Haskell - P45

ZipList

在 Control.Applicative 中提供的 Applicative functor

```
instance Applicative ZipList where
    pure x = ZipList (repeat x)
    ZipList fs <*> ZipList xs = ZipList (zipWith (\f x -> f x) fs xs)
```

在 Applicative ZipList 的语境下：

pure 实现为将传入的参数 x 生成一个无限长的 list，并装入 ZipList

即如 pure "haha" ⟶ ZipList (["haha", "haha", ..])

<*> 实现为将 xs 中的第 k 个元素传入 fs 中的第 k 个元素

其中 zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

```
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = (f x y) : (zipWith f xs ys)
```

即 zipWith 返回结果的长度等于 fs 与 xs 中长度较短的一个

另外 type ZipList a 并没有实现 show instance

所以提供 getZipList 函用于从 ZipList 中提取 list

即 getZipList :: ZipList a -> [a]

如 getZipList $ max <$> ZipList [1,2,3,4,5,6] <*> ZipList [5,3,1,2]

⟶ [5,3,3,4]

getZipList $ (++) <$> (pure "haha") <*> ZipList [".","?","!"]

⟶ ["haha.", "haha?", "haha!"]

在 Control.Applicative 中提供 3 liftA2 函数

```
liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b = f <$> a <*> b
```

注意第一行中的 f 为一个 Applicative functor

第二行中的 f 为类型为 (a -> b -> c) 的函数

而 a, b 分别为类型为 f a, f b 的传入参数

如 fmap (\x -> [x]) (Just 4) ⟶ Just [4]

liftA2 (:) (Just 3) (Just [4]) ⟶ Just [3,4]

(:) <$> (Just 3) <*> (Just [4]) ⟶ Just [3,4]

Haskell - P46

sequenceA

在Haskell中提供了函数 sequenceA，
take list of applicative and return applicative has list as result value
即定义为 sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA [] = pure []
sequenceA (x:xs) = (:) <$> x <*> sequenceA xs
则对于 sequenceA [Just 1, Just 2]
→ (:) <$> Just 1 <*> sequenceA [Just 2]
→ (:) <$> Just 1 <*> ((:) <$> Just 2 <*> sequenceA [])
→ (:) <$> Just 1 <*> ((:) <$> Just 2 <*> Just [])
→ (:) <$> Just 1 <*> Just [2]
→ Just [1,2]
于是实现了从 [Maybe Int] 到 Maybe [Int] 的转换

还可以定义为使用 foldr 函数和 liftA2 函数的形式
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA = foldr (liftA2 (:)) (pure [])
其中 foldr :: (Foldable t) => (a->b->b) -> b -> t a -> b

如对于从 [Maybe a] -> Maybe [a]
有 sequenceA [Just 3, Nothing, Just 1] → Nothing
由于在 instance Functor Maybe 的实现中
fmap f Nothing = Nothing
即 f <$> Nothing → Nothing
而在 instance Applicative Maybe 的实现中
Nothing <*> _ = Nothing
于是 [Maybe a] 中只要存在 Nothing，则 sequenceA 的结果即为 Nothing

如对于从 [(-> r) a] → (-> r) [a]
有 sequenceA [(+3),(+2),(+1)] 3 → [6,5,4]
即 sequenceA [(+3),(+2),(+1)] 实际上返回3一个函数
(Num a) => [a->a] → (Num a) => a -> [a]
将传入的参数传递给 list 中的每个函数，返回的 list 对应地保存着每个结果
等价于 \x -> map (\f -> f x) [(+3),(+2),(+1)]

Haskell - P47

sequenceA

sequenceA 还可以与 and 函数结合使用，以及 or 函数

其中 and :: (Foldable t) => t Bool -> Bool

or :: (Foldable t) => t Bool -> Bool

如 and $ sequenceA [(>4), (<12), even] 11 ⟶ False

or $ sequenceA [(>4), (<12), even] 11 ⟶ True

当 sequenceA 作用于 [[a]] 时，会返回一个 [[a]]

且具体的行为与列表表达式 (list comprehension) 类似

如 sequenceA [[1,2],[3,4]] ⟶ [[1,3],[1,4],[2,3],[2,4]]

[[x,y] | x<-[1,2], y<-[3,4]] ⟶ [[1,3],[1,4],[2,3],[2,4]]

sequenceA [[1,2],[3,4],[5,6]]

⟶ [[1,3,5],[1,3,6],[1,4,5],[1,4,6],[2,3,5],[2,3,6],[2,4,5],[2,4,6]]

其中具体的运算步骤为

对于 sequenceA [[1,2],[3,4]]

⟶ (:) <$> [1,2] <*> sequenceA [[3,4]]

⟶ (:) <$> [1,2] <*> ((:) <$> [3,4] <*> sequenceA [[]])

⟶ (:) <$> [1,2] <*> ((:) <$> [3,4] <*> [[]])

又 (:) <$> [3,4] 等价于 fmap (:) [3,4]

⟶ (:) <$> [1,2] <*> ([(3:), (4:)] <*> [[]])

⟶ [(1:), (2:)] <*> [[3], [4]]

⟶ [[1,3], [1,4], [2,3], [2,4]]

可以一般地描述为 sequenceA [$L_1, L_2, \cdots, L_n$]，其中 $L_1, \cdots, L_n$ :: [a]

⟶ [[$l_1, \cdots, l_n$] | $l_1 \leftarrow L_1, \cdots, l_n \leftarrow L_n$]，类似于 $\{\langle l_1, l_2, \cdots, l_n\rangle \in L_1 \times L_2 \times \cdots \times L_n\}$

当 sequenceA 作用于 [IO a] 时，返回 IO [a]

如 sequenceA [getLine, getLine] ⟶ hello

world

⟶ ["hello", "world"]

与 Functor 类似，Applicative functor 也服从 functor law，除了基础的 functor laws

```
pure id <*> v = v
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
pure f <*> pure x = pure (f x)
u <*> pure y = pure ($ y) <*> u
```

Haskell - P48

newtype

在Haskell中，提供3 newtype 关键字用于 type wrapping

如 对于类型 Ziplist a，实现为 Applicative Functor

实质上是将一个[a]包装进 Ziplist 的一个字段(field)

于是有 data Ziplist a = Ziplist [a]

one value constructor Ziplist, one field

use record syntax to automatically get one function

that extract list from Ziplist

即有 data Ziplist a = Ziplist {getZiplist :: [a]}

注意 getZiplist 即指定3 Ziplist 中的字段名

同时也是一个函数 getZiplist :: Ziplist a -> [a]

这样即通过 data 关键字 wrap one type into another type

to make the other type an instance of one type class in another way

newtype关键字可以得到同样的效果

如 newtype Ziplist a = Ziplist {getZiplist :: [a]}

在Haskell中，newtype关键字 take just one type and wrap in sth to present as another way

与data关键字相比：

newtype 关键字的编译执行速度更快

when using data keyword, there's some overhead

to all wrapping / unwrapping when program running

when using newtype keyword,

Haskell know wrap one existing type into new type

to be same internally but have one different type

can get rid of wrapping / unwrapping one resolve which value of which type

但是 newtype 关键字的使用条件更加严格

can only have one value constructor

value constructor can have only one field

而 data 关键字允许有多个 value constructor 且 constructor 允许有 zero or more field

如 data Profession = Fighter | Archer | Accountant

data Race = Human | Elf | Orc | Goblin

data PlayerCharacter = PlayerCharacter Race Profession

另外 newtype 关键字可以与 deriving 关键字一同使用

derive instance for Eq / Ord / Enum / Bounded / Show / Read

如 newtype CharList = CharList {getCharList :: [Char]} deriving (Eq, Show)

Haskell - P49

newtype

在Haskell中，当fmap应用于二元tuple时，
实现为将传入的函数作用在第二个元素上，再返回一个二元tuple
如 fmap (+100) (1, 2) → (1, 102)
则考虑使用newtype实现 将传入的函数参数作用于第一个元素
newtype Pair b a = Pair { getPair (a, b)}
deriving (Eq, Show)
instance Functor (Pair b) where
fmap f (Pair (x, y)) = Pair (f x, y)
注意这里是将 (Pair b) 实现为 Functor 的 instance
其中 :k (Pair Int) :: * -> *
而 :k Pair :: * -> * -> *
则有 getPair $ (+100) <$> (Pair (1, 2)) → (101, 2)

newtype not only faster, but also lazier
any computation take place only when try to actually print result of function
only computation necessary for function resule will get carried out

在Haskell中提供3特殊值 undefined 来表示错误计算 (erronous computation)
当尝试 evaluate undefined 时，会抛出异常
如 ghci > undefined → *** Exception: Prelude.undefined
而如果不需 evaluate undefined，仅仅出现并不会抛出异常
如 head [3, 4, 5, undefined] → 3
注意 undefined 可视为任意类型，即 undefined :: a
且有 head [("a", 1), (undefined, 2), ("c", undefined)] → ("a", 1)

定义 data CoolBool = CoolBool { getCoolBool :: Bool }
helloMe :: CoolBool -> String
helloMe (CoolBool _) = "hello"
有 helloMe undefined
→ *** Exception :: Prelude.undefined
由于 data 关键字支持 multipde value constructor，
于是 Haskell 会尝试计算传入参数的值以确认是否匹配 (CoolBool _) pattern
而由于 evaluate undefined 会抛出异常，
所以 helloMe undefined 的结果为抛出异常

Haskell - P50

newtype

如果定义 newtype CoolBool = CoolBool {getCoolBool :: Bool}

而使用同样的函数定义 helloMe :: CoolBool -> String

helloMe (CoolBool _) = "hello"

则 helloMe undefined ⟶ "hello"

Haskell can internally represent value of new type same way as original value

not have to add another box, just have to be aware of value being different type

know type made with newtype keyword can only have one constructor

not have to evaluate value passed to the function

to make sure conform to (CoolBool _) pattern

because newtype type can only have one possible value constructor and one field

difference between keywords type / newtype / data

type keyword: make type synonym to easier to refer to

just give another name to already existing type

如 type IntList = [Int]

([1,2,3] :: IntList) ++ ([4,5,6] :: [Int])

⟶ [1,2,3,4,5,6]

make type signature more descriptive by giving type name

to tell about purpose in context of function

newtype keyword: take existing type and wrap in new type

easier to make it instance of certain type class

new type is separate from original type

如 newtype CharList = CharList {getCharList :: [Char]}

不可使用 ++ 来连接 CharList 与 [Char] / CharList

new type not automatically made instance of type class for original

data keyword: make own data type, to implement any algebraic data type

with any number of value constructors */ fields

data keyword used to make own type from scratch

newtype keyword for making completely new type out of existing type

Haskell - P51

在Haskell中，对于如函数 (*) :: (Num a) => a -> a -> a
或 (++) :: [a] -> [a] -> [a]
有一系列相似的特征和行为：
传入两个参数并返回一个返回值，即有 a -> b -> c
传入参数与返回值的类型均相同，即有 a -> a -> a
存在一个特殊值，当作为参数传入二元函数时，不会改变另一个参数，即返回另一个参数
如 11 * 1 → 11 , [1,2,3] ++ [] → [1,2,3]
满足结合律(associativity)，
如 (3 * 4) * 5 = 3 * (4 * 5) , ("ha" ++ "he") ++ "hi" = "ha" ++ ("he" ++ "hi")

monoid

在Haskell中，满足以上条件的二元函数及其特殊值
associative binary function and identity value with repect to function
在Data.Monoid中提供了type class Monoid

```
class Monoid m where
  mempty  :: m                  ] identity value
  mappend :: m -> m -> m        ] associative binary function
  mconcat :: [m] -> m           ] 默认的mconcat实现
  mconcat = foldr mappend mempty ]
```

注意在Monoid中，m为concrete type
而在Functor和Applicative中，f为type constructor taking one type parameter
mempty：注意这实际上并不是一个函数，因为调用并不传入参数
而仅是一个多态常量(polymorphic constant)
represent identity value for particalar monoid
mappend: represent associative binary function
take two monoid values and return one monoid value
mconcat: take list of monoid value and reduce to single value by mappend
Data.Monoid中提供了一种默认的实现
take mempty as starting monoid value
and fold list of monoid from the right with mappend
通常默认实现已经可以满足大多数 monoid
所以在实现instance时，只定义 mempty 和 mappend 即足可
而对于一些特殊的instance，可能存在更有效的 mconcat 实现

Haskell - P52

Monoid

通常而言，monoid满足 associative binary function and identity value respect to function

但是在 Haskell 中，对于提供的 Data.Monoid type class

可以构造 Monoid 的 instance，使得不满足 monoid 的要求

如 new NonMonoidList a = NonMonoidList {getNonMonoidList :: [a]}

```
    deriving (Eq, Show)
instance Monoid (NonMonoidList a) where
    mempty = NonMonoidList []
    mappend lst1 lst2 =
        let l1 = getNonMonoidList lst1
            l2 = getNonMonoidList lst2
        in NonMonoidList (l1 ++ (reverse l2))
```

于是有 ((NonMonoidList [1,2]) `mappend` (NonMonoidList [3,4])) `mappend` (NonMonoidList [5,6])

⟶ NonMonoidList [1,2,4,3,6,5]

(NonMonoidList [1,2]) `mappend` ((NonMonoidList [3,4]) `mappend` (NonMonoid [5,6]))

⟶ NonMonoidList [1,2,5,6,4,3]

但是注意在 Haskell 中，还需提前定义 instance Semigroup (NonMonoidList a)

于是在定义 instance Monoid a 时，需要遵循以下规则：

```
mempty `mappend` x = x                                  ] identity value
x `mappend` mempty = x
(x `mappend` y) `mappend` z = x `mappend` (y `mapped` z)] associative
```

如对于 Haskell 中内置的 List 类型

```
instance Monoid [a] where
    mempty = []
    mappend = ++
    mconcat xss = [x | xs <- xss, x <- xs]
```

注意 instance Monoid [a] 中的 [a] 是 concrete type

instance Functor [] 中的 [] 是 type constructor

另外 mconcat 的实现是 list comprehension and inlining

而非默认实现中的 foldr

Haskell - P53

Monoid

在Haskell中，Data.Monoid 模块中提供了乘法和加法的monoid

有 newtype Product a = Product {getProduct :: a}
    deriving (Eq, Ord, Read, Show, Bounded)
instance (Num a) => Monoid (Product a) where
    mempty = Product 1
    Product x `mappend` Product y = Product (x * y)

newtype Sum a = [redacted] {getSum :: a}
    deriving (Eq, Ord, Read, Show, Bounded)
instance (Num a) => Monoid ([redacted] a) where
    mempty = Sum 0
    Sum x `mappend` Sum y = Sum (x + y)

注意在定义 instance Monoid 时，对于 Product a 和 Sum a
添加了 class constraint (Num a)
    mean Product a / Sum a is instance of Monoid
        for all a already instance of Num

在Haskell中，Data.Monoid 模块中还提供了针对 Bool 的逻辑与/逻辑或的 monoid

有 newtype Any = Any {getAny :: Bool}
    deriving (Eq, Ord, Read, Show, Bounded)
instance Monoid Any where
    mempty = Any False
    Any x `mappend` Any y = Any (x || y)

newtype All = All {getAll :: Bool}
    deriving (Eq, Ord, Read, Show, Bounded)
instance Monoid All where
    mempty = All True
    All x `mappend` All y = All (x && y)

注意在定义 instance Monoid 并没有加入 class constraint
由于 Any / All 已经是 concrete type，而 Product / Sum 是 type constructor

Haskell - P54

Monoid

在Haskell中，提供3 Ordering类及其 instance Monoid 实现

```
data Ordering = LT | EQ | GT
instance Monoid Ordering where
    mempty = EQ
    LT `mappend` _ = LT
    EQ `mappend` x = x
    GT `mappend` _ = GT
```

mappend 作用于 Ordering 的机制为

当第一个传入参数为 LT / GT 时，忽略第二个传入参数，返回 LT/GT

当第一个传入参数为 EQ 时，返回第二个传入参数

如 mempty `append` LT ⟶ LT

LT `append` GT ⟶ GT

考虑字符串的比较，含比较条件为字符串长度 > 字典序

即先按长度比较，长度相等时比较字典序

```
提有 lengthCompare :: String -> String -> Ordering
lengthCompare x y =
    let a = length x `Compare` length y
        b = x `compare` y
    in if a== EQ then b else a
```

另外可以使用 Monoid 的版本

```
lengthCompare' :: String -> String -> Ordering
lengthCompare' x y = (length x `compare` length y)
                     `mappend` (x `compare` y)
```

再考虑先按长度比较，再按元音个数比较，最后按字典序比较

```
lengthCompare'' :: String -> String -> Ordering
lengthCompare'' x y = (length x `compare` length y) `mappend`
                      (vowels x `compare` vowels x) `mappend`
                      (x `compare` y)
    where vowels = length . filter (`elem` "aeiou")
```

则有 lengthCompare'' "zen" "anna" ⟶ LT

lengthCompare'' "zen" "ana" ⟶ LT

Haskell - P55

Monoid

在Haskell中，提供了 Maybe 的 instance Monoid (Maybe a) 的实现

```
instance Monoid a => Monoid (Maybe a) where
  mempty = Nothing
  Nothing `mappend` m = m
  m `append` Nothing = m
  Just m1 `mappend` Just m2 = Just (m1 `mappend` m2)
```

注意在这个实现中，有对于 a 的 constraint，(Monoid a)

即 Maybe a is instance of Monoid only if a is instance of Monoid

于是当 a 不是 Monoid 的 instance 时，无法使用 Monoid (Maybe a)

```
ex: Nothing `mappend` Just LT --> Just LT
  Just (Sum 3) `mappend` Just (Sum 4)
    --> Just (Sum {getSum = 7})
```

于是 Data.Monoid 中提供了 First / Last 类来处理 Maybe a 中 a 不是 instance of Monoid 的情形

```
newtype First a = First {getFirst :: Maybe a}
  deriving (Eq, Ord, Read, Show)
instance Monoid (First a) where
  mempty = First Nothing
  First (Just x) `mappend` _ = First (Just x)
  First Nothing `mappend` x = x

newtype Last a = Last {getLast :: Maybe a}
  deriving (Eq, Ord, Read, Show)
instance Monoid (Last a) where
  mempty = Last Nothing
  _ `mappend` Last (Just x) = Last (Just x)
  x `mappend` Last Nothing = x
```

在 mconcat 过程中，保留 first / last non-Nothing value

```
ex: getFirst . mconcat . map First $ [Just "hello", Nothing, Just "world"]
    --> Just "hello"
  getLast . mconcat . map Last $ [Just "hello", Nothing, Just "world"]
    --> Just "world"
```

Haskell - P56

Foldable

在Haskell中，提供3 Data.Foldable type class

除了对于常见的可折叠的类型，如 list，还可作用于其他类型

如 foldl (+) 2 (Just 9) → 11

foldr (||) False (Just True) → True

Foldable 中支持函数 foldMap

即 class Foldable t where

```
foldMap :: (Monoid m) => (a -> m) -> t a -> m
foldl :: (b -> a -> b) -> b -> t a -> b
foldr :: (a -> b -> b) -> b -> t a -> b
```

则定义 data Tree a = Empty | Node a (Tree a) (Tree a)] 定义类型表示二叉树

deriving (Show, Read, Eq)

```
instance Functor Tree where
  fmap _ Empty = Empty
  fmap f (Node x l r) =
    Node (f x) (fmap f l) (fmap f r)
```

] instance of Functor 实现

提有 instance Foldable Tree where

```
foldMap f Empty = mempty
foldMap f (Node x l r) =
  (foldMap f l) `mappend` (f x) `mappend` (foldMap f r)
```

注意传入的参数函数 f :: a -> m

即将一个树中的元素转换为一个 Monoid m 的值

且在 foldMap f Empty = mempty 中的 f 不应忽略

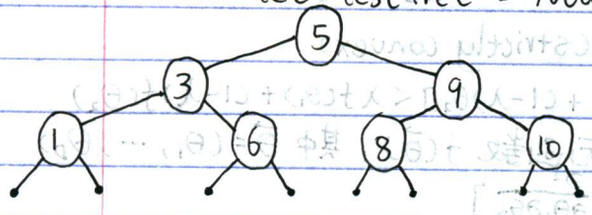
由其类型 a -> m 决定返回的 mempty 的值

如 let testTree = Node 5 (Node 3 (Node 1 Empty Empty)

(Node 6 Empty Empty))

(Node 9 (Node 8 Empty Empty)

(Node 10 Empty Empty))



foldl (+) 0 testTree → 42 (求和)

getAny $ foldMap (\x -> Any $ x == 3) testTree → True (查找是否存在元素3)

getAny $ foldMap (\x -> Any $ x > 15) testTree → False (检查是否有元素大于15)