Command | Function	**Description**
type type IntArr = [Int]	This is used as a definition of a type using predefined types. Aka this combines types to another type.
newtype newtype IntArr = [Int] instance Eq...	This achieves the same thing with the only difference being the ability to call instances like Eq, Show and more.
data data BinaryTree = Leaf | Branch Tree Tree	This is the most complex version that allows you to branch with the | operator and also allows instances of Eq, Show and more.
case x of case ok of True... False...	this is a fancy version of if then else. You define the cases below like you would in a function with pattern matching -> |
let x in let x = 5 in x + 10	define a variable in the following code. Most used in the do notation, can also be chained together, which is weird but super useful.
Lambda \x -> x * 5	This is a function without a real name. The parameter is passed implicitly from the right into the function via replacing ever y x(name of function) inside the function.
Currying (a -> b -> c) (a -> b) -> (b -> c)	In haskell the functions are usually unary, meaning the take 1 input and create 1 output. Therefore haskell does syntactic sugar in order to provide more complex functions. This is called Currying.
foldr foldr func end List	Foldr applies a function (a -> b) for one element to an entire list and returns a 'b' -> the return type of the function. This differs to map in that the return type is defined by the function and doesn't stay a list. foldr (+) 0 [1,4,8] would be 1+4+8+0 -> 13 : note that foldr is defined as the following : foldr f end (x:xs) = x 'f' foldr f end xs
foldl foldl func end List	foldl is very weird, it essentially applies the function to the end element every step. In other words, this is a weird mix between iterative and recursive. example foldl (+) 1 [1,2,3] -> foldl f (f end x) xs -> end is now 2 -> go to 3 add 2 -> 5. Here it is the same result as with foldr, but with division the result would be very different! foldr div 1 [1,2] is an error div by 0! foldl div 1 [1,2] is simply 0 due to the iterative nature.
map map func List	This maps a function to an entire list and returns the list of the results. (+3) [1,5,8] would return [4,8,11]
zip zip list1 list2	This puts together 2 lists in pairs. -> zip [1 .. 10] [a .. z] -> [(1,a),(2,b),(3,c) .. (10,j)] Note that the zip functions ends when one of the list returns nothing -> aka the list is at its end. In other words (11,k) or (,k) will not be in the result.
list comprehension ex: [x + 5 | x <- [0 .. 5]] would return [5,6,7,8,9,10]	This is an easy way to calculate with lists. Important note, if you want to combine 2 lists in a list comprehension, then you need to know that it doesn't zip perfectly. If you want that functionality, then you need to use the zip command explicitly. Otherwise it does things like this [(1,1),(1,2),(1,3),(1,4)....] FUN
dot notation (+3).head.reverse [5,6,3,4] 4 + 3 -> 7	This is syntactic sugar in order to make chaining commands easier by removing the () around the functions. Note that due to the way the dot notation works you don't need to specify the parameter in the function declaration, it is passed implicitly.
Dollar sign $	the dollar sign is an easy way to remove (). Example: (+3) $ 5 + 3 -> (5+3) + 3
the where notation a = b + 5 where b = 5	This is essentially the opposite way to the let notation, while achieving the same result.

pure pure func -> wrappedFunc	This is essentially the wrap function that will allow you to apply functions to wrapped values. It is defined in the applicative class. (see below) Also note that the pure function often only shows one type: for example the pure function for Maybe only shows Just and doesn't take the function as an explicit parameter -> pure = Just
Functor fmap func wrappedValue fmap f (Just a) = Just(f a) fmap f Nothing = Nothing	A functor is a type that can be modified by regular functions via the fmap function. example: (+3) to Maybe Int -> fmap (+3) (Just 4) -> 7 while fmap (+3) Nothing -> Nothing fmap::(a -> b) -> f a -> f b where f is a wrapper type!
Applicative Maybe func <*> Maybe Int (Just f) <*> v = fmap f v Nothing <*> v = Nothing	Applicative::() Apply a wrapped function to a wrapped value. The interesting part here is that the function can also be the return type of another function. Essentially allowing you to chain functions. -> pure func <$> val1 <*> val2 <*> val3 (<*>):: f (a -> b) -> f a -> f b
Monad wrappedValue »= func (Just v) »= f = f v Nothing »= f = Nothing	The Monad has many more functions but the described to the left is the bind operator, it allows you to do the same with as with the functor, just this time you can chain it because of the applicative class. (»=) :: f a -> (a -> f b) -> f b
Alternative	The useful operator <|> that allows you to make choices, aka if the first fails then try the second. ex: choice = parseLevel1 <|> parseLevel2 <|> parseLevel3
do notation do x <- monadtype y <- monadtype expression|return statement	This is used to easily evaluate a function dealing with Monads. The example is from the parser for the lambda interpreter.
curry and uncurry	curry:: ((a,b) -> c) -> (a -> b -> c) curry f = \x -> (\y -> f (x,y)) uncurry::(a -> b -> c) -> ((a,b) -> c) uncurry f = \(x,y) -> f x y
Equational style	double 2 = {applying double } 2 + 2 = {applying +} = 6
prefix and infix	infix : (#+) -> 1 + 2 OR (#+) 1 2 definition: (#+) a b = a + b OR a #+ b = a + b prefix : div 4 2 = 2 OR 4 'div' 2 definition div a b = a / b Note, text strings can't be defined as infix