INF-3910 Lecture 10

Interlude

- we have now covered all the major abstractions (patterns) of functional programming!
  functions, currying, partial application, higher-order functions, generics, functors, applicative functors, monads and (function) composition.

- we can put these abstractions to use by recognizing patterns and repetitions in our codes

- Let's assume the following functions

```
lines   : string -> [string]   // chop into lines
unlines : [string] -> string   // concat into lines
( words : char -> string -> [string] )
( unwords : char -> [string] -> string )
```

```
let process t = unlines (sort (lines t))
    process' t = t |> lines |> sort |> unlines
    process" t = (lines >> sort >> unlines) t
    process'''  = lines >> sort >> unlines
```

INF-3910 Lecture 10

```
let sortLines = lines >> sort >> unlines
let reverseLines = lines >> reverse >> unlines
let twoFirstLines = lines >> take 2 >> unlines
```
↖ partial application!

- There is a pattern we can factor out using a higher-order function:

```
let byLines (f: [string]->[string]) = unlines >> f >> lines

let sortLines = byLines sort
let reverseLines = byLines reverse
let twoFirstLines = byLines (take 2)

let indent s = "    " + s
let rec repeat n f = if n>0 then
                        repeat (n-1) (f >> f) //TCO!
                     else f

let indentEachLine = byLines (map indent)

let indentEachLineN n =
    byLines (map (repeat n indent))
```

new patterns!

```
let eachLine f = byLines (map f)
let eachLineN n f = eachLine (repeat n f)
```

- But what if lines/unlines can fail?

```
lines : string -> [string] option
unlines: [string] -> string option
```

- Now things don't compose anymore!

```
let sortLines t = t |> lines |> ??
let sortLines' t = t |> lines >>= fun x ->
   Some(sort x) >>= unlines
let sortLines" t =
   Some t >>= lines >>= sort' >>= unlines

let sortLines''' t = (lines >=> sort' >=> unlines) t.
let sortLines'''' = Some >=> lines >=> sort' >=> unlines

let sortLines''''' t =
   option {
      let! l = lines t
      let sl = sort l
      return! unlines sl
   }
                  can fail
let byLines f = Some >=> lines >=> f >=> unlines
let eachLine f = byLines (bind f)
 (what about repeat?)
```

- What if the input data could fail?

```
let sortLines (t : string option) =
    map (byLines sort) t
let sortLines' t = apply (Some (byLines sort)) t
let sortLines'' t = Some (byLines sort) <*> t
```

<u>In summary</u>:

Function application:

```
f x y          <=> A f <*> Ax <*> Ay
map f x        <=> bind f (M x)
x |> f |> g    <=> Mx >>= f' >>= g'
f >> g         <=> f' >=> g'
```

- Seq monad example