Lesson 11

Applicatives and Monads

The motivation for Applicatives is that of applying to containers functions of any arity

but also that of applying (pure) functions to arguments that have effects:
-failure
-non-determinism
-performing I/O

and also generic functions that use applicative operators in Prelude:

```
sequenceA :: Applicative f => [f a]-> f [a]
sequenceA []       =  pure []
sequenceA (x:xs) = pure (:) <*> x <*> sequenceA xs
```

```
getChars ::  Int -> IO String
getChars n = sequenceA (replicate n getChar)
```

Applicative laws

1) pure id <*> x = x
2) pure (g x) = pure g <*> pure x
3) x <*> pure y = pure (\g -> g y) <*> x
4) x <*> (y <*> z) = (pure (.) <*> x <*> y) <*> z

we work out types of both endsides:

1) pure id :: f (a -> a), x :: f a
2) g :: a -> b e x :: a => pure (g x) :: f b

pure g :: f (a -> b), pure x :: f a, pure g <*> pure x :: f b

3) x :: f (a -> b), y :: a, pure (\g -> g y) :: f ((a -> b) ->b)

f ((a -> b) -> b) <*> x :: f b

4) x <*> (y <*> z) = (pure (.) <*> x <*> y) <*> z

y :: f (a -> b), z :: f a, (y <*> z) :: f b
x :: f (b -> c), (x <*> (y <*> z)) :: f c

pure (.) :: f (b -> c) -> (a -> b) -> (a -> c)
pure (.) <*> x :: f (a -> b) -> (a -> c)
pure (.) <*> x <*> y :: f (a -> c)
(pure (.) <*> x <*> y) <*> z :: f c

pure :: a -> f a
embeds a pure value into the pure fragment of an effectful world of type f a (impure)

the laws show that any correct expression with pure and <*> can be rewritten in applicative style,
pure g <*> x1 <*> x2 <*>….<*> xn

it is always true that

```
fmap g x = pure g <*> x
```

```
fmap :: (a -> b) -> f a -> f b
```

```
\ g x -> pure g <*> x :: (a -> b) -> f a -> f b
```

the same type !! By unicity of fmap they are the same function attention with [], (<*>) :: [a -> b] -> [a] -> [b]

pure g = [g] a list with 1 function only

new notation: fmap g x = g <$> x

applicative style: g <$> x <*> y <*> z

Monads
we start with one example

```
data Expr = Val Int | Div Expr Expr

eval :: Expr -> Int
eval  (Val n)   =  n
eval (Div x y) =  eval x `div` eval y
```

possible fatal exception

```
eval (Div (Val 1) (Val 0))
***Exception: divide by zero
```

```
safediv :: Int -> Int -> Maybe Int
safediv _ 0  = Nothing
safediv n m = Just (n `div` m)
```

using it, we write an evaluator that is able to handle div by 0

```
eval :: Expr -> Maybe Int
eval (Val n)    = n
eval (Div x y) =case eval x of
                    Nothing -> Nothing
                    Just n     -> case eval y of
                                     Nothing -> Nothing
                                     Just m -> safediv n m
```

clearly with this eval,

```
eval (Div (Val 1) (Val 0))
Nothing
```

but eval is ugly, since Maybe is Applicative, we could try to write eval in applicative style

```
eval :: Expr -> Maybe Int
eval (Val n)    = Just n
eval (Div x y) = pure safediv <*> eval x <*> eval y
```

But is not type correct !! safediv :: Int -> Int -> Maybe Int whereas we would need a Int -> Int -> Int

in the applicative style we can apply pure functions (as Int -> Int -> Int) to effectful arguments
but safediv may itself fail!

in eval there is a pattern that repeats

```
case eval x of
    Nothing -> Nothing
    Just m      -> case eval y of
                        Nothing ->  Nothing
                        Just n      -> safediv n m
```

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b

mx >>= f = case mx of
                    Nothing -> Nothing
                     Just x    -> f  x
```

bind operator

```
eval :: Expr -> Maybe Int
eval (Val n)     = Just n
eval (Div x y)  = eval x >>= \n ->
                    eval y >>= \m ->
                     safediv n m
```

generalizing:

```
m1 >>= \x1 ->
m2 >>= \x2 ->
.
.
mn >>= \xn ->
f x1 x2...xn

do x1 <- m1
   x2 <- m2
    ........
   f x1 ... xn
```

```
eval :: Expr -> Maybe Int
eval (Val n)     =  Just n
eval (Div x y) = do n <- eval x
                    m <- eval y
                    safediv n m

class Applicative m => Monad m where
return :: a -> m a
(>>=) :: m a -> (a -> m b) -> m b

return = pure
```

Examples

```
instance Monad Maybe where
-- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing  >>= _    = Nothing
(Just  x)  >>= f     = f x

return = pure
```

```
instance Monad [] where
-- (>>=) :: [a] -> ( a -> [b]) -> [b]
xs >>= f = [y | x <-xs, y <- f x]


pairs :: [a] -> [b] -> [(a,b)]
pairs xs ys = do x <- xs
                 y <- ys
                 return (x,y)
```

Also IO type is a Monad
the definitions of return and (>>=) are built-in to the language.