ECS713: Functional Programming week 9: Monadic Programming

Week 9

Monadic Programming

Monadic Programming: Functor, Applicative, Monad

- understand the role played by these three fundamental type classes: Functor, Applicative and Monad

- make a given data type an instance of these classes in a reasonable way

Monadic Programming: Monads

- understand the the different monad operations
- give examples of monads

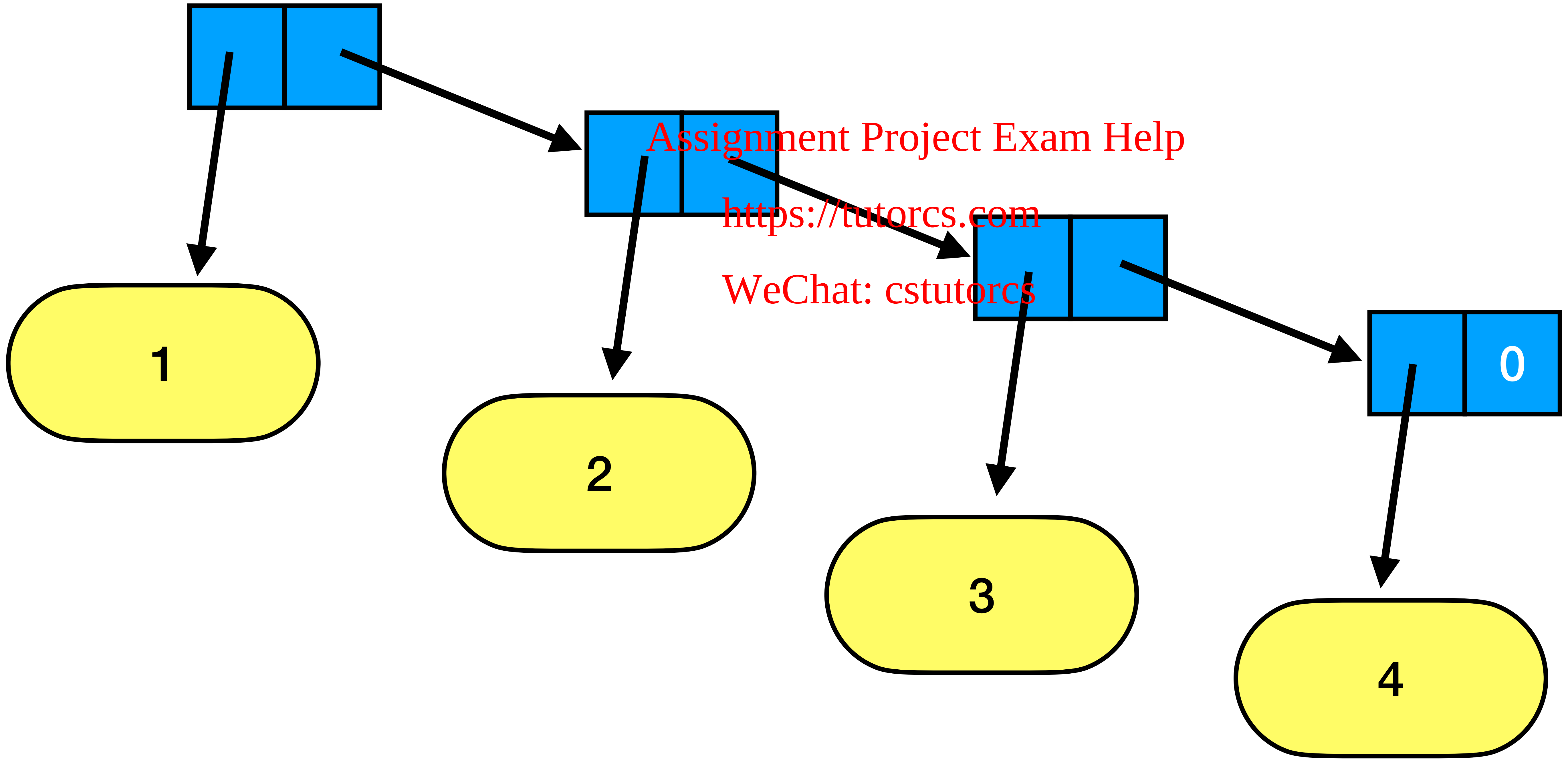
Functor

```
Prelude> :t map
map :: (a -> b) -> [a] -> [b]
```

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

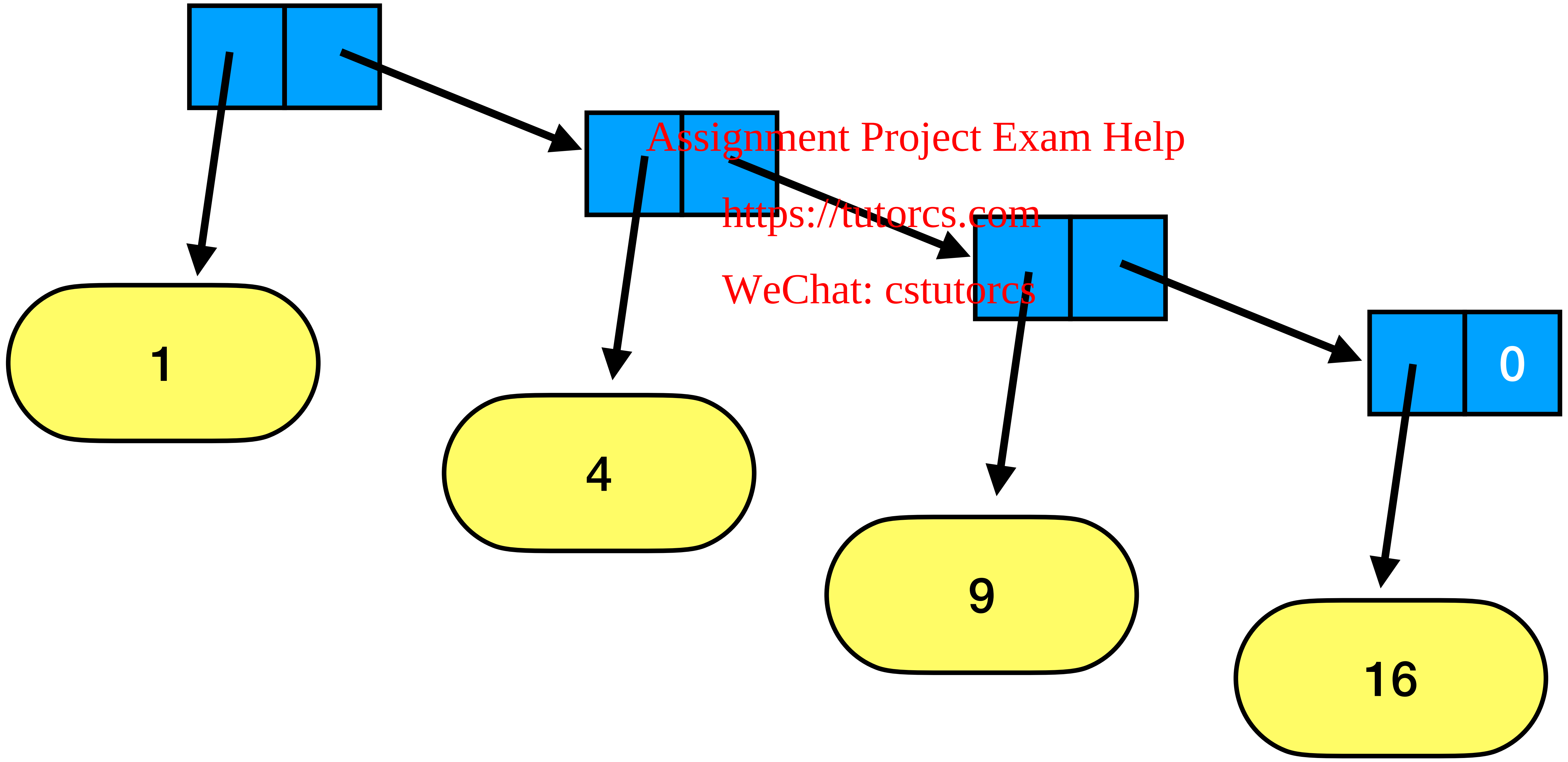

- map converts a function (a -> b) into a function ([a] -> [b])
- fmap converts a function (a -> b) into a function (f a -> f b), where f is a suitable type constructor
- fmap is like map, but for more general types

```
Prelude> :t map
map :: (a -> b) -> [a] -> [b]
```



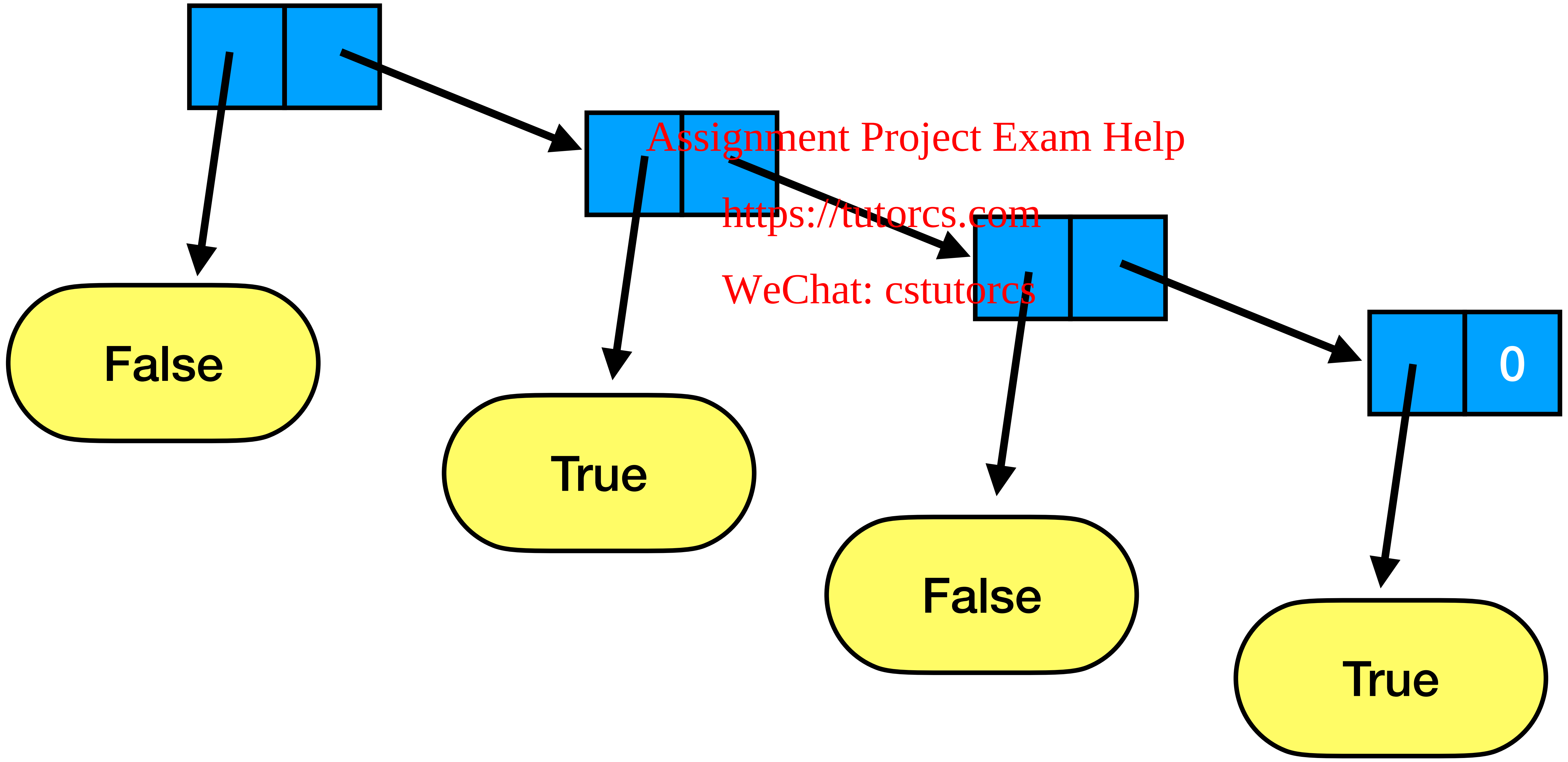
[1,2,3,4]

```
Prelude> :t map
map :: (a -> b) -> [a] -> [b]
```



map (^2) [1,2,3,4]

```
Prelude> :t map
map :: (a -> b) -> [a] -> [b]
```



map even [1,2,3,4]

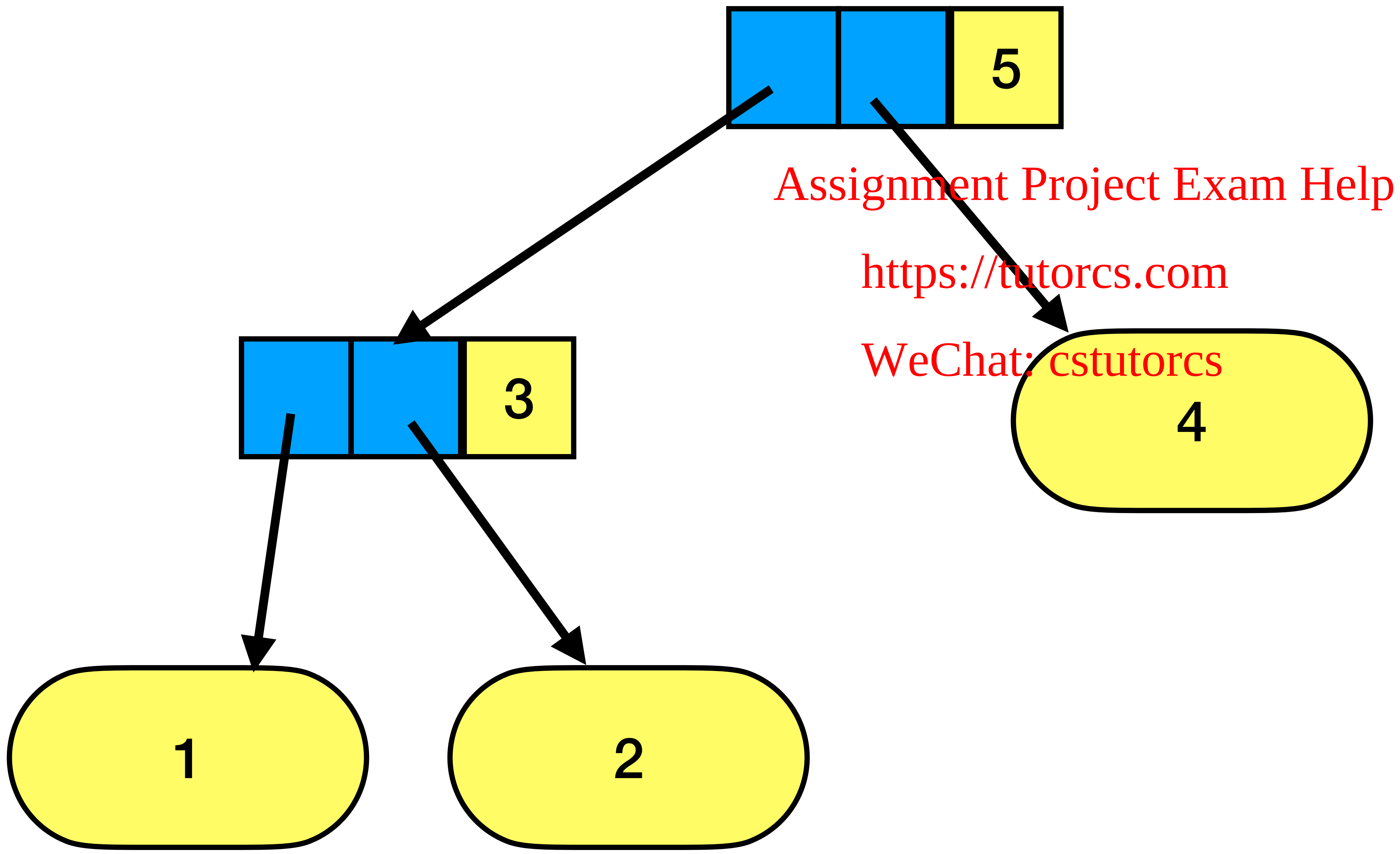
```haskell
class Functor f where
 fmap :: (a -> b) -> f a -> f b
```


- Other types also give a shaped container for a collection of values.
- For these types: (fmap g) :: fa -> fb keeps the shape and applies g to each of the values.

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

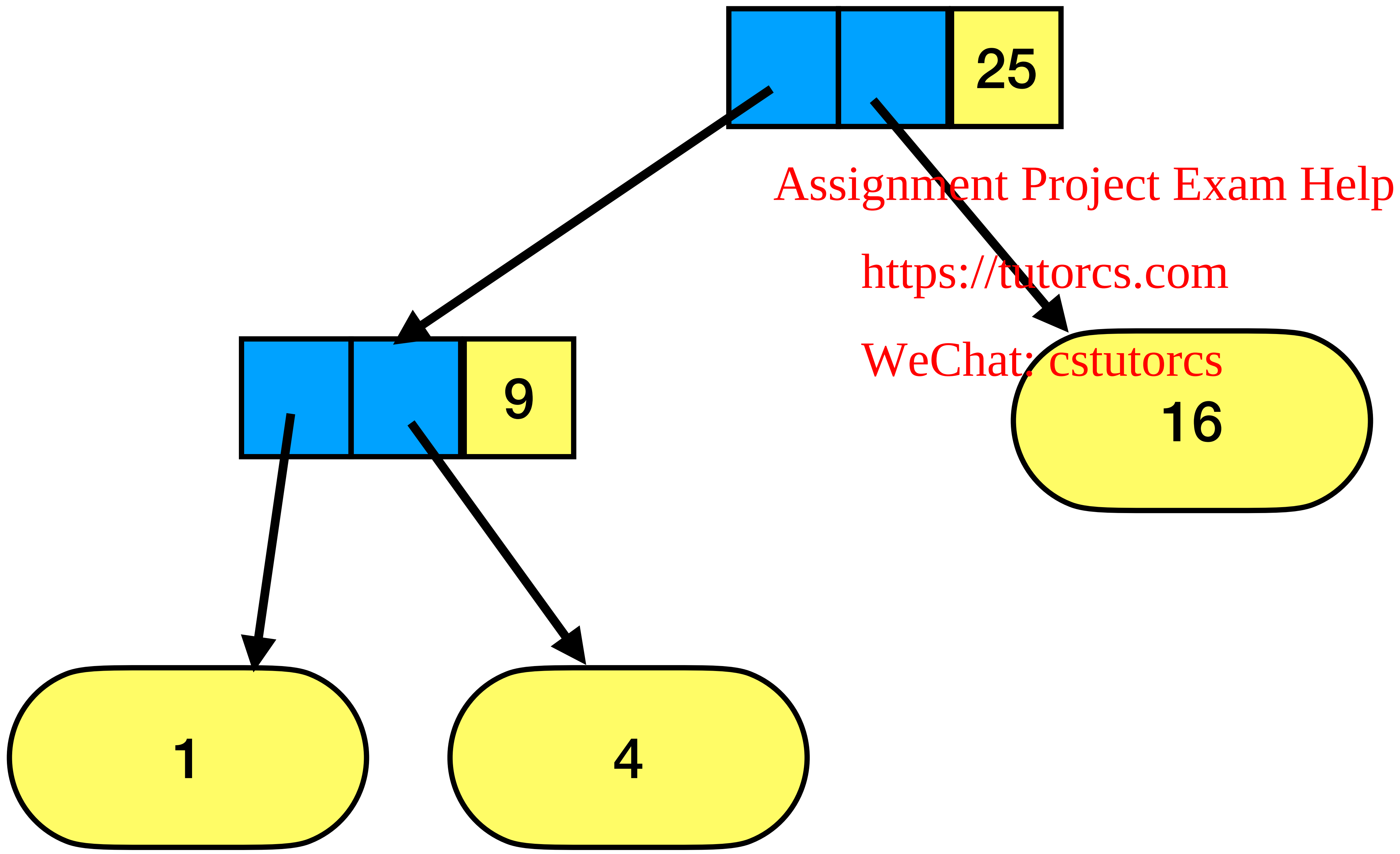
```
data Btree a = Empty | Leaf a | Node (Btree a) (Btree a) a

fmap g Empty = Empty
fmap g (Leaf a) = Leaf (g a)
fmap g (Node L R a) = Node (fmap g L) (fmap g R) (g a)
```

```
data Btree a = Empty | Leaf a | Node (Btree a) (Btree a) a

fmap g Empty = Empty
fmap g (Leaf a) = Leaf (g a)
fmap g (Node L R a) = Node (fmap g L) (fmap g R) (g a)
```



Node (Node (Leaf 1) (Leaf 2) 3) (Leaf 4) 5

```
data Btree a = Empty | Leaf a | Node (Btree a) (Btree a) a

fmap g Empty = Empty
fmap g (Leaf a) = Leaf (g a)
fmap g (Node L R a) = Node (fmap g L) (fmap g R) (g a)
```



fmap (^2) (Node (Node (Leaf 1) (Leaf 2) 3) (Leaf 4) 5)

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

```haskell
data Btree a = Empty | Leaf a | Node (Btree a) (Btree a) a

fmap g Empty = Empty
fmap g (Leaf a) = Leaf (g a)
fmap g (Node L R a) = Node (fmap g L) (fmap g R) (g a)
```

Other examples

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


Maybe a


```
data Maybe a = Nothing | Just a

fmap g Nothing = Nothing
fmap g (Just a) = Just (g a)
```

Other examples

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


Either a b


```
data Either a b = Left a | Right b

fmap g (Left a) = Left a
fmap g (Right b) = Right (g b)
```

Other examples

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


Either a b


```
data Either a b = Left a | Right b

fmap g (Left a) = Left (g a)
fmap g (Right b) = Right b
```

Other examples

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

(a,b)


```
fmap g (a,b) = (a, g b)
```

Other examples

- Records (whose fields are not functions)
- Datatypes that do not involve functions

Different examples: functions

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


f = x -> ()


```
fmap :: (a -> b) -> (x -> a) -> (x -> b)
fmap g h = g . h
```

A non-example

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


f = () -> y


```
fmap :: (a -> b) -> (a -> y) -> (b -> y)
fmap g h = ???

fmapr :: (a -> b) -> (b -> y) -> (a -> y)
fmapr g h = h . g
```

A hard example: continuations

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


f = (() -> x) -> x


```
fmap :: (a -> b) -> ((a -> x) -> x) -> ((b -> x) -> x)
fmap g h = \k -> h $ k . g
```

mapping pipelines

$$a \xrightarrow{g} b \xrightarrow{h} c \xrightarrow{k} d$$

$$f\ a \xrightarrow{\text{fmap } g} f\ b \xrightarrow{\text{fmap } h} f\ c \xrightarrow{\text{fmap } k} f\ d$$

We should not care about the build order

fmap (h.g) = (fmap h).(fmap g)

Summary

- A **functor** is a type constructor that is mappable.
- We can apply the functor to a whole pipeline of functions.

Applicative

- Functors are fine for unary functions
- Suppose we have a binary function:


- ```
  g :: a -> b -> c
  ```

- Can we get a binary map:

- ```
  bmap g :: f a -> f b -> f c
  ```

Applicative

- Functors are not strong enough to give us this binary map
- We need something called an **applicative**

Applicative: v0

- Suppose f is a functor
- If we have
```
g :: a -> b -> c
```
- Then
```
uncurry g :: (a,b) -> c
```

- So
```
fmap (uncurry g) :: f (a,b) -> f c
```
- We need a
```
p :: f a -> f b -> f (a,b)
```
- Then
```
fmap (uncurry g) . p :: f a -> f b -> fc
```

Applicative: v0 (lists)

- Suppose f is a functor
- If we have `g :: a -> b -> c`
- Then `uncurry g :: (a,b) -> c`

- So `fmap (uncurry g) :: f (a,b) -> f c`
- We need a `p :: [a] -> [b] -> [(a,b)]`
- Then `\as -> fmap (uncurry g) . p as :: f a -> f b -> fc`

Applicative: v0 (lists)

```
p :: [a] -> [b] -> [(a,b)]
-- p as bs = concat $ fmap (\a -> fmap (\b -> (a,b)) bs) as
p as bs = [(a,b) | a <- as, b <- bs]
```



```
*Main> p [1..4] [1,2]
[(1,1),(1,2),(2,1),(2,2),(3,1),(3,2),(4,1),(4,2)]
*Main>
```

Applicative: v0 (lists)

```
p :: [a] -> [b] -> [(a,b)]
p as bs = concat $ fmap (\a -> fmap (\b -> (a,b)) bs) as
p as bs = [(a,b) | a<-as, b <- bs]
q as bs = concat $ fmap (\b -> fmap (\a -> (a,b)) as) bs
q as bs = [(a,b) | b <- bs, a <- as]
```



```
*Main> p [1..4] [1,2]
[(1,1),(1,2),(2,1),(2,2),(3,1),(3,2),(4,1),(4,2)]
*Main> q [1..4] [1,2]
[(1,1),(2,1),(3,1),(4,1),(1,2),(2,2),(3,2),(4,2)]
```

Applicative: v0 (lists)

```
p :: [a] -> [b] -> [(a,b)]
p as bs = concat $ fmap (\a -> fmap (\b -> (a,b)) bs) as

bmap g = \as -> fmap (uncurry g) . p as
```



```
*Main> p [1..4] [1,2]
[(1,1),(1,2),(2,1),(2,2),(3,1),(3,2),(4,1),(4,2)]
*Main> bmap (-) [1..4] [1,2]
[0,-1,1,0,2,1,3,2]
```

Applicative: v1

- Suppose f is a functor
- If we have

```
g :: a -> b -> c
```

- Then

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


- We need a

```
(<*>) :: f (b -> c) -> f b -> f c
```

- Then

```
\as bs -> fmap g as <*> bs :: f a -> f b -> fc
```

Applicative: v1 (lists)

```
fs <*> xs = [f x | f <- fs, x <- xs]

bmap1 g = \as bs -> fmap g as <*> bs
```



```
*Main> bmap (-) [1..4] [1,2]
[0,-1,1,0,2,1,3,2]
*Main> bmap1 (-) [1..4] [1,2]
[0,-1,1,0,2,1,3,2]
```

Applicative -> Functor

- We want every Applicative to be a Functor.
- To make that happen we need another function:

  - pure :: a -> f a
- With that:
  - fmap g xs = pure g <*> xs

Two versions of Applicative

- Type classes have certain required functions
- Sometimes some of the required functions can be defined in terms of others.

- For example:
  - in Eq, we have == and /=, and either can be defined in terms of the other.
  - Eq has two minimal presentations, we can give either == or /= and the other iwll be inferred.
- Applicative also has two minimal presentations

The usual Applicative

```
class Functor f => Applicative f where
    {-# MINIMAL pure, ((<*>) | liftA2) #-}
    -- | Lift a value.
    pure :: a -> f a
    -- | Sequential application.
    (<*>) :: f (a -> b) -> f a -> f b
```

The less usual Applicative

```
class Functor f => Applicative f where
    {-# MINIMAL pure, ((<*>) | liftA2) #-}
    -- | Lift a value.
    pure :: a -> f a
    -- | Lift a binary function to actions.
    liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

The relation between them

```
class Functor f => Applicative f where
    {-# MINIMAL pure, ((<*>) | liftA2) #-}
    -- | Lift a value.
    pure :: a -> f a
    -- | Sequential application.
    (<*>) :: f (a -> b) -> f a -> f b
    -- | Lift a binary function to actions.
    liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```



```
(<*>) fs as = liftA2 id fs as

liftA2 f x = (<*>) (fmap f x)
```

Monads I

Aims

- The use of monads to handle effects is one of the most distinctive and challenging features of Haskell.

- But it also has lessons for other programming paradigms.
- In this presentation we set out the problem.

Learning Objectives

- learn why Monads are a key construct for Haskell
- learn what Monads are and what they do

Pure Functional Programming

- The basic idea of functional programming is that we construct pieces of data by applying functions to existing data.

- We don’t change data, we produce new data instead.
- The Haskell type systems tells us what we need to know about plug compatibility: it tells us what functions we can apply to which data.

```
*Main Lib> :t words
words :: String -> [String]
```


- The function words takes a String as input and produces a list of Strings.
- It is a pure function, it has no side effects.

$$\textbf{String} \xrightarrow{\textbf{words}} \textbf{[String]} \xrightarrow{\textbf{length}} \textbf{Int}$$

Effects

- Some functions are not pure, they have effects.
- Examples:
  - they carry out **IO**
  - they can produce an **exception**
  - they produce something alongside the return value
  - they are **non-deterministic**
  - they make use of a **State**

Haskell and Effects

- Haskell documents this through its type system:
- Examples:
  - they carry out IO: use the IO type constructor: **IO a**
  - they can produce an exception: use Either to allow the return of the exception as a value: **Either Exception a**
  - they produce something alongside the return value: return a tuple containing the something alongside the return value: **(Something, a)**
  - they are non-deterministic: return the (lazy) list of possible return values: **[a]**
  - they make use of a State: pass the state explicitly as input (if just reading the State) and as both input and output (if changing it): **State -> (State,a)**

Example

- Let's suppose we want to take the logarithm of the head of a list of numbers.

```
Prelude> :t log $ head xs
log $ head xs :: (Floating a, Enum a) => a
Prelude>
```


- We can just pipe the list through the functions head and log

- But:

```
Prelude> head []
*** Exception: Prelude.head: empty list
Prelude> log (-2)
NaN
```

Use Either to allow return of an Exception

```
safeHead [] = Left “exception: head []”
safeHead (x:xs) = Right x

safeLog x | x<=0 = Left “exception: log of negative”
          | otherwise = Right $ log x
```

But now: a broken pipeline

$$\texttt{[Float]} \xrightarrow{\texttt{head}} \texttt{Float} \xrightarrow{\texttt{log}} \texttt{Float}$$

```
safeHead [] = Left “exception: head []”
safeHead (x:xs) = Right x

safeLog x | x<=0 = Left “exception: log of negative”
          | otherwise = Right $ log x
```

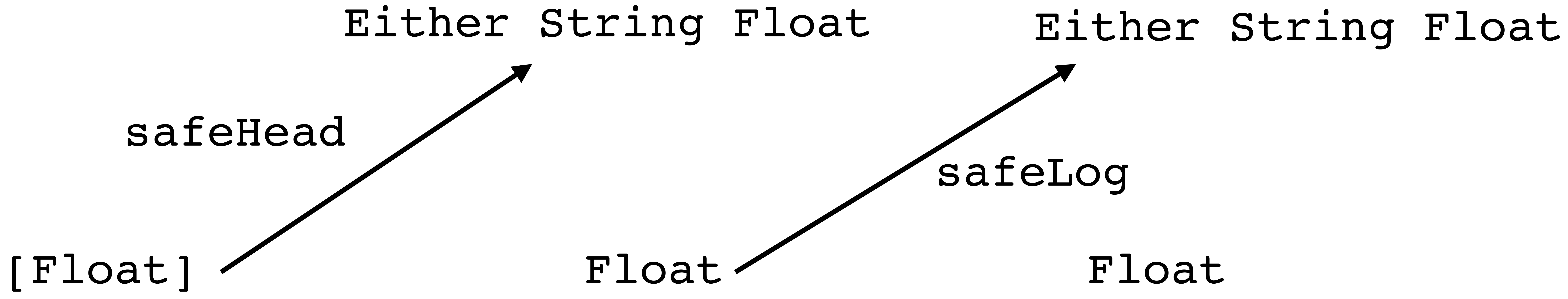

$$\begin{tikzcd}
& \texttt{Either String Float} & & \texttt{Either String Float} \\
\texttt{[Float]} \arrow[ur, "\texttt{safeHead}"] & & \texttt{Float} \arrow[ur, "\texttt{safeLog}"'] & & \texttt{Float}
\end{tikzcd}$$

But now: a broken pipeline

```
safeLog $ safeHead [1..4]
<interactive>:1:1: error:
    • Non type-variable argument in the constraint: Floating (Either String b)
      (Use FlexibleContexts to permit this)
    • When checking the inferred type
        it :: forall b. (Ord b, Floating (Either String b), Enum b, Num b) =>
Either String (Either String b)
```



```
          | otherwise = Right $ log x
```



Typeclasses and higher-order functions to the rescue

Problem: We need a way of applying a function of type (in this instance): Float -> Either String Float to a value of type: Either String Float.

Analogy: Haskell has higher-order functions, and one of these is ordinary application: ($) :: (a -> b) -> a -> b

Solution: Following that model we just need to equip Either with a strange application function:

(>>=) :: Either String Float -> (String -> Either String Float) -> Either String Float

or more generally:

(>>=) :: Either String a -> (a -> Either String b) -> Either String b

- Read this as value feeding into function, so we are finding a way to feed a value of type Either String a into a function that expects a value of type a. This function is called **bind**.

Typeclasses and higher-order functions to the rescue

(>>=) :: Either String a -> (a -> Either String b) -> Either String b

- Read this as value feeding into function, so we are finding a way to feed a value of type Either String a into a function that expects a value of type a.


```
(>>=) :: Either String a -> (a -> Either String b) -> Either String b
(>>=) (Left s) f = Left s
(>>=) (Right a) f = f a
```

- So in (e >>= f), if the evaluation of e produces an error, we get that, otherwise it terminates normally, and we just feed the value produced into f.

Typeclasses and higher-order functions to the rescue

- We also (less obviously) need a way of converting a basic value of type b to something of type Either String b.

- This is the purpose of the return function.


```
return :: b -> Either String b
return b = Right b
```

Typeclasses and higher-order functions to the rescue

- So what we need is a typeclass that specifies types that have these functions.
- This is (in essence) the monad class.
- Below taken from Prelude.


```
class Applicative m => Monad m where
    -- @
    (>>=)       :: forall a b. m a -> (a -> m b) -> m b

    -- | Inject a value into the monadic type.
    return      :: a -> m a
```

Typeclasses and higher-order functions to the rescue

```
class Applicative m => Monad m where
    -- @
    (>>=)       :: forall a b. m a -> (a -> m b) -> m b

    -- | Inject a value into the monadic type.
    return      :: a -> m a
```



```
instance Monad Either a where
  (>>=) :: Either String a -> (a -> Either String b) -> Either String b
  (>>=) (Left s) f = Left s
  (>>=) (Right a) f = f a
  return :: b -> Either a b
  return b = Right b
```

Summary

- In Haskell, effects are encapsulated in the return types of functions.
- Monads allow us to compose functions that may produce effects.
- They are a type-class that contains two operations:
  - bind (>>=) provides composition
  - return allows us to include the pure world in the effectful.

ECS713: Functional Programming week 9: Monads and Effects

Part 2: Example of computational effects as monads

Aims

- To give some examples of monads and how they relate to computational effects.

Learning Objectives

- Be familiar with the Haskell monads for:
  - exceptions (Either)
  - non-determinism (Lists)
  - logs (Pair)
  - state
  - IO

Basic Idea

- In the pure world a function that takes input a and produces output b gets type a -> b

- In the effectful world, a function that takes input a and produces output b gets type a -> m b

- Here, m is a type constructor, a monads, that encapsulates the effect.

- So a -> mb could be a -> IO b, if the function does IO

- Or it could be a -> (Either String b), if the function produces an exception.

- IO and Either are both monads.

Monads

```
class Applicative m => Monad m where
    -- @
    (>>=)       :: forall a b. m a -> (a -> m b) -> m b

    -- | Inject a value into the monadic type.
    return      :: a -> m a
```

Exceptions

- A function that produces an exception can either:
  - return normally
  - or produce an exception

- This is modelled by a datatype that has two constructors:
  - Right: for normal values
  - Left: for exceptions

Either

```
data  Either a b  =  Left a | Right b
  deriving ( Eq, Ord, Read, Show )
```



```
instance Monad (Either a) where
  (>>=) :: Either a b -> (b -> Either a c) -> Either a c
  (>>=) (Left a) f = Left a
  (>>=) (Right b) f = f b
  return :: b -> Either a b
  return b = Right b
```

```haskell
(>>=) :: Either String b -> (b -> Either String c) -> Either String c
 (>>=) (Left s) f = Left s
 (>>=) (Right b) f = f b
```

- In this monad
  - e >>= f works out as follows:

    - if e produces an exception: Left “whatever”, we get that as result.
    - otherwise it produces Right a, and we return (f a), which can either be:
      - an exception: Left “another error”
      - or Right b.

Either

- The inclusion of pure values is just the function Right
  - return = Right

Non-determinism: lists

- The standard way to model non-determinism is to produce the list of all possible results.

- Haskell is a lazy functional language, which means it will not compute more than necessary. Typically this might be the first element of this list.

- So if you were writing a program to model coin-tossing,

  - the result of a single toss would be the list of all possible outcomes: [Head, Tail]

  - the result of two tosses would be [[Head,Head],[Head,Tail],[Tail,Head],[Tail,Tail]].

List monad

```
instance Monad [a] where
  (>>=) :: [a] -> (a -> [b]) -> [b]
  (>>=) as f = concat $ map f $ as

  return :: b -> [b]
  return b = [b]
```


- In this monad
  - a pure value is deterministic, and so has only one possible result.
  - return sends that value to the list that contains only that one element.
- e >>= f works out as follows:
  - e produces a list of possible results,
  - we apply f to each of those results to get a list of lists of possible answers
  - we concatenate all those possible answers together to get the final list of possibilities

List monad

```
instance Monad [ ] where
  (>>=) :: [a] -> (a -> [b]) -> [b]
  (>>=) as f = concat $ map f $ as

  return :: b -> [b]
  return b = [b]
```



```
-- initial value: two possibilities
e = [1,2]
-- function, input produces 3 possibilities, all the same
f = replicate 3
```

```
map f e
[[1,1,1],[2,2,2]]
e >>= f
[1,1,1,2,2,2]
```

Logs

- With exceptions we get either a value or an exception

- With logs we get both a value and a log.

- This means we model the result as the pair (log,value), and so as a member of the type (String, b).

- Or more accurately as a datatype that is isomorphic to this

Logs

```
data  Log a  =  Log String a
```



```
instance Monad Log where
  (>>=) :: Log a -> (a -> Log b) -> Log b
  (>>=) (Log s a) f = let (Log s’ b) = f a in (Log (s++s’) b)

  return :: b -> Log b
  return b = Log “” b
```

State

- State is more complicated. Let’s suppose we have a type State, representing the possible states we are interested in.
- If we have a function that uses and changes the State, then the data the function uses is its parameter and the initial State. The result it produces is the result of the function and the final changed State. So we model the function as having type:
  - (State,a) -> (State,b)
- But this does not fit the a -> m b pattern, so we curry it, switching the order of the parameters to:
  - a -> State -> (State, b).
- This does fit the a -> m b pattern with m b = State -> (State, b)

State

```
– type State s a = s -> (s,a)
data State s a  =  State (s -> (s,a))
```

```
instance Monad State s where
  (>>=) :: State s a -> (a -> State s b) -> State s b
  (>>=) (State f) g =
     State (\s ->
       let (s',b) = f s
           State h = g b
       in h s’)

  return :: a -> State s a
  return a = State f where f s = (s,a)
```

IO

- IO is still more complex because we don't have any type expression that represents it.

- This means we can't implement bind and return in the same kind of way.

- But…

IO

- We know that
  - return “a string” :: IO String
  - so return (part of the monad structure) is just return (part of the IO structure)

IO

```
(>>=) :: IO a -> (a -> IO b) -> IO b
(>>=) e f = do
     x <- e
     f x
```


- We can program up bind using this do block.
- Later we will see that we can use do blocks with any monad, and that this is always an equivalent of bind.

Summary

- We’ve now seen how five different types of effect can be implemented as monads.

- In each case bind and return make sense.

ECS713: Functional Programming
week 9: Monads and Effects

Part 3: Functors, Applicatives and Equations

Aims

- To fill in some details about:

  - how Monads are built on other typeclasses
  - how to declare them in practice
  - what properties the operations bind and return must have.

Learning Objectives

- Be familiar with the typeclasses:
  - Functor
  - Applicative
  - and how they relate to Monads
- Know how to declare monads.
- Understand the equations needed for the Functor and Monad classes

Monad is an extension

- The type class Monad is an extension of two other classes
  - Functor
  - Applicative
- **This means every monad also has to be made an instance of these.**
- The declarations given in video 2 will not work because of this.

Monad equations

- To work in the way we expect, bind and return must have certain properties.

- These are expressed as equations.

Monad is an extension: Functor

- Functor is the typeclass of types that have a map function.
- Every Monad is also a Functor.

Functor

- Functor is the typeclass of types with a "map" operation:

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


- Every Monad is also a Functor, with map defined as:

```
fmap f ta = ta >>= (return . f)
```

Functor

- fmap is the functor equivalent of map

- Often, an element of t a is given by some shape (eg a list) that has entries tagged with elements of a

- fmap leaves the shape the same but applies the function to each of the elements

- This is fine for unary functions, but there is no similar recipe for binary ones, a -> b -> c not just a -> b.

Applicative

```
class Functor t => Applicative t where
    {-# MINIMAL pure, ((<*>) | liftA2) #-}
  pure :: a -> t a
  liftA2 :: (a -> b -> c) -> t a -> t b -> t c
  (<*>) :: t (a -> b) -> t a -> t b
```


- liftA2 is the binary equivalent of fmap
- <*> is a souped up version of unary fmap
- You only need one of these.

<*> and liftA2 are equivalent

```haskell
class Functor t => Applicative t where
    {-# MINIMAL pure, ((<*>) | liftA2) #-}
   pure :: a -> t a
   liftA2 :: (a -> b -> c) -> t a -> t b -> t c
   (<*>) :: t (a -> b) -> t a -> t b
```



```haskell
-- does not use functor
(<*>) = liftA2 id
-- uses functor
liftA2 f x y = (fmap f x) <*> y = ((pure f) <*> x) <*> y
```

Every Monad is an Applicative

```
class Functor t => Applicative t where
    {-# MINIMAL pure, ((<*>) | liftA2) #-}
   pure :: a -> t a
   liftA2 :: (a -> b -> c) -> t a -> t b -> t c
   (<*>) :: t (a -> b) -> t a -> t b
```



```
pure = return
liftA2 f x y = (x >>= (return . f)) >>=  (\x1 -> y >>= (\x2 -> return (x1 x2))
tf <*> ta = tf >>= (\x1 -> ta >>= (\x2 -> return (x1 x2)))
```

Every Monad is an Applicative

```
class Functor t => Applicative t where
    {-# MINIMAL pure, ((<*>) | liftA2) #-}
   pure :: a -> t a
   liftA2 :: (a -> b -> c) -> t a -> t b -> t c
   (<*>) :: t (a -> b) -> t a -> t b
```



```
pure = return

liftA2 f tx ty = do
   x <- tx
   y <- ty
   return f x y

tf <*> ta = do
  f <- tf
  a <- ta
  return f a
```

Equations

- Formally membership of a type class only requires that we implement functions of particular types.

- But they are often required to have particular properties.

- For example, in Ord, <= is required to be an ordering, not an arbitrary function a -> a -> Bool.

- In Functor, Applicative and Monad all require certain properties of their operations.

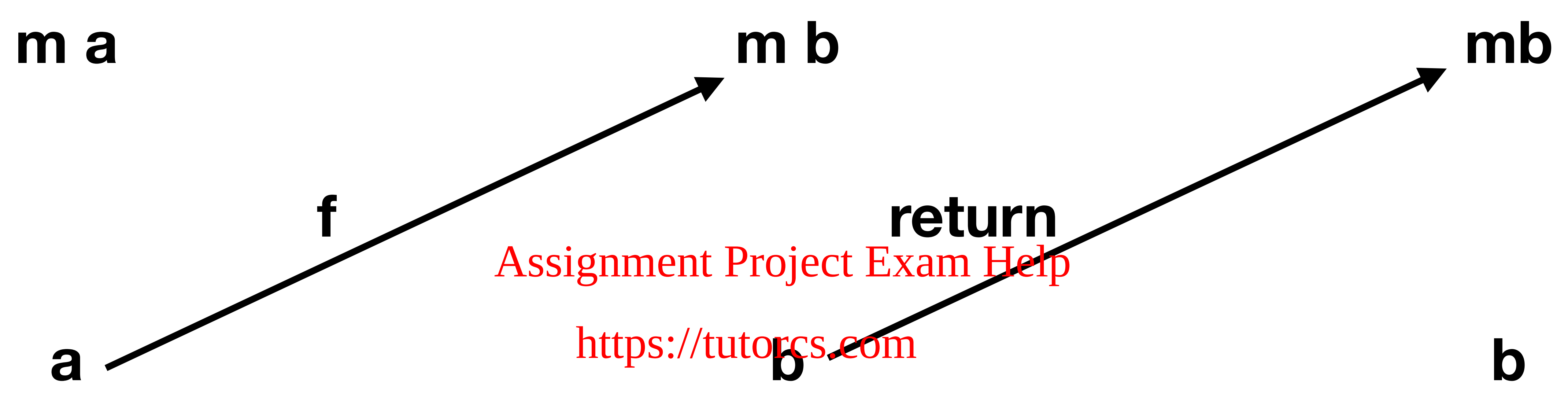
- These ensure that code behaves as expected.

Functor equations

$$
\begin{tikzcd}
t\,a \arrow[r, "\text{fmap } f"] & t\,b \arrow[r, "\text{fmap } g"] & t\,c \\
a \arrow[r, "f"] & b \arrow[r, "g"] & c
\end{tikzcd}
$$

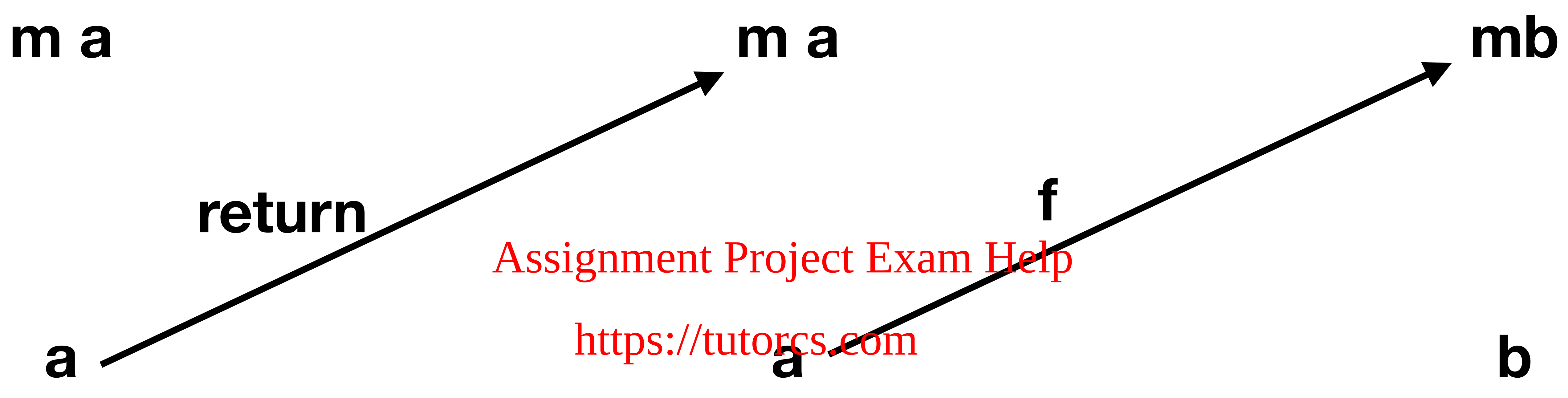

```
– fmap of the identity on a is the identity on ta
fmap id == id

– we might compose f and g before or after applying fmap
fmap (g . f) == (fmap g) . (fmap f)
```

Monad equations



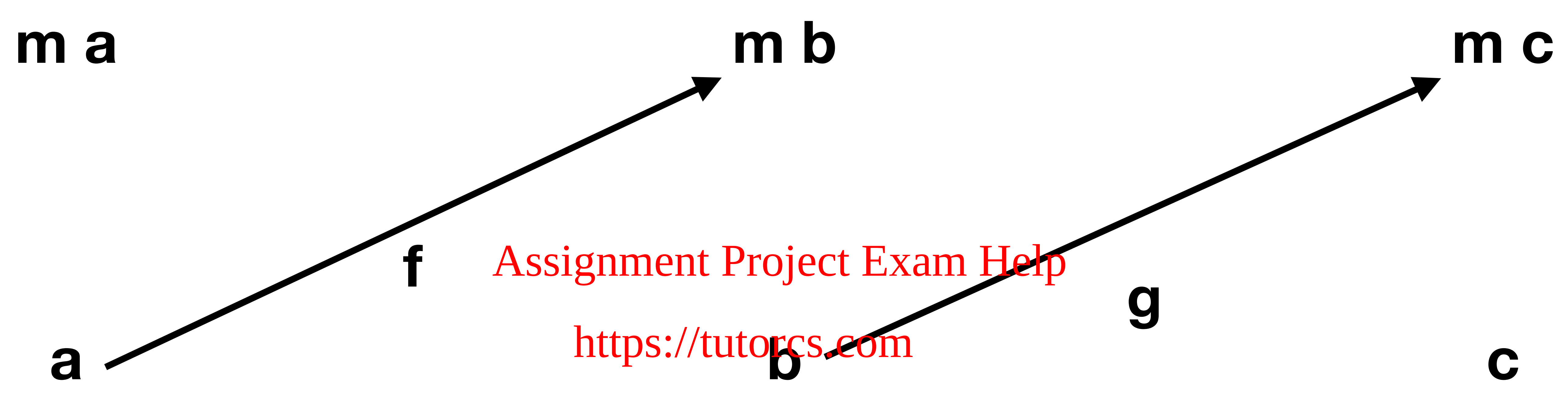

```
– binding into return does nothing
mb >>= return == mb
```

Monad equations




```
– return and bind is just function application
return a >= f = f a
```

Monad equations




```
- There are two ways to get from ma to mc.
- Apply successive binds
- Or use the monadic composite of f and g, and bind into it.
- They produce equal results.
(ma >>= f) >>= g == ma >>= (\a -> f a >>= g)
```

Example Declaration

```
data Log a = Log String a

instance Monad Log where
  return a = Log “” a
  (Log s a) >>= f = let (Log s’ b) = f a in Log (s++s’) b

– These two are necessary
instance Applicative Log where
  pure = return
  tf <*> ta = tf >>= (\x1 -> ta >>= (\x2 -> return (x1 x2)))

instance Functor Log where
  fmap f ta = ta >>= (return . f)
```

Example Declaration

```
data Log a = Log String a

instance Monad Log where
  return a = Log “” a
  (Log s a) >>= f = let (Log s’ b) = f a in Log (s++s’) b

– These two are necessary
instance Applicative Log where
  pure = return
  tf <*> ta = do
    f <- tf
    a <- ta
    return (f a)

instance Functor Log where
  fmap f ta = do
    a <- ta
     return (f a)
```

Summary

- Functor is the typeclass of types with a unary map operator.
- Applicative is the typeclass of types with a binary map operator.

- Monad extends both of these.
- Bind and return have to satisfy equations.
- Monad declarations have to include boilerplate making the Monad into a Functor and an Applicative.