Haskell - P27

a -> Bool

```
instance YesNo (Maybe a) where
    yesno (Just _) = True
    yesno Nothing  = False
yesno :: (YesNo a) => a -> Bool
```

注意在 instance 类型 (Maybe a) 时，对于 constructor Maybe 传入的任何 type variable a 都不影响结果，但仍必须写为 (Maybe a)，因为函数中要求 concrete type

注意在 yesno 的类型中出现了 typeclass constraint

Functor

a type class basically for things can be mapped over, 支持函数 fmap

```
class Functor f where
```

注意 f 不是 concrete type 而是一个 constructor

```
  fmap : (a->b) -> (f a) -> (f b)
```

fmap 传入一个函数 (a->b) 和 (f a)，返回 (f b)

注意在定义 Functor 的 member 时，应该使用 constructor，如 list 的 constructor []

```
instance Functor [] where
  fmap = map
```

可见 fmap 的逻辑与 map 是类似的，可以将 map 视为 fmap 的一个特例 (用于 list)

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

注意这个 constructor 指的是 type 总括性的，而非细分的，如果有多个 constructor，必须分别定义，而且必须注意不同的 constructor 下如果存在不同类型的数值

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

Functor 可以便利地应用在如 Tree，Map，之类的嵌套的 Type 上，如 k-Tree

```
data Tree a = EmptyTree | Node a [Tree a]
  deriving (Show)

instance Functor Tree where
  fmap _ EmptyTree = EmptyTree
  fmap f (Node x ts) =
    Node (f x) (map (fmap f) ts)
```

有如 fmap (+1) (Node 1 [EmptyTree, (Node 1 [])])

→ Node 2 [EmptyTree, (Node 2 [])]

Tree a 可改为 EmptyTree | Node a (Tree a) [Tree a]

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

注意在对如 Either 这样引入多个 type variable 的 type 只能对最后一个类型使用 fmap，即将前边整体作为 constructor 的 partially applied func

如 (Either a)

```
instance Functor (Map.Map k) where
  fmap f fromList [] = fromList []
  fmap f fromList ms =
    let fs = \(k,v) -> (k,(f v))
    in fromList (map fs ms)
```

考虑将 Map 用于 fmap 函数，可能需要定义一个 (k,v) → (k,v') 的函数并将其用于 map [(k,v)]

Haskell - P28

type constructor 用于使用传入的 concrete type 来生成 type，与 constructor 函数的作用类似
如 Maybe 是 type constructor，Maybe Int 是传入 Int 所形成的 concrete type

kind 在 Haskell 中，kind 可以理解为 type 的 type，可以用 :k 命令查看
如 :k Maybe → * -> *，:k Either → * -> * -> *
注意 * 表示 concrete type，即不传入任何 type parameters 以及只能具有 concrete type 的 values

curried 注意 type constructor 也支持 partially applied 的方式，使用方式与 partially applied function 类似
如 :k Either → * -> * -> *，而 :k (Either Int) → * -> *，因为 Int 已作为一个 parameter 传入

Constraint 注意 :k 可作用于 typeclass，可以用于查询 typeclass 要求一个具有何种形式的 kind
如 :k Num → * -> Constraint，:k Functor → (* -> *) -> Constraint
可见 Functor 要求其 member 必须有 kind 形如 (* -> *)
另外 constraint 表示这种形式用于函数声明的 type constraint 部分，传入一个 type parameter 生成一个关于这个 type 的 Constraint

type-foo

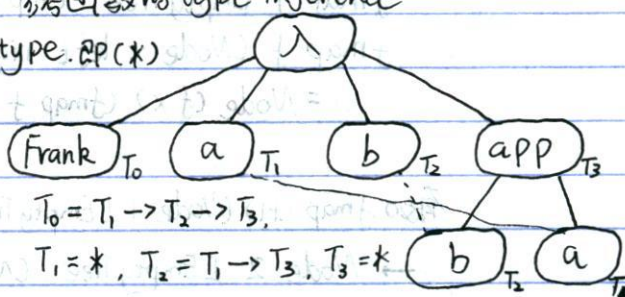
```
class Tofu t where
  tofu :: j a -> t a j

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

Frank 两部分必分别是 concrete type，即 (*)

注意左右的 Frank 是不同的东西，
左侧是 type constructor，有 :k Frank → * -> (* -> *) -> *
右侧是 constructor function，有 :t Frank → (b a) -> Frank a b
如 :t Frank {frankField = Just "hello"} → Frank [Char] Maybe
:t Frank {frankField = "HAHA"} → Frank Char []

有定义 instance Tofu Frank where
  tofu x = Frank a

首先有 :k Tofu → (* -> (* -> *) -> *) -> Constraint
等号函数的 type inference



$T_0 = T_1 -> T_2 -> T_3$,
$T_1 = *$, $T_2 = T_1 -> T_3$, $T_3 = *$
∴ 有 $T_0 = * -> (* -> *) -> *$

有 :t tofu → (Tofu t) => j a -> t a j
则 tofu (Just 'a') :: Frank Char Maybe
→ Frank {frankField = Just 'a'}

如果要将 data Barry t k p = Barry {yappa :: p, dabba :: t k}
声明为 Functor 的 member，注意 Functor :: (* -> *) -> Constraint
应有 instance Functor (Barry t k) where
  fmap :: (a -> b) -> (Barry t k a) -> (Barry t k b)
  fmap f (Barry {yappa = x, dappy = y}) = Barry {yappa = f x, dappa = y}

注意 :k Barry → (* -> *) -> * -> *
利用 partially applied 有 (Barry (* -> *) *) :: * -> *

Haskell - P29

后缀记法

又称逆波兰记法(reverse Polish notation, RPN), 如: 10 4 3 + 2 * - → 10-(4+3)*2

即有 <expr> ::= <num> | <expr> <expr> <operator>. 假定<operator>都是二元运算符

```
solverRPN :: (Integral a, Read a) => String -> a ] Integral支持`div`和`mod`, Read支持read
solverRPN = head . foldl func [] . words         ] 直接将函数见试给solverRPN, 省略参数
  where func (x:y:xs) "+" = (y+x):xs             ] 计算部分, 根据不同的运算符进行不同计算
        func (x:y:xs) "-" = (y-x):xs             ] 从堆栈中弹出两个数字用于计算(pop)
        func (x:y:xs) "*" = (y*x):xs             ] 注意堆栈采用 Last In First Out (LIFO)
        func (x:y:xs) "/" = (y `div` x):xs       ]   即如弹出 x,y 应该是y在前, 即 y-x
        func (x:y:xs) "%" = (y `mod` x):xs       ]
        func xs       num = (read num):xs        ] 将下一个数字压入堆栈 (Stack.push)
```

注意: 计算器可以在 func 的 pattern matching 部分进行扩展以支持更多运算,

如 一元运算符 ln: func (x:xs) "ln" = (log x):xs

或 多元运算符 sum: func xs "sum" = [sum xs]

但要特别注意不同运算对于类型的限制, 保持类型兼容

groupsOf

用于将传入的list中的元素按n个一组的方式进行分组.

```
groupsOf :: Int -> [a] -> [[a]]
groupsOf 0 _ = undefined          注意此处 undefined 函数相当于占位用的报错函数
groupsOf _ [] = []
groupsOf n lst = (take n lst) : (groupsOf n (drop n lst))
```

如 groupsOf 2 [1..11] → [[1,2],[3,4],[5,6],[7,8],[9,10],[11]]

polymorphism

在Haskell中, purity(纯度), higher order function(高阶函数), parameterized data types algebraic 以及 typeclass 允许实现 polymorphism(多态)

无需考虑使 type 属于某个更大的 type hierarchy, 而是考虑 type 的功能并关联到合适的typeclass

Functor

在 fmap :: (a->b) -> (f a) -> (f b) 中, f 常被比喻为 box, 即 a box with a type

但是 box 并不是一个非常准确的描述

更准确的描述是 computational context (可计算的上下文)

context 的 computation 可能含有 a value, failed (如 Maybe), 或 more values (list)

注意于如须为 形如 * -> * 的 kind, 所以在使用时应注意.

如 fmap :: (b->c) -> (Either a b) → (Either a c). 注意其中 Either a 是 fixed

而 fmap :: (b->c) -> (Either b) -> (Either c) 是无效的

Haskell - P30

Functor IO

在Haskell中，do type IO String 表示这是一个I/O action，这个动作会进入real world并取回一个String

在do语法中，可以使用←将结果绑定到一个变量名，

而IO同样可以成为Functor的成员

即在result返回前将其传入函数f

并将函数结果返回

```
instance Functor IO where
  fmap f action = do
    result ← action
    return (f result)
```

注意：return is a function makes an I/O action that doesn't do anything but only presents something as its result

do block procedures will always have the result value of its last action

所以在这种情况下，fmap实际上是生成3一个新的I/O action

即 fmap :: (a -> b) -> IO a -> IO b

Functor (->r)

在Haskell中，(->)运算符的意义与入门时算中所用符号↦对应，

:k (->) → (TYPE q) → (TYPE r) → *，即传入两个TYPE q和r，得到一个concrete type

另外 (->)运算符中中缀运算符，即 :i (->) → infixr 0 `(->)`，且为右结合的

(-> r)实际表示的是 ((->) r)，即 r->，此处r为传入的第一个参数

((->) r)也可以为Functor的instance

```
instance Functor ((->) r) where
  fmap f g = f . g
```

即有 fmap :: (a->b) -> (r->a) -> (r->b)

所以也可以写为简略的形式

即fmap的作用与复合函数一致

```
fmap = (.)
```

:t fmap (*2) → fmap (*2) :: (Functor f, Num b) => (f b) -> (f b)

有 fmap (* 2) (Just 1) → (Just 2)

:t fmap (replicate 3) → fmap (replicate 3) :: (Functor f) => (f a) -> (f [a])

fmap (replicate 3) [1,2,3] → [[1,1,1],[2,2,2],[3,3,3]]

注意：fmap可以作用于传入多个参数的函数，即将函数视为 a -> r 的形式

如 :t fmap (^) → fmap (^) :: (Functor f, Integral b, Num a) => (f a) -> (f (b->a))

即有 fmap (^) [1,2,3] → [(1^), (2^), (3^)]，(!! 2) 2 → 9

如 groupsOf :: Int -> [a] -> [[a]]

有 :t fmap groupsOf → fmap groupsOf :: (Functor f) => (f Int) -> (f ([a] -> [[a]]))

<*> :: (Applicative f) => (f (a->b)) -> (f a) -> (f b)

运算符<*>用于将第一个参数中的函数，作用于第二个参数中的内容

如 (fmap (^) [1,2,3]) <*> [1,2,3] → [1, 1, 1, 2, 4, 8, 3, 9, 27]