Haskell patterns Cheat Sheet

by logcat via cheatography.com/27075/cs/12218/

Composition and application

`(.) :: (b -> c) -> (a -> b) -> a -> c`	function composition
`($) :: (a -> b) -> a -> b`	application operator, has low, right-associative binding precedence, for example `f $ g $ h x = f (g (h x))`

Monoid

typeclass where empty/append are defined

`mempty :: M onoid a => a`	identity of mappend (mappend mempty x = x)
`mappend :: Monoid a => a -> a -> a`	append two monoids (associative: brackets does not matter) `mappend x (mappend y z) = mappend (mappend x y) z`
`<> :: M onoid m => m -> m -> m`	infix synonym for mappend (`" he" <> " llo "`)
`mconcat :: [a] -> a`	fold list using mappend and mempty

Functor

typeclass where `fmap` (`map`/`<$>`) is defined

should satisfy laws	`fmap id == id` `fmap (f . g) == fmap f . fmap g`
`fmap :: Functor f => (a -> b) -> f a -> f b`	map function over functor `fmap (+1) (Just 3)` is `Just 4`
`<$> :: Functor f => (a -> b) -> f a -> f b`	function mapped over functor infix synonym for `fmap` `(+1) <$> (Just 3)` is `Just 4`

Applicative

typeclass where `pure`/`<*>` are defined

have `Functor` as super class	every instance of `Applic ative` must have instance of `Functor` so `fmap` (`map`/`<$>`) can be used

`pure :: Applic ative f => a -> f a`	create an instance of Applicative
	`pure 3 :: [Int]` is `[3]`
	`pure 3 :: Maybe Int` is `Just 3`
	`pure (+3) :: Maybe (Int -> Int)` is `Just` a function from Int to Int
	`pure (+3) :: [Int -> Int]` is list of function
	`pure 1 :: IO Int` is how it is printed in ghci

`(<* >) :: Applic ative f => f (a -> b) -> f a -> f b`	sequential application / apply
	`Just (+1) <*> Just 1 :: Maybe Int` is `Just 2`
	`[(+1), (+2)] <*> [0] :: [Int]` is `[1, 2]`

Monad	
typeclass	
have `Applic ative` as super class	every instance of `Monad` must have instance of `Applic ative` and `Functor` so `fmap` (`map`/`<$>`) and `<*>`/`pure` can be used
`return :: Monad m => a -> m a`	is `pure`

`(>>) :: Monad m => m a -> m b -> m b`	sequentially compose two monads, first is usually
	`Just 2 >> Just 3` is `Just 3`
	`Nothing >> Just 3` is `Nothing`
	`[9, 9] >> [0, 0, 0]` is `[0,0,0 ,0,0,0]`

`(>>=) :: Monad m => m a -> (a -> m b) -> m b`	bind, sequentially compose two monads, value of first passed as argument to the second
	`Just 3 >>= \x -> Just (x + 1)` is `Just 4`
	`Nothing >>= \x -> Just (x + 1)` is `Nothing`
	`[0, 0] >>= \x -> [x + 1]` is `[1, 1]`
	`[0, 0] >>= \x -> [x + 1, 2]` is `[1,2,1,2]`
	`[] >>= \x -> [x + 1]` is `[]`