|  |  |  |
| --- | --- | --- |
| Edward Kmett's answer is obviously great. But, it is a bit technical. Here is a perhaps more accessible explanation.  Free monads are just a general way of turning functors into monads. That is, given any functor f Free f is a monad. This would not be very useful, except you get a pair of functions  liftFree :: Functor f => f a -> Free f a  foldFree :: Functor f => (f r -> r) -> Free f r -> r  the first of these lets you "get into" your monad, and the second one gives you a way to "get out" of it.  More generally, if X is a Y with some extra stuff P, then a "free X" is a a way of getting from a Y to an X without gaining anything extra.  Examples: a monoid (X) is a set (Y) with extra structure (P) that basically says it has an operations (you can think of addition) and some identity (like zero).  so  class Monoid m where  mempty :: m  mappend :: m -> m -> m  now, we all know lists  data [a] = [] | a : [a]  well, given any type t we know that [t] is a monoid  instance Monoid [t] where  mempty = []  mappend = (++)  and so lists are the "free monoid" over sets (or in Haskell types).  Okay, so free monads are the same idea. We take a functor, and give back a monad. In fact, since monads can be seen as monoids in the category of endo functors, the definition of a list  data [a] = [] | a : [a]  looks a lot like the definition of free monads  data Free f a = Pure a | Roll (f (Free f a))  and the Monad instance has a similarity to the Monoid instance for lists  --it needs to be a functor  instance Functor f => Functor (Free f) where  fmap f (Pure a) = Pure (f a)  fmap f (Roll x) = Roll (fmap (fmap f) x)  --this is the same thing as (++) basically  concatFree :: Functor f => Free f (Free f a) -> Free f a  concatFree (Pure x) = x  concatFree (Roll y) = Roll (fmap concatFree y)  instance Functor f => Monad (Free f) where  return = Pure -- just like []  x >>= f = concatFree (fmap f x) --this is the standard concatMap definition of bind  now, we get our two operations  -- this is essentially the same as \x -> [x]  liftFree :: Functor f => f a -> Free f a  liftFree x = Roll (fmap Pure x)  -- this is essentially the same as folding a list  foldFree :: Functor f => (f r -> r) -> Free f r -> r  foldFree \_ (Pure a) = a  foldFree f (Roll x) = f (fmap (foldFree f) x)   |  |  | | --- | --- | |  |  | |