Haskell CheatSheet - https://github.com/isthissuperuser | **legenda**: { } optional, [ ] one or more,

MISC

```haskell
3
data Tree2 a = Nil | B a (Tree2 a) (Tree2 a)
data Gtree a = Nil | Gtree a [Gtree a] deriving (Show)
data Name a = Name {getterX, getterY :: a}
 -- constructor takes 2 arg of different type a and also expose the getters
let var1 = value1
    var2 = value2
in  body
$ -- evaluate before right expression
show -- print
v@(T x' l' c' r') -- pick up data on the right inside v
if condition then body else body
f (x, y) = x -- couple as arg
```

LISTS

```haskell
[a] -- list of variable type
list !! 0 -- take element at index 0
[1, 3 .. 10] -- [1, 3, 6, 9]
1:[] -- cons operator: single: list not contrary
++ -- list concat
take 3 list -- return first 3 elems
drop 3 list -- remove first 3 elems and return new list
head list -- return first element
tail list -- return list without first element
last l -- return last elem from list
init l -- returns all elem of l except last
reverse list -- reverse list
lst [ x * 2 | x <- [0, 1 ..]] -- even numbers
zip lst1 lst2 -- couples from each lst
null v -- returns true if v empty
length l -- returns the length of the list
```

FUNCTIONS

```haskell
fname arg = body
map (f1 . f2) lst -- first map f2 then f1
concatMap f lst -- for every elem f creates a list, concatMap concats all the lists into a single list
(\ x -> x * 2) -- lambda
fname arg@(cur:next:rest) = body -- arg is list decomposed
fname num
 | f1 < 2 = "small"
 | f1 > 2 = "big"
 | otherwise = "normal"
 where
   f1 = fbody
id -- identity function
4 `div` 2 -- integer division
filter f v -- f boolean function, returns a list of the elements of v for which f returned true
maximum v--returns v max element
```

CLASSES

```haskell
class Equal a where -- pseudo class
   (==) :: a -> a -> bool
   x /= y = not (x == y)
instance (Equal a) => Equal (Tree2 a) where
   Nil == s = s == Nil
   s == Nil = s == Nil
   (B v l r) == (B v2 l2 r2) = v == v2 && l == l2 && r == r2
   _ == _ = False
-- instantiating Equal pseudo class with a Tree as arg
```

FOLDABLE

```haskell
instance Foldable Slist where
  foldr f acc (Slist list len) = foldr f acc list
instance Foldable Tree where
  foldr :: (a -> b -> b) -> b -> Tree a -> b
  foldr f acc Nil = acc
  foldr f acc (Leaf x) = f acc x
  foldr f acc (B l r) = foldr f (foldr f acc r) l
```

FUNCTORS

```haskell
instance Functor Slist where
  fmap :: (a -> b) -> Slist a -> Slist b
  fmap f (Slist list len) = Slist (fmap f list) len
instance Functor Tree where
  fmap :: (a -> b) -> Tree a -> Tree b
  fmap f Nil = Nil
  fmap f (Leaf a) = Leaf (f a)
  fmap f (B l r) = B (fmap f l) (fmap f r)
```

APPLICATIVES

```haskell
--[(+1), (*2), (^3)] <*> [1,2,3]
--[2, 3, 4, 2, 4, 6, 1, 8, 27]
-- partial f applied to list and concat
instance Applicative Slist where
  pure :: a -> Slist a
  pure a = Slist [a] 1
  (<*>) :: Slist (a -> b) -> Slist a -> Slist b
  (Slist flist _) <*> (Slist a alen) = Slist (flist <*> a) alen
instance Applicative Tree where
  pure x = Leaf x
  Nil <*> _ = Nil
  _ <*> Nil = Nil
  (Leaf f) <*> (Leaf x) = Leaf (f x)
  (Leaf f) <*> (B left right) = B (Leaf f <*> left) (Leaf f <*> right)
  (B leftF rightF) <*> tree = B (leftF <*> tree) (rightF <*> tree)
instance Applicative Tree where
  pure :: a -> Tree a
  pure = Leaf
  (<*>) :: Tree (a -> b) -> Tree a -> Tree b
  Nil <*> _ = Nil
  _ <*> Nil = Nil
  (Leaf f) <*> t = fmap f t
  (B t1 t2) <*> t = B (t1 <*> t) (t2 <*> t)
ltconcat :: BTT (BTT a) -> BTT a
ltconcat t = foldr (<++>) Nil t
ltconcmap :: ((a->b) -> BTT b) -> BTT (a -> b) -> BTT b
ltconcmap f t = ltconcat (fmap f t)
instance Applicative BTT where
  pure :: a -> BTT a
  pure x = B2 x Nil Nil
  (<*>) :: BTT (a -> b) -> BTT a -> BTT b
  x <*> y = ltconcmap (\f -> fmap f y) x
(+++) :: Gtree a -> Gtree a -> Gtree a
Nil +++ s = s
s +++ Nil = s
(Gtree a l) +++ g = Gtree a (g:l)
instance Applicative Gtree where
  pure a = Gtree a []
  x <*> y = foldr (+++) Nil (fmap (\f -> fmap f y) x)
```

MONADS

```haskell
instance Monad Result where
   Ok x >>= f = f x
   Err >>= _ = Err
instance Monad Slist where
  (>>=) :: Slist a -> (a -> Slist b) -> Slist b
  (Slist list len) >>= f = let finalL = (list >>= (\x -> let Slist xs _ = f x in xs)) in Slist finalL $ length finalL
instance Monad Tree where
  (>>=) :: Tree a -> (a -> Tree
```

```
  b) -> Tree b
 Nil >>= f = Nil
 (Leaf a) >>= f = f a
 (B t1 t2) >>= f = B (t1 >>= f)
   (t2 >>= f)
```

STATE MONADS

```
import Control.Monad.State
type Stack = [Int]
popM :: State Stack Int
popM = state $ \(x : xs) -> (x,
  xs)
pushM :: Int -> State Stack ()
pushM a = state $ \xs -> ((), a :
   xs)
stackManipM :: State Stack Int
stackManipM = do
  pushM 3
  a <- popM
  popM
state0 = [1,2,3,4,5]
result = runState stackManipM
  state0
```

Examples

```
data Btree a = Leaf a | Branch (
  Btree a)(Btree a) deriving (
  Show, Eq)
instance Functor Btree where
    fmap f (Leaf x) = Leaf (f x)
    fmap f (Branch x y) = Branch
    (fmap f x) (fmap f y)
addLevel :: Btree a -> Btree a
addLevel t = Branch t t
btrees :: a -> [(Btree a)]
btrees x = (Leaf x) : [addLevel
  t | t <- btrees x]
incBtrees :: [Btree Integer]
incBtrees = (Leaf 1) : [
  addLevel (fmap (+1) t) | t <-
   incBtrees]
counts :: [Integer]
counts = map (\x -> 2^x - 1)
  [1..]
```
