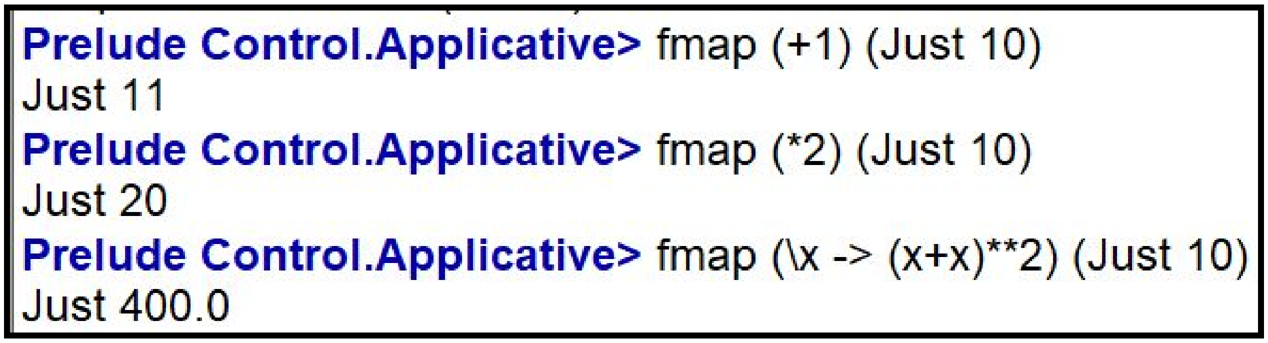
Fmap:

- **fmap :: Functor f => (a -> b) -> f a -> f b**
- This means that fmap takes a function and a functor and applies the function over the functor.
- E.g.

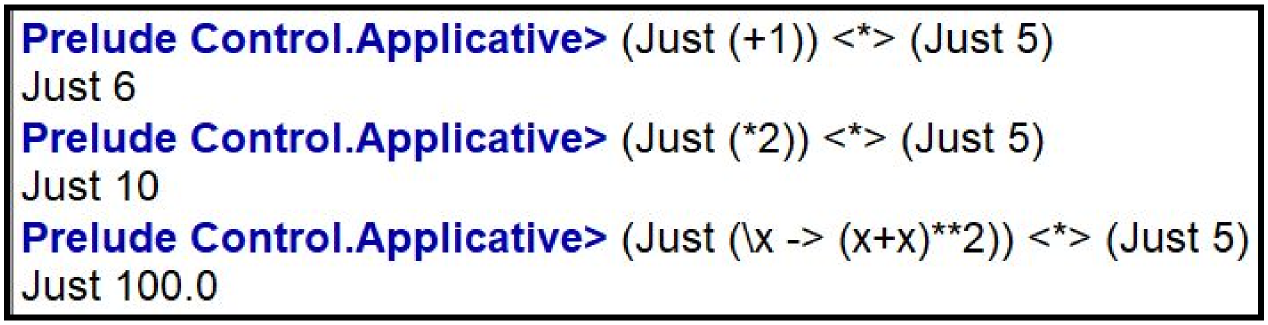


```
Prelude Control.Applicative> fmap (+1) (Just 10)
Just 11
Prelude Control.Applicative> fmap (*2) (Just 10)
Just 20
Prelude Control.Applicative> fmap (\x -> (x+x)**2) (Just 10)
Just 400.0
```

- Can be thought of as liftA1. This will be explained below.

<*>:

- Also called ap.
- **(<*>) :: f (a -> b) -> f a -> f b**
- <*> takes a functor with a function in it and another functor and applies the function to the second functor.
- E.g.

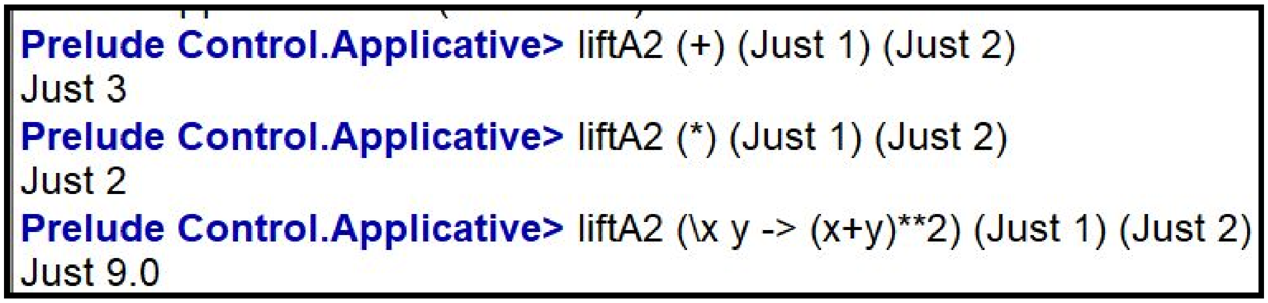


```
Prelude Control.Applicative> (Just (+1)) <*> (Just 5)
Just 6
Prelude Control.Applicative> (Just (*2)) <*> (Just 5)
Just 10
Prelude Control.Applicative> (Just (\x -> (x+x)**2)) <*> (Just 5)
Just 100.0
```

- **Note:** You need to do **import Control.Applicative** to use ap.

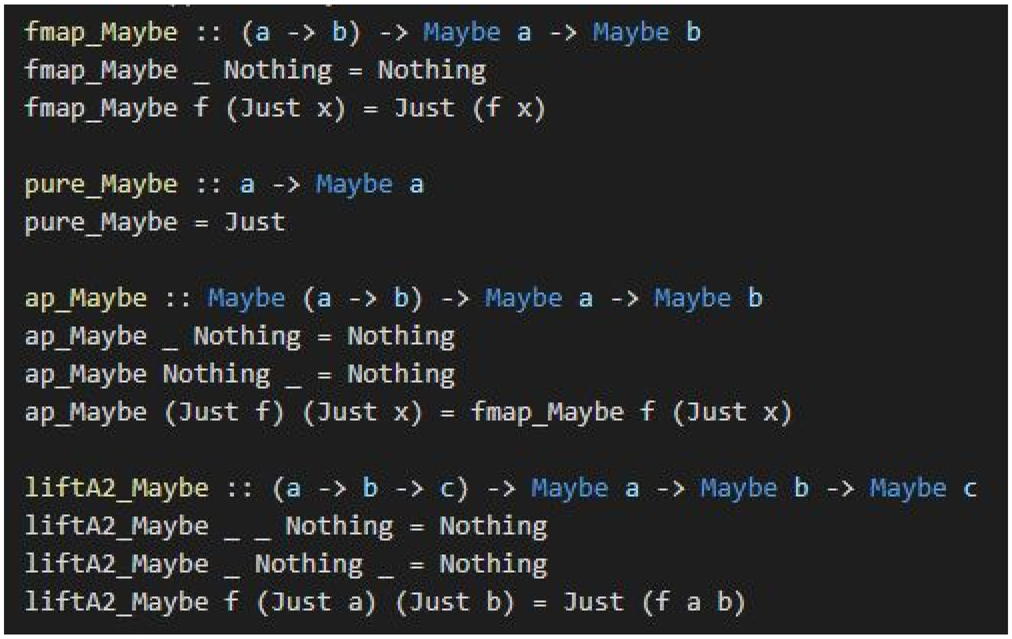
LiftA2:

- **liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c**
- If you compare the above line with fmap, you'll see that they're very similar, but fmap takes 1 functor while applicative takes 2. Furthermore, the function fmap uses only takes in 1 argument, while the function liftA2 uses takes 2 arguments. This is why we can think of fmap as liftA1.
- E.g.



```
Prelude Control.Applicative> liftA2 (+) (Just 1) (Just 2)
Just 3
Prelude Control.Applicative> liftA2 (*) (Just 1) (Just 2)
Just 2
Prelude Control.Applicative> liftA2 (\x y -> (x+y)**2) (Just 1) (Just 2)
Just 9.0
```

- We can use fmap and <*> to implement liftA2.
  Here's an implementation of fmap, pure, <*> and liftA2 for the functor Maybe.



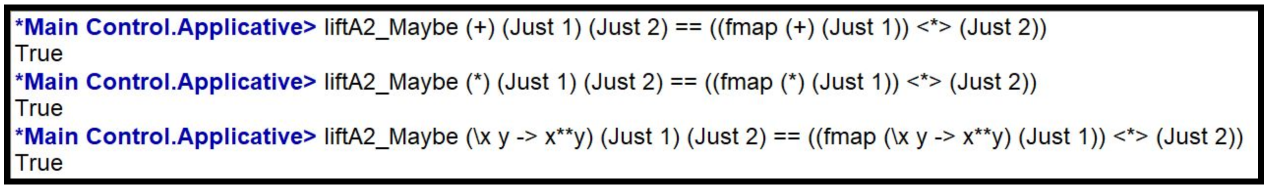
```haskell
fmap_Maybe :: (a -> b) -> Maybe a -> Maybe b
fmap_Maybe _ Nothing = Nothing
fmap_Maybe f (Just x) = Just (f x)

pure_Maybe :: a -> Maybe a
pure_Maybe = Just

ap_Maybe :: Maybe (a -> b) -> Maybe a -> Maybe b
ap_Maybe _ Nothing = Nothing
ap_Maybe Nothing _ = Nothing
ap_Maybe (Just f) (Just x) = fmap_Maybe f (Just x)

liftA2_Maybe :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
liftA2_Maybe _ _ Nothing = Nothing
liftA2_Maybe _ Nothing _ = Nothing
liftA2_Maybe f (Just a) (Just b) = Just (f a b)
```

  Note that for liftA2, we're not using fmap and <*> to implement it.
  Here's how we can use fmap and <*> to implement liftA2.
  liftA2 f xs ys = (fmap f xs) <*> ys
  E.g.

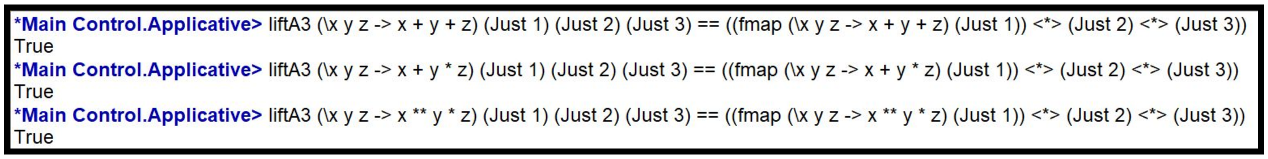


```
*Main Control.Applicative> liftA2_Maybe (+) (Just 1) (Just 2) == ((fmap (+) (Just 1)) <*> (Just 2))
True
*Main Control.Applicative> liftA2_Maybe (*) (Just 1) (Just 2) == ((fmap (*) (Just 1)) <*> (Just 2))
True
*Main Control.Applicative> liftA2_Maybe (\x y -> x**y) (Just 1) (Just 2) == ((fmap (\x y -> x**y) (Just 1)) <*> (Just 2))
True
```

  The reason why **liftA2 f xs ys = (fmap f xs) <*> ys** is because fmap applies the function, f, on xs, so xs is a functor with a function in it, and then <*> applies that function onto ys.

<u>LiftA3:</u>

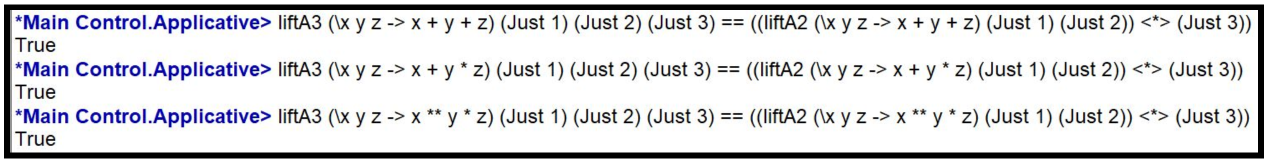
- **liftA3 :: Applicative f => (a -> b -> c -> d) -> f a -> f b -> f c -> f d**
- This is similar to liftA2, but it takes 3 arguments instead of 2.
- We can implement liftA3 using fmap and <*>.
  liftA3 f xs ys zs = (fmap f xs) <*> ys <*> zs
  E.g.



```
*Main Control.Applicative> liftA3 (\x y z -> x + y + z) (Just 1) (Just 2) (Just 3) == ((fmap (\x y z -> x + y + z) (Just 1)) <*> (Just 2) <*> (Just 3))
True
*Main Control.Applicative> liftA3 (\x y z -> x + y * z) (Just 1) (Just 2) (Just 3) == ((fmap (\x y z -> x + y * z) (Just 1)) <*> (Just 2) <*> (Just 3))
True
*Main Control.Applicative> liftA3 (\x y z -> x ** y * z) (Just 1) (Just 2) (Just 3) == ((fmap (\x y z -> x ** y * z) (Just 1)) <*> (Just 2) <*> (Just 3))
True
```

- We can implement liftA3 using liftA2 and <*>.
  liftA3 f xs ys zs = (liftA2 f xs ys) <*> zs

E.g.



```
*Main Control.Applicative> liftA3 (\x y z -> x + y + z) (Just 1) (Just 2) (Just 3) == ((liftA2 (\x y z -> x + y + z) (Just 1) (Just 2)) <*> (Just 3))
True
*Main Control.Applicative> liftA3 (\x y z -> x + y * z) (Just 1) (Just 2) (Just 3) == ((liftA2 (\x y z -> x + y * z) (Just 1) (Just 2)) <*> (Just 3))
True
*Main Control.Applicative> liftA3 (\x y z -> x ** y * z) (Just 1) (Just 2) (Just 3) == ((liftA2 (\x y z -> x ** y * z) (Just 1) (Just 2)) <*> (Just 3))
True
```

This is because **(fmap f xs) <*> ys** is equivalent to **liftA2 f xs ys**.

<u>In general:</u>

- If you have liftAn, where n > 1, you can implement in the following way:
    1. liftAn f a b c … z = (fmap f a) <*> b <*> c … <*> z
    2. liftAn f a b c … y z = (liftA(n-1) f a b c … y) <*> z