MEKING HOUR ON TAIPS AND TAIPKLASHES

```
data Bool = False | True
data Shape = Circle Float Float Float | Rectangle Float Float Float Float

λ :t Circle
Circle :: Float → Float → Float → Shape

surface :: Shape → Float
surface (Circle _ _ r) = pi * r^2
surface (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) * (abs $ y2 - y1)

λ surface $ Circle 10 20 10
314.15927
λ surface $ Rectangle 0 0 100 100
10000.0
λ Circle 10 20 10 ^    cant show

data Shape = ... deriving (Show)
λ Circle 10 20 10 ✓
Circle 10.0 20.0 10.0

λ map (Circle 10 20) [4,5,6,7]
[Circle 10.0 20.0 4.0, Circle 10.0 20.0 5.0, ...]

data Point = Point Float Float deriving (Show)
data Shape = Circle Point Float | Rectangle Point Point ...

surface :: Shape → Float
surface (Circle _ r) = ...
surface (Rectangle (Point x1 y1) (Point x2 y2)) = ...
```

```
nudge :: Shape → Float → Float → Shape
nudge (Circle (Point x y) r) a b = Circle (Point (x+a) (y+b)) r
...

module Shapes (
  Point (..),
  Shape (..),          -- or Shape (Rectangle, Circle)
  surface,
  nudge,
) where

data Person = Person {
    firstName :: String,
    lastName :: String,
    age :: Int,
    height :: Float,
    phoneNumber :: String,
    flavour :: String,
} deriving (Show)

λ:t flavour
flavour :: Person → String
λ:t height
height :: Person → Float

λ Person {firstName = "As", lastName = "Rao", ...}
Person {...}
```

record syntax

data Maybe a = Nothing | Just a
(what type of empty list? [] ⟶ [a]

λ:t Just 84
Just 84 :: (Num t) => Maybe t
λ :t Nothing
Nothing :: Maybe a
λ Just 10 :: Maybe Double
Just 10.0

data (Ord k) => Map k v =...
↳ dont use, have to use everywhere

data Vector a = Vector a a a deriving (Show)

vplus :: (Num t) => Vector t -> Vector t -> Vector t
(Vector i j k) `vplus` (Vector l m n)
= Vector (i+l) (j+m) (k+n)

vmult...
vscale...

data Person = Person {
name :: String,
age :: Int
} deriving (Eq)

```
λ let mike = Person {name = "Mike", age = 43}
λ mike == Person {...}
True
λ mike `elem` [Person {...}, ...]
True

data Person = Person {...} deriving (Eq, Show, Read)
λ show mike
"Person {name = \"Mike\", age = 43.}"
λ read "Person {...}" == mike
True

λ read "Just 't'" :: Maybe char
Just 't'

data Bool = False | True deriving (Ord)
λ True `compare` False
GT
λ True < False
False
λ Nothing < Just 100
True
λ Just 3 `compare` Just 2
GT

data day = Mon | Tue | ... | Sun
        deriving (Eq, Ord, Show, Read, Bounded, Enum)
```

λ show Wed
"Wed"
λ read "Wed" :: Day
Wed
λ Sat == Sat
True
λ Mon `compare` Wed
LT
λ minBound :: Day
Mon
λ maxBound :: Day
Sun
λ succ Mon
Tue
λ pred Fri
Thu
λ [Thu .. Sun]
[Thu, Fri, ...]
λ [minBound .. maxBound] :: Day
[Mon, Tue, ...]

type String = [Char]

type PhoneNumber = String
type Name = String
type PhoneBook = [(Name, PhoneNumber)]

type AssocList k v = [(k, v)]

```
type IntMap v = Map Int v
type IntMap = Map Int

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

λ :t Right 'a'
Right 'a' :: Either a Char

data List a = Empty | Cons a (List a)
        deriving (show, Read, Eq, Ord)

λ Empty
Empty
λ 5 `Cons` Empty
Cons 5 Empty
λ 3 `Cons` (4 `cons` (5 `Cons` Empty))
Cons 3 (Cons 4 (Cons 5 Empty))

infixr 5 :-:
data List a = Empty | a :-: (List a)
        deriving (show, Read, Eq, Ord)

λ 3 :-: 4 :-: 5 :-: Empty
(:-:) 3 ((:-:) 4 ((:-:) 5 Empty))
```

```
infixr 5 ++
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

infixr 5 .++
(.++) :: List a -> List a -> List a
Empty    .++ ys = ys
(x :-: xs) ++ ys = x :-: (xs ++ ys)
```

pattern matching works on constructors (only)

```
data Tree a = EmptyTree
  | Node a (Tree a) (Tree a) deriving (Show, Read, Eq)

singleton :: a -> Tree a
singleton x = Node x EmptyTree EmptyTree

treeInsert :: (Ord a) => a -> Tree a -> Tree a
treeInsert x EmptyTree = singleton x
treeInsert x (Node a left right)
  | x == a = Node x left right
  | x < a  = Node a (treeInsert x left) right
  | otherwise = Node a left (treeInsert x right)
```

```
treeElem :: (Ord a) => a -> Tree a -> Bool
treeElem x EmptyTree = False
treeElem x (Node a left right)
  | x == a = True
  | x < a = treeElem x left
  | otherwise = treeElem x right
```

```
λ let nums = [8, 6, 4, 1, 7, 3, 5]
λ let numsTree = foldr treeInsert EmptyTree nums
Node 5 (...) (...)

λ 8 `treeElem` numsTree
True
λ 100 `treeElem` numsTree
False
```

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x == y = not (x /= y)
  x /= y = not (x == y)

data TrafficLight = Red | Yellow | Green
instance Eq TrafficLight where
  Red == Red = True
  Yellow == Yellow = True
  Green == Green = True
  _ == _ = False
```

```
instance Show TrafficLight where
  show Red = "Red light"
  show Yellow = "Yellow light"
  show Green = "Green light"
```

λ Red == Red

True

λ Red == Yellow

False

λ Red `elem` [Red, Yellow, Green]

True

λ [Red, ...]

[Red light, ...]

```
Class (Eq a) => Num a where
```

(subclass of Eq)

```
    (Eq m) =>
instance Eq (Maybe m) where
  Just x == Just y = x == y
  Nothing == Nothing = True
  _ == _ = False
```

λ :info Num ⇒

↳ type class functions & list of types

λ :info Maybe

λ :info <function> — type declaration

```
class YesNo a where
  yesno :: a -> Bool

instance YesNo Int where
  yesno 0 = False
  yesno _ = True

instance YesNo [a] where
  yesno [] = False
  yesno _  = True

instance YesNo Bool where
  yesno = id

instance YesNo (Maybe a) where
  yesno (Just _) = True
  yesno Nothing  = False

instance YesNo (Tree a) where
  yesno EmptyTree = False
  yesno _         = True


λ yesno $ length []
False
λ yesno "haha"
True
λ yesno ""
False
```

```
λ yesno Just 0
True
λ yesno True
True
λ yesno EmptyTree
False
λ yesno []
False
λ yesno [0,0,0]
True
λ :t yesno
yesno :: (YesNo a) => a -> Bool
```

```
yesNoIf :: (YesNo y) => y -> y -> a -> a
yesNoIf yesNoVal yesResult noResult =
  if yesno yesNoVal then yesResult else noResult
```

```
λ yesNoIf [] "YEAH!" "NO!"
"NO!"
λ yesNoIf (Just 500) ...
"YEAH!"
```

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

instance Functor [] where
  fmap = map
```

```
map :: (a->b) -> [a] -> [b]
λ fmap (*2) [1..3]
[2,4,6]

instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap f Nothing = Nothing

λ fmap (*2) (Just 200)
Just 400
λ fmap (*2) Nothing
Nothing

instance Functor Tree where
  fmap f EmptyTree = EmptyTree
  fmap f (Node x left right)
        = Node (f x) (fmap f left) (fmap f right

instance Functor (Either a) where
  fmap f (Left x) = Left x
  fmap f (Right x) = Right (f x)

λ :K Int
Int ... :: *

λ :K Maybe
... :: * -> *
```

λ :K Maybe Int

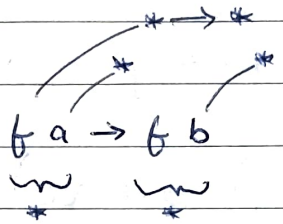
... :: *

λ :K Either

... :: * → * → *

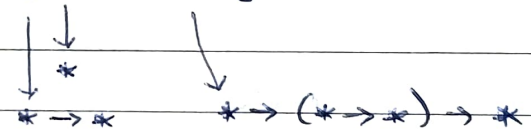
class Functor f where

fmap :: (a → b) → f a → f b

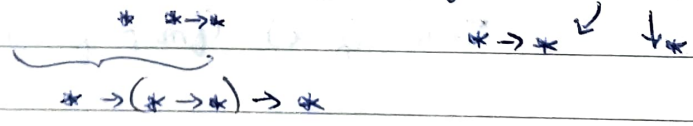


class Tofu t where

tofu :: j a → t a j



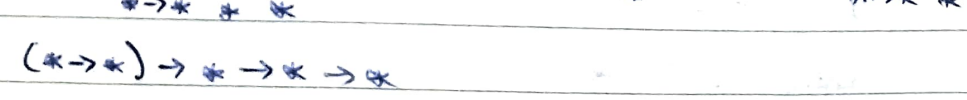
data Frank a b = Frank {field :: b a} deriving (Show)



instance Tofu Frank where

tofu x = Frank x

data Barry t k p = Barry {yabba :: p, dabba :: t k}



instance Functor (Barry a b) where

fmap f (Barry {yabba = x, dabba = y})

= Barry {yabba = f x, dabba = y}