)
Prelude Debug.Trace> f 0
0
1
2
3
4
5
6
7
8

7800
7801
Interrupted.
Prelude Debug.Trace>
```

Nice, we can add up integers, but nothing stops.

```
Prelude Debug.Trace> let f 0 = 0; f x = (trace (show x)) f (x-1)
Prelude Debug.Trace> f 3
3
2
1
0
Prelude Debug.Trace>
```

Let's decrement to zero, at least this program stops by itself.

```
Prelude Debug.Trace> let f 0 = 0; f x = (trace (show x)) x + f (x-1)
Prelude Debug.Trace> f 10
1
2
3
4
5
6
7
8
9
10
55
Prelude Debug.Trace>
```

First result ! I can add up numbers.

```
Prelude Debug.Trace> let f g 0 = 0; f g x = (trace (show x)) x `g` f g (x-1)
Prelude Debug.Trace> f (+) 10
1
2
3
4
5
6
7
8
9
10
55
Prelude Debug.Trace>
```

Let's generalize a bit.

```
Prelude Debug.Trace> let f g [] = 0; f g (x:xs) = (trace (show x)) x `g` f g xs
Prelude Debug.Trace> f (+) [1..10]
10
9
8
7
6
5
4
3
2
1
55
Prelude Debug.Trace>
```

Generalization on lists.

```
Prelude Debug.Trace> let f g a [] = a; f g a (x:xs) = (trace (show x)) x `g` f g a xs
Prelude Debug.Trace> f (+) 0 [1..10]
10
9
8
7
6
5
4
3
2
1
55
Prelude Debug.Trace> foldr (+) 0 [1..10]
55
Prelude Debug.Trace>
```

Congrats, you just understood foldr.

And foldl ?

```
Prelude Debug.Trace> let f 0 = 0;f x = (trace (show x)) f (x-1);
Prelude Debug.Trace> f 3
3
2
1
0
Prelude Debug.Trace> let f a 0 = a;f a x = (trace (show x)) f (x+a) (x-1);
Prelude Debug.Trace> f 0 3
3
2
1
6
Prelude Debug.Trace>
```

Using an accumulator in the parameters, I can add up numbers too.

```
Prelude Debug.Trace> let f a [] = a;f a (x:xs) = (trace (show x)) f (x+a) xs;
Prelude Debug.Trace> f 0 [1..3]
1
2
3
6
Prelude Debug.Trace>
```

And I can generalize to lists.

```
Prelude Debug.Trace> let f g a [] = a;f g a (x:xs) = (trace (show x)) f g (x `g` a) xs;
Prelude Debug.Trace> f (+) 0 [1..3]
1
2
3
6
Prelude Debug.Trace> foldl (+) 0 [1..3]
6
Prelude Debug.Trace>
```

Congrats, you understood foldl.

And what about fmap ?

```
Prelude Debug.Trace> let f g a [] = a; f g a (x:xs) = (trace (show x)) x `g` f g a xs
Prelude Debug.Trace> f (:) [] [1..3]
[1,2,3]
Prelude Debug.Trace> 1
2
3

Prelude Debug.Trace>
```

Let start with the indentity.

```
Prelude Debug.Trace> let i = (f (:) [])

Prelude Debug.Trace> i [1..3]
[1,2,3]
Prelude Debug.Trace> 1
2
3

Prelude Debug.Trace>
```

Generalizing with a function i.

```
Prelude Debug.Trace> let i = (f (\x a -> x:a) [])

Prelude Debug.Trace> i [1..3]
1
2
3
[1,2,3]

Prelude Debug.Trace>
```

Rewriting identity to get more specific.

```
Prelude Debug.Trace> let i g = (f (\x a -> (g x):a) [])

Prelude Debug.Trace> i (+1) [1..3]
1
2
3
[2,3,4]

Prelude Debug.Trace>
```

Generalizing by adding a function g.

```
Prelude Debug.Trace> fmap (+1) [1..3]

[2,3,4]
Prelude Debug.Trace>
```

Congrats, you understood fmap on lists.

May be it's time to read real stuff, with Real World Haskell. And to learn more Category Theory.