The monad pattern

- Functors (or mappables) cover an important and very common pattern: A structure preserving transformation of a container type.

Examples:

$$\left.\begin{array}{lcl} \text{Some } x & \xrightarrow{map} & \text{Some } x' \\ \text{None} & \rightarrow & \text{None} \end{array}\right]$$

$$[1; 2; 3] \xrightarrow{map} [\text{"one"}; \text{"two"}; \text{"three"}]$$

- There is another very common and related pattern, where the function being mapped returns a new container of the same type:

$$\boxed{M.map\ (f : 'a \rightarrow M\langle 'b\rangle)\ (x : M\langle 'a\rangle) : M\langle M\langle 'b\rangle\rangle}$$

- This results in the resut being "wrapped" in an extra level of the container.

- This may or may not be what you want (eg. Some (Some 42) or [[1; 2]; [3; 4]])

- Let us consider Option<'a> :

let f x = if x < 0 None else Some x

Option.map f (Some 1) => Some (Some 1)

None => None

- Some (Some 1) is clearly redundant and does not relly make any sense. So what can we do about it?

let flatten = function
| Some x -> x
| None -> None

flatten (Some (Some 1)) => Some 1

None => None

- we can compose the two functions:

$\boxed{\text{let flatMap f x = map f >> flatten}}$

This is the <u>monad</u> pattern!

Note! flatten is often called <u>join</u>
and flatMap is called <u>bind</u>.

- we can gain a bit more insight into the soul of the monad by looking at the type signatures in both prefix and infix:

bind : (a -> M<b>) -> M<a> -> M<b>

we can flip arguments freely, in infix:

(>>=) : M<a> -> (a -> M<b>) -> M<b>

compare with

(|>) : a -> (a -> b) -> b

(|>) and (>>=) are cousins!!

- How about ordinary function composition?

(>>) : (a -> b) -> (b -> c) -> (a -> c)

(>=>) : (a -> M<b>) -> (b -> M<c>) -> a -> M<c>

↑ kleisli composition

- People sometimes struggle with monads, asking what they are. The question is wrong, it is <u>when</u> is a monad!

- Let us consider Option<a> and the pyramid of doom:

let f, g and h be functions whic can fail

f, g, h : a → Option<b> and let happy be a function which depends on the results of f, g and h:

1
```
r1 = f(x1)
if ok(r1):
    r2 = g(x2)
    if ok(r2):
        r3 = h(x3)
        if ok(r3):
            happy(r1, r2, r3)
```

2
```
f x1 >>= (fun r1 ->
    g x2 >>= (fun r2 ->
        h x3 >>= (fun r3 ->
            Some (happy r1 r2 r3)
        )
    )
)
```

3 Computation expr.
```
option {
    let! r1 = f x1
    let! r2 = g x2
    let! r3 = h x3
    return happy r1 r2 r3
}
```

3 is syntactic sugar for 2!

The applicative functor pattern

- In the previous example none of the failable functions were dependent on the result of an earlier function (except happy of course!)
- This is a pattern! The applicative functor!

```
let pure : a -> Option<a> = Some
let apply : Option<a -> b> -> Option<a> -> Option<b>
let (<*>) = apply
```

Using these functions the previous example can be witten:

```
pure happy <*> f x1 <*> g x2 <*> h x3
```

We can even define an inline operator

```
let (<!>) = Option.map
```

```
=> happy <!> f x1 <*> g x2 <*> h x3
```

Now, that is beautiful!

(also liftA, liftA2, liftA3 ...)

The monad laws

- So far we have only looked at the patterns. But in order to be a proper monad the functions must satisfy a set of rules (as for functors):

let return : a -> M<a> = ...

1. Left identity : return a >>= f ≡ f a
2. Right identity : m >>= return ≡ m
3. Associativity : (m >>= f) >>= g ≡ m >>= (fun x -> f x >>= g)

(Kleisli 3 : (f >=> g) >=> h ≡ f >=> (g >=> h))

Computation expressions

- Because monads can be a bit clumsy and verbose to work with, F# provides syntactic sugar for them, called <u>computation expressions</u>

Ex. async, seq, query

- We can define our own CEs!

- Computation expressions are much richer than the monad typeclass in Haskell
- They provide a rich syntax for <u>sequencing</u> and <u>composing</u> computations.
- CE:s have special syntax/keywords for flattening or joining results from computations:

$\boxed{\texttt{let!, do!, return!, match!, yield!}}$

- These get desugared into continuations, eg.

`{ let! x = f z }` ⇒ `M.bind (f z, fun x -> ...)`

Example:

```
option {
    let x1 = 1
    let! a = f x1
    match! g a with
      | (0, 1) -> return 0
      | (x, y) -> return! h x y
}
```

f : int → Option<int>

g : int → Option<int * int>

h : int → int → Option<int>

- We can implement our own CE:s by defining and instantiating a so called builder class:

```
type OptionBuilder () =
    member x.Bind (e, c) = Option.bind e c
    member x.Return v = Some v
    member x.ReturnFrom v = v

let option = OptionBuilder ()
```

- CE:s can define a lot more:

  - for loops
  - while loops
  - "iterators" using yield
  - exception handling

The list/seq monad:

```
let flatten m =
    Seq.fold (fun a x ->
        Seq.append a x) Seq.empty m

let bind f = Seq.map f >> flatten
let return = Seq.singleton
```

```
seq {
    let! a = [1;2]
    let! b = [3;4]
    return (a,b)
}
```

=> [(1,3); (1,4); (2,3); (2,4)]