FIT2102 Programming Paradigms Tutorial 8

What is the point (free)?

```
mapM f xs = sequence (fmap f xs)
```

```
mapM = curry (sequence . uncurry fmap)
```

```
mapM = ((.).(.)) sequence fmap
```

```
mapM = fmap fmap fmap sequence fmap
```

Faculty of Information Technology

Point Free Code

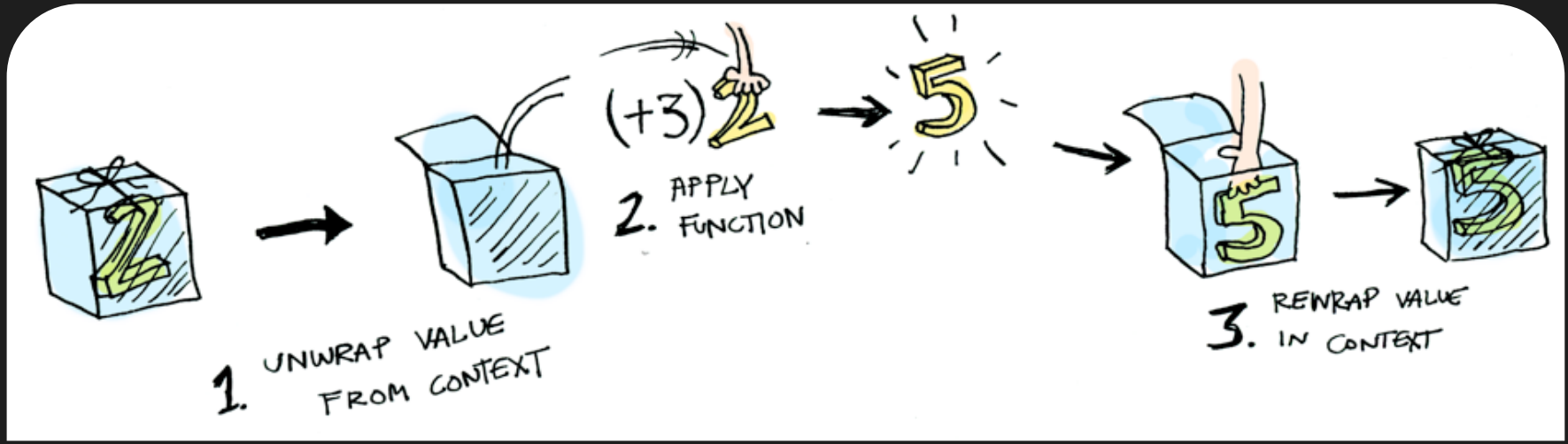
```
f a b c = (a+b)*c
f a b c = (*) (a+b) c -- operator sectioning
f a b   = (*) (a+b) -- eta reduce
f a b   = (*) ((a+) b) -- adding some brackets, now it is in the form f (g x)
f a b   = ((*) . (a+)) b -- compose, f (g x) === (f . g) x
f a     = (*) . (a+) -- eta reduce
f a     = (*) . ((+) a) -- operator sectioning
f a     = ((*) .) ((+) a) -- adding some brackets, now it is in the form f (g x)
f a     = (((*) .) . (+)) a -- compose, f (g x) === (f . g) x
f       = ((*) .) . (+) -- eta reduce
```

Real Functional Programming

Functors and Applicatives, two of the main building blocks of functional programming theory.

Functor and Applicative are typeclasses like we saw last week. That is they are properties you apply on types. Types, by themselves, cannot enforce certain properties it is therefore the programmer’s task to implement them.

What is a Functor?



Warning: We will use the box analogy quite often. However, it is not just a box! It's a computational context. It can be many things, not just a box!

Functor

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

instance Functor Maybe where
 fmap f (Just x) = Just (f x)
 fmap _ Nothing = Nothing
```

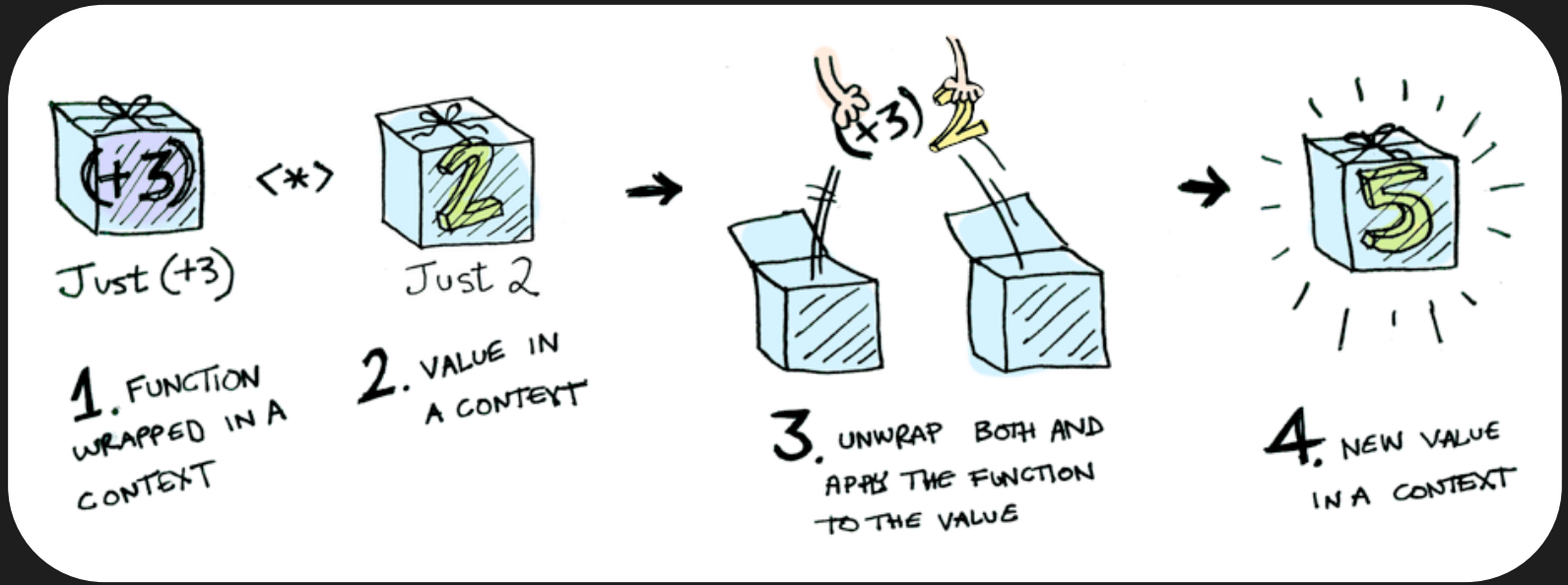
```
ghci> fmap (+3) (Just 2)
Just 5
(+3) <$> (Just 2)
```

Functors You have already seen!

```
instance Functor [] -- Defined in `GHC.Base'
instance Functor Maybe -- Defined in `GHC.Base'
instance Functor IO -- Defined in `GHC.Base'
instance Functor ((,) a) -- Defined in `GHC.Base'
instance Functor ((->) r)
```

Applicative!

Basically, beefed up Functors!



Applicative!

Basically, beefed up Functors!

```
class Functor f => Applicative (f :: * -> *) where
   pure :: a -> f a
   (<*>) :: f (a -> b) -> f a -> f b
```

Pure?

```
pure :: a -> f a
```

Pure gets a value and puts the value inside of a context!

Example:

```
pure :: a -> Maybe a
pure 3
> Just 3
```

Applicative Examples

```
instance Applicative [] -- Defined in `GHC.Base'
instance Applicative Maybe -- Defined in `GHC.Base'
instance Applicative IO -- Defined in `GHC.Base'
```

Parsing

```
Parsing a single character from a string!
The function returns
      - Nothing, if the parsing fails
      - Just (r, p), where r is the unparsed portion of the input, and p is the
parsed input

-- >>> parseChar "abc"
-- Just ("bc",'a')
-- >>> parseChar ""
-- Nothing
parseChar :: String -> Maybe (String, Char)
parseChar "" = Nothing
parseChar (c:rest) = Just (rest, c)
```

Parser Type

Wrap the previous function in a type! Now we can make it part of functor and applicative…..

```
newtype Parser a = Parser (String -> Maybe (String, a))
```