1) Functors
2) Applicative
3) Monads

eg: a

```
class Functor t where
    fmap :: (a -> b) -> t a -> t b
```

```
instance Functor [] where
    fmap :: (a -> b) -> [a] -> [b]
    fmap = map
```

```
list :: Type -> Type
[] :: [a]
[] :: Type -> Type
[] Int = [Int]
[] Bool = [Bool]
```

~~fmap~~ fmap id = id

fmap f . fmap g = fmap (f . g)

} These laws should be followed by the programmer

```
instance Functor Maybe where
    fmap :: (a -> b) -> Maybe a -> Maybe b
    fmap f (Just a) = Just (f a)
    fmap f Nothing  = Nothing
```

(.) :: (b → c) → (a → b) → (a → c)

($) :: (a → b) → a → b

Just (f a) can be written as $ f Just $ f a

hlint will suggest to replace such code with $

```
data Tree a = Empty
            | Node (Tree a) a (Tree a)

instance Functor Tree where
    fmap :: (a → b) → (Tree a) → (Tree b)
    fmap f (Node l a r) = Node (fmap f l) (f a) (fmap f r)
    fmap f Empty = Empty
```

"There is only one rule in the Fight Club, i.e.
There is no rule in the Fight Club"
— Fight Club

"Recursion is the Big Daddy of looping. Anything that you can do with loop is can be done with recursion"
— Piyush

2-1-18

```
class Functor t where
  fmap :: (a -> b) -> t a -> t b

instance Functor Maybe where
  fmap f (Maybe a) = Maybe (f a)
  fmap f Nothing   = Nothing

instance Functor [] where
  fmap = map
```

[] is used as a parametrized type

[] a — [a]

```
instance Functor Tree where
```

Properties to follow

```
fmap id = id
fmap f fmap g = fmap (f . g)
```

IO a

type of i/o actions that result in a value of type a.

```
getLine :: IOString
```

~~read :: "120" :: Int~~ read :: Read a => String -> a

(read "123" :: Int)

123 :: Int

foo :: String → Int

~~foo <= read s + read~~

foo s = read s + 1

-- write the diffin definition ~~on the~~ of the getIntLine

getIntLine :: IO Int

IO is known to be an Functor. ∴ It has a fmap.

getIntLine = fmap read getLine

foo = fmap read (Read a, Functor f) =>

f String → f a.

data Person = Person string

dat type foo = Int -- type aliasing

data Person = Person String.

newtype Person = Person string

11.2 : ~~Hmm~~ Define a function getPerson :: IO Person.

make use of newtype Person.

getPerson = fmap Person getLine.

6.3 Applicative

data Person = Person String Int

get = fmap Person getline

get :: IO (Int → Person)

we can't ~~to do get~~ get IO Person (that person).

There is no function ~~to~~ to get 'a' from 'IO a'

IO a → a . There is no function of this type.

This is made by design.

class Functor t ⇒ Applicative t where

pure :: a → t a

(<*>) :: t (a → b) → t a → t b

(<$>) : (a → b) → a → b

```
get = (fmap Person getLine) <*> fmap getIntLine
```

Q) Read 3 integers from the user and add them up.

```
add3 :: Int -> Int -> Int -> Int

add3 x y z = x + y + z

fmap add3 :: IO (Int) -> IO (Int -> Int -> Int)

((fmap add3 getIntLine) <*> getIntLine) <*> getIntLine
```

```
f <$> tx = fmap f tx

f $ x = f x
```

∴ It can also be written as

```
add3 <$> getIntLine <*> getIntLine <*> getIntLine
```

```
f :: a -> b -> c -> d -> e

ta :: IO a
tb :: IO b
tc :: IO c
td :: IO d
```

f <*> ta <*> tb <*> tc <*> td :: If (e)

Law of Applicative

pure f <*> ta ≡ fmap f ta

pure f <*> ta ≡ fmap f ta

HW

data Person = Person String String Int

define getPerson for this definition of person.

getPerson = Person <$> getLine <*> getLine <*> getLine

Think!

instance Applicative Maybe where

;

8-3-24

class Functor t ⇒ Applicative t where

pure :: a → t a

(<*>) :: t (a→b) → ta → tb

instance Applicative Maybe where

-- pure :: a → Maybe a

pure = Just.

```
-- (<*>) :: Maybe (a->b) -> Maybe a -> Maybe b

(<*>) (Just f) (Just a) = Just $ f a

(<*>)    _       _      = Nothing
```

```
data Exp = C Int
         | PLUS Exp Exp
         | DIV Exp Exp

eval :: Exp -> Maybe (Int)

eval (C x) = Just x

eval (DIV e1 e2) =                    -- complete yourself

eval (PLUS e1 e2) = case eval e1 of
                      Just x -> case eval e2 of
                                  Just y -> Just x+y
                                  Nothing -> Nothing
                      Nothing -> Nothing.
```

put this is very large definition and put neat

eval ~~PLU PLUS~~.

f : a → b → c → d

-- f <$> ~~ta~~ ta = fmap f ta

f <$> ta = pure f <*> ma

ma : Maybe a

mb : Maybe b

mc : Maybe c.

eval PLUS = (+) <$> eval (e1) <*> eval (e2)

instance Applicative [] where

pure :: a → [ ]

pure x = []

(<*>) :: [a → b] → [a] → [b]

fs <*> xs = [f x | f ← fs, x ← xs]

Notation:

[x+y | x ← xs, y ← ys, x < 10, y < 100]

~~Step~~

Sort room Ju ... ... 12 ~~Maybe~~

adjective = ["cutting edge", "innovative"]

noun = ["product", "technology"]

join xs ys = unwords [xs, ys]

all = join <$> adjective <*> noun

11-3-24

class Functor t ⇒ Applicative t where

pure :: a → t a

(<*>) :: t(b → a) → ta → t b

f <$> [x1 ... xn] <*> [y1, ..., ym]

= f [x1 y1 ...]

[f1 ... fm] <*> [y1 ... yn]

[.... fi yj ....]

g :: a → b → c → d

g <$> [x1 ... xl] <*> [y1 ... ym] <*> [z1 ... zk]

the earlier definition of Applicative of a list does not computes all possible combinations.

Cauchy sequence

$x_0, \ldots$

$\forall \varepsilon \; \exists n \quad \forall m_1, m_2 > n \mid x_{m_1} - x_{m_2} | < \varepsilon$

```
newtype ZipList a = ZipList [a]

instance Functor ZipList where
    fmap :: (a -> b) -> ZipList a -> ZipList b

    fmap f (ZipList []) = ZipList []
    fmap f (ZipList (x:xs)) = ZipList (

    fmap f ZipList lst = ZipList (fmap f lst)

instance instance Applicative ZipList where
    pure :: a -> ZipList a.

    (<*>) :: ZipList (a->b) -> ZipList a ->
                                ZipList b
```

(<*>) (ZipList fs) (ZipList xs) = ZipList (zipWith ($) fs xs)

-- fs :: [a -> b]

-- xs :: [a]

-- to_write zipList3 or zipListn

-- f <$> ZipList [1, 2, 3] <*> ZipList [...] <*>

we need to satisfy the problem

pure f <*> xs = fmap f xs

pure x = ZipList xs

where xs = x : xs

13-3

- Functor : Applying pure function "inside" the data structure
  → fmap: (a->b) -> fa -> fb
- Applicative | Monoidal functor : Extending Functor interface
  (Applicative Functor)
  for "multi-argument" function.
  → (<*>) :: f (a->b) -> fa -> fb
  (apply)
- Monad: (>>=) Bind operator

(1) Read an int, read a string, ...

(2) Read an int, if n is read then read n things.

(<$>): map fp

(<*>): apply fp

(>>=): bind fp

```
(>>=) :: ta -> (a -> t b) -> t b
```

```
class Applicative t => Monad t where
  return :: a -> t a        -- return is same as pure
                            -- is a useless relic from the past
  (>>=) :: t a -> (a -> t b) -> (t b)
  (>>) :: ta -> tb -> tb    -- (>>) ta tb = (>>=) ta (\_ -> tb)
(<*>) tf ta = tf >>= (\f -> (ta >>= (\a -> return (f a))))
```

Syntactic Sugar

The do-notation

```
                    ta
stmt = do stmt | do action; stmt      (1)
       | do x <- action; stmt         (2)
       | action.
```

(1) action >> stmt

(2) action >>= \x -> stmt

```
(<*>)  tf  ta  =  do  f <- tf
                     x <- ta
                     return (f x)
```

15-3-24

(<$>) :: (a → b) → ta → t b (map) Functor)

(<*>) :: t (a → b) → ta → t b (apply) Applicative

(>>=) :: ta → (a → t b) → t b (bind) Monad.

[] [x] is an ∞ dimensional vector space

```
ta1 >>= \x1 -> (ta2 >>= \x2 -> ta .... )

do  x1 <- ta1
    x2 <- ta2
    .
    .
    .
    xm <- tam
    ye
```

```
class Show a where
    show :: a -> String
```

```
class Read a where
    read :: String -> a
```

```
print  :: Show a => a -> IO ()
putStr :: String -> IO ()
print x = putStr (show x
print   = putStr . show
```

```
getLine :: IO String

get :: Read a => IO a

get = fmap read getLine

get = read <$> getLine
```

~~print~~

```
do putStr "Give an integer".
   inp <- get
   print (inp+1)
```

~~put str "Give an integer"~~

~~get >>= \inp -> (print (inp+1))~~

Un sugar the above statement

action >>= \x -> (stmt)

```
putStr "Give an integer" >> get >>= \inp -> (print (inp+1))
```

Q) It will ~~print~~ prompt the user for an integer and should print its square and the program should be infinite.

```
Square :: IO ()
Square = do putStr "Give an integer"
            inp <- get
            print (inp * inp)
            square.
```

inf_Square = putStr "Enter n" >> (getLine >>= \n -> (n*n >> inf_Square))

```
foo = putStrLn "Hello"
main = putStrLn "World"
```

In Haskell, the program has one main whose type is IO (). Entire computation is forced by eval of it.

```
foo = putStrLn "Hello"
main = putStrLn "World"
```

```
main = putStrLn "World"
foo = putStrLn "Hello"
```

```
main = do x <- getLine
          y <- getLine
          putStr (x ++ y)
```

// Takes two strings and concatenate them.

```
foo = getLine

main = do x <- getLine
          y <- foo
          putStr (x ++ y)
```

// It is also the same as the first one.

```
foo = getLine

main = do x <- foo
          y <- foo
          putStr (x ++ y)
```

// It is also the same as the first one

as foo is a plan to generate IO (String) and the plan is used twice in the plan of main.

```
foo = getLine

main = do x <- getLine
          y <- getLine
          putStr (x ++ y)
```

// Takes two strings and concatenate them

The first line is ignored as it is never executed

getLine is a plan to generate a string using some side effectful computation.

```
foo = do return "Hello"
         return "World"
```

// The output will be World in the terminal.

①
```
main : do x ← foo
           putStrLn x
```

return "Hello" >> return "World"

return "Hello" >>= _ -> return "World"

~~main = do x ← getLine~~
~~y ← (++) <$> getLine <*> getLine~~

②
```
main : do x ← (++) <$> getLine <*> getLine
          return x.
```

- ~~Eve~~ everything we can do with applicative can be done with a monad. Because monad happens to be a applicative.
- Always try to ~~go~~ do things generally whenever possible, the second definition of is more general that it can used for both applicative and monad.

Take a number and check whether it is prime.

~~instance~~

instance Monad Maybe where

(>>=) :: Maybe → (a → Maybe b) → Maybe b

(>>=) Nothing _ = Nothing

(>>=) (Just a) f = ~~Just (f a)~~ f a

20-3-24

do
$x_1 \leftarrow foo_1$
$x_2 \leftarrow foo_2$
⋮
$x_m \leftarrow foo_m$
something

Nothing >>= f = Nothing

Just x >>= f = f x

return = Just

~~{-- here bracket is not~~

-- here bracket is not required around Just x

as `>>=` is an operator and Just is a function.

```
data Exp v = Var v
           | Plus (Exp v) (Exp v)
           | Mul (Exp v) (Exp v)
           | Const Int

instance Functor Exp where
  -- fmap :: (v1 -> v2) -> Exp v1 -> Exp v2
  fmap uT (Var x)      = Var (uT x)
  fmap uT (Plus e1 e2) = Plus (fmap uT e1) (fmap uT e2)
  fmap uT (Mul e1 e2)  = Mul (fmap uT e1) (fmap uT e2)
```

> Guiding point of abstraction is selective amnesia
>
> — Pi yusha

```
do x <- e1
   e2

e1 >>= (\x -> e2)
```

```
join :: Monad m => m (m a) -> m a

join mma = do ma <- mma
              ma
         | mma >>= \x -> x
```

Monad can be defined in terms of either ~~monad~~ interms of `join` or `bind`.

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b

(>>=) m f = join (fmap f ma)
```

```
join :: Exp (Exp u) -> Exp u

join (Var v) = v
join (Plus e1 e2) = Plus (join e1) (join e2)
join (Mul e1 e2) = Mul (join e1) (join e2)
```

~~join (Var~~

```
join x = x
join (Const x) = Const x
```

3-24

Syntactic Monad

```
data Exp v = Var v
           | Const Int
           | Plus (Exp v) (Exp v)
```

what is the meaning of >>= in exp (exp substitution) ?

List Monad

```
do x ← alist          |  alist >>= foo
   foo x              |
```

alist :: [a], foo :: a → [b]

~~alist >>= foo = concat $~~

alist >>= foo = concat $ map foo alist

Distr a : generate a ~~value from~~ value from

Gen a : random generate an element of type a.

The idea of list monad is sampling.

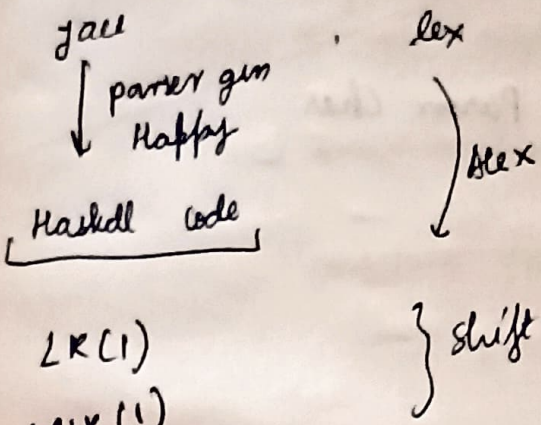
```
gI :: Gen Int

gen2I :: Gen (Int, Int)
gen2I = do x ← gI
           y ← gI
           return (x, y)
```

- Read the Gen monad

27-3-24

Parsing Monad

Build a Library for constructing Parser.



LL - Parsing Recursive descent parsing

(-- Ocaml - menhir LR)

newtype Parser a = Parser (String → a)

Parser Combinator libraries

Parsec, megaParsec, altoParsec

Parser should handle these

(1) "Rest of the input"

(2) "Parse Error".

```
String → Maybe(a, String)

newtype Parser a = Parser (String → Maybe (a, String))

satisfy : (Char → Bool) → Parser Char

digit = satisfy isDigit

char : Char → Parser Char

char x = satisfy (\c → c == x)

char x = satisfy ((==) x)

satisfy pr = Parser fn
    where
      fn :: String → Maybe (char, String)
      fn (x:xs) = if pr x then Just (x, xs)
                  else Nothing
      fn (nil) = Nothing

(<|>) : Parser a → Parser a → Parser a
```

p1 <|> p2 =

type Result a : Maybe (a, String)

newtype Parser a = Parser (String → Result a)

runParser : Parser a → String → Result a

runParser (Parser fn) = fn

p1 <|> p2 = Parser fn

where fn input = case runParser p1 input of

Nothing → runParser p2 input

x → x

many : Parser a → Parser (a)

many p = Parser fn

where fn input =

newtype Parser a = Parser (String → Result a)

// ~~this case~~

if the input doesnt match 'a' then return empty else they matching the remaining

this can be done differently.

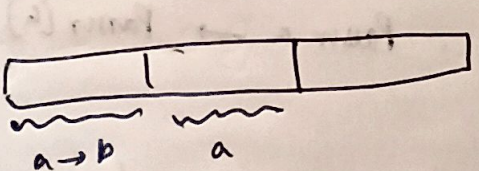
```
instance Functor Parser where

fmap :: (a -> b) -> Parser a -> Parser b

fmap f pa = Parser fn

    where fn :: String -> Result b

          fn input = runParser

          fn input = fmap f (runParser pa input)
```

```
data Result a = Ok a String
              | Err

Instance Functor Result where
  fmap f (Ok a s) = Ok (fa) s
  fmap f Err      = Err
```

HW

Define the applicative instance of

```
<*> Parser (a -> b) -> Parser a
                    -> Parser b
```



Tryout megaParsec or Parsec library to write small parsers

Parser as Monads

```
data Result a = Ok a String
              | Err

newtype Parser a = Parser (String -> Result a)

runParser :: Parser a -> (String -> Result a)

runParser (Parser fn) = fn

instance Functor Result where
  -- fmap :: (a -> b) -> Result a -> Result b
  fmap f (Ok x str) = Ok (f x) str
  fmap f Err        = Err

instance Functor Parser
-- fmap :: (a -> b) -> Parser a -> Parser b
  fmap f pa = Parser (pfn :: String -> Result b)
    where pfn str = fmap f (runParser pa str)
```

```
instance Applicative Parser where
  -- pure :: a -> Parser a
  pure a = Parser (pfn :: String -> Result a)
    where pfn str = Ok a str

(<*>) :: Parser (a -> b) -> Parser (a) -> Parser (b)

pf <*> pa = Parser ( pb :: String -> Result b)
  where
    pb str = case runParser pf str of
               Ok f rest ->
               Err -> Err

pf <*> pa = Parser (pb :: String -> Result b)
where
  pb str = case runParser pf str of
             Ok f rest -> case runParser pa rest of
             Bad -> Err    Ok a restp -> Ok (f a) restp
                           Err -> Err
             Err -> Err
```

```
pf <*> pa = Parser (pb :: String -> Result b)
where
  pb str = case runParser pf str of
             Ok f rest -> fmap f (runParser pa rest)
             Err -> Err

digit = satisfy isDigit
alpha = satisfy isAlpha

many :: Parser a -> Parser [a]
many1 :: Parser a -> Parser [a]

data Person String Int
  name = many1 alpha
  age = read <#> (many1 digit)

Person <$> name <*> age
```

Ans

Define many interms of Applicative interface.

```
many1 p = (:) <$> p <*> many p

many p = many1 p <|> pure []
```