Haskell - P57

Monad

natural extension of applicative functor, support >>= (bind function)

(>>=) :: (Monad m) => m a -> (a -> m b) -> m b

apply function of type a -> m b to value of type m a

>>= take one monadic value (m a)

one function take normal value (a) and return monadic value (m b)

and apply function a -> mb to monadic value m a

>>= 作用于 Maybe 类型时，其行为类似于函数

```
applyMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
applyMaybe Nothing _ = Nothing
applyMaybe (Just x) f = f x
```

```
ghci> Just "smile" `applyMaybe` (\x -> Just (x ++ " :)"))
"smile :)"
ghci> Nothing `applyMaybe` (\x -> Just (x + 1))
Nothing
```

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y
    fail :: String -> m a
    fail msg = error msg
```

when Haskell made, not occured to people that applicative functor are good fit

目前 GHC.Base 中的定义：

```
class (Applicative m) => Monad m where
    (>>=) :: forall a b. m a -> (a -> m b) -> m b
    (>>) :: forall a b. m a -> m b -> m b
    m >> k = m >>= \_ -> k
    return :: a -> m a
    return = pure
    fail :: String -> m a
    fail s = errorWithoutStackTrace s
```

Haskell - P58

Monad

```
return :: (Monad m) => a -> m a
```

take a value and put in minimal default context
in other word wrap in a monad that still hold the value
do same thing as "pure" function from Applicative type class
nothing like "return" in most other languages
not end function execution or anything

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

like function application : take normal value and feed to normal function
bind : take monadic value (a value with a context)
and feed to function that take normal value but return monadic value

```
(>>) :: (Monad m) => m a -> m b -> m b
m >> k = m >>= \_ -> k
```

instance come with default implementation and never implement when making Monad

```
fail :: (Monad m) => String -> m a
```

"fail" function never used explicitly in code
used by Haskell to enable failure in special syntactic construct for monad

如对于将 Maybe 实现为 Monad 的 instance

```
instance Monad Maybe where
    return x = Just x
    Nothing >>= f = Nothing
    Just x >>= f = f x
    fail _ = Nothing
```

其中 return 等价于 pure，do same as in Applicative type class
>>= (bind function) 等价于之前定义的函数 apply Maybe

```
do return "what" :: Maybe String ⟶ Just "what"
   Just 11 >>= \x -> return (x * 7) ⟶ Just 77
   Nothing >>= \x -> return (x * 7) ⟶ Nothing
```

Haskell - P59

考虑一个平衡问题，假定一个走钢丝的人手持一根平衡棒（balancing pole）
在其走钢丝的过程中随机地有鸟降落/离开平衡棒的一侧
如果左右两侧停留的鸟的数量相差不超过3只，则可以保持平衡
否则走钢丝的人会失去平衡

首先定义 Birds 为 Int 的别名，Pole 是 (Birds, Birds) 的别名

```
type Birds = Int
type Pole = (Birds, Birds)
type State = Maybe Pole    , State 是 Maybe Pole 的别名
```

再定义函数 landLeft 和 landRight

```
landLeft :: Birds -> Pole -> State
landLeft n (left, right)
  | abs ((left+n) - right) < 4 = Just ((left+n), right)
  | otherwise                  = Nothing
landRight :: Birds -> Pole -> State
landRight n (left, right)
  | abs (left - (right+n)) < 4 = Just (left, (right+n))
  | otherwise                  = Nothing
```

于是可知 landLeft Int 和 landRight Int 有类型 Pole -> Maybe Pole
则可以用于 Monad Maybe 的实现

```
do return (0,0) >>= landRight 2 >>= landLeft 2 >>= landRight 2
   -> Just (2,4)
   return (0,0) >>= landLeft 1 >>= landRight 5 >>= landLeft 4
   -> Nothing
```

注意 Applicative operator will fetch result and feed to function
in manner appropriate for each applicative
and then put final applicative value together

假设当走钢丝的人踩到香蕉皮会立刻失去平衡

```
banana :: Pole -> State
banana _ = Nothing
do return (0,0) >>= landLeft 1 >>= banana >>= landRight 1
   -> Nothing
```

注意 >>= banana 实际上等价于 >> Nothing

```
do return (0,0) >>= landLeft 1 >> Nothing >>= landRight 1 -> Nothing
```

Haskell - P60

do notation 起源 `Just 11 >>= (\x -> Just "?" >>= (\y -> Just (show x ++ y)))`
⟶ `Just "11?"`
注意 从 Just 11 中提取 11 并传给参数 x
从 Just "?" 中提取 "?" 并传给参数 y
value with failure context, can replace with failure
如 `Nothing >>= (\x -> Just "?" >>= (\y -> Just (show x ++ y)))`
⟶ `Nothing`
`Just 11 >>= (\x -> Nothing >>= (\y -> Just (show x ++ y)))`
⟶ `Nothing`

可以用 do notation 代替嵌套的 lambda function

```
foo :: Maybe String
foo =
  Just 11 >>= (\x ->
  Just "?" >>= (\y ->
  Just (show x ++ y)))
```

```
bar :: Maybe String
bar = do
  x <- Just 11
  y <- Just "?"
  Just (show x ++ y)
```

foo ⟶ `Just "11?"`
bar ⟶ `Just "11?"`

temporarily extract thing from Maybe value
without having to check if Maybe value is Just / Nothing at every step
if any of value extract from Nothing, whole do expression result in Nothing
do expression just different syntax for chaining monadic value

every line is monadic value in do expression
use <- to inspect value
如 variable become String if bind with <- Maybe String
just like use >>= to feed monadic value to lambda function

2如 `fool :: Maybe Bool`

```
fool =
  Just 11 >>= (\x ->
  Just (x > 7))
```

fool ⟶ `Just True`

```
bar1 :: Maybe Bool
bar1 = do
  x <- Just 11
  Just (x > 7)
```

bar1 ⟶ `Just True`

Haskell - P61

Monad

考虑将一个过程转换为 do notation

```
routine :: Maybe Pole
routine = case landLeft 2 (0,0) of
    Nothing -> Nothing
    Just first -> case landRight 2 first of
        Nothing -> Nothing
        Just second -> landLeft 1 second
```

有 routine ⟶ Just (3,2)

类似于 use >>= to chain successive step

如 return (0,0) >>= landLeft 2 >>= landRight 2 >>= landLeft 1
⟶ Just (3,2)

each step rely on previous step and have added context of possible failure

可以等价地改写为 do notation 的形式

```
routineDo :: Maybe Pole
routineDo = do
    start <- return (0,0)
    first <- landLeft 2 start
    second <- landRight 2 first
    landLeft 1 second
```

routineDo ⟶ Just (3,2)

each line of do expression must feature monadic value

do notation look like imperative code (指令式编程)
because written line by line

do notation is sequential
each value in each line rely on result of previous ones
along with context, whether succeeded or failed

在 do notation 也支持 pattern match

如

```
justH :: Maybe Char
justH = do
    (x:xs) <- Just "hello"
    return x
```

justH ⟶ Just 'h'

Haskell - P62

Monad

考虑 return (0,0) >>= landLeft 2 >> Nothing >>= landRight 2 >>= landLeft 1
→ Nothing

可以转换成函数形式

```
routineNothing :: Maybe Pole
routineNothing = case landLeft 2 (0,0) of
  Nothing -> Nothing
  Just first -> case banana first of
    Nothing -> Nothing
    Just something -> case landRight 2 something of
      Nothing -> Nothing
      Just second -> landLeft 1 second
```

则有 routineNothing → Nothing

可以进一步地改写成 do notation 的形式

```
routineDoNothing :: Maybe Pole
routineDoNothing = do
  start <- return (0,0)
  first <- landLeft 2 start
  Nothing
  second <- landRight 2 first
  landLeft 1 second
```

equivalent to `_ <- Nothing`

则有 routineDoNothing → Nothing

write line in do notation without binding monadic value with <-
just like put >> after monadic value whose result want to ignore

if matching fall through all patterns for given function
error thrown and program crash

failed pattern matching in let expression result in error produced right away
while in do expression, call fail function
part of Monad type class, enable failed pattern matching result in failure in context

如 justN :: Maybe Char

```
justN = do
  (x:xs) <- Just ""
  return x
```

justN → Nothing

```
fail :: (Monad m) => String -> m a
fail msg = error msg
```

而 instance Monad Maybe 实现中

```
fail _ = Nothing
```

Haskell - P63

Monad

when deal with non-determinism, can make many choices
try all the choices, result non-determinism has many more values

如 Monad type class 对于 list 的实现

```
instance Monad [] where
  return x = [x]
  xs >>= f = concat (map f xs)
  fail _ = []
```

return : do the same thing as pure
make a list has only input one value
useful when wrap normal value into list
so can interact with non-deterministic value

```
>>= :: [a] -> (a -> [b]) -> [b]
```

take all possible values in [a]
feed into function take normal value and return monadic value
即 类型为 a -> [b]
最后通过 concat :: (Foldable t) => t [a] -> [a]
将 [[b]] 组装为 [b]

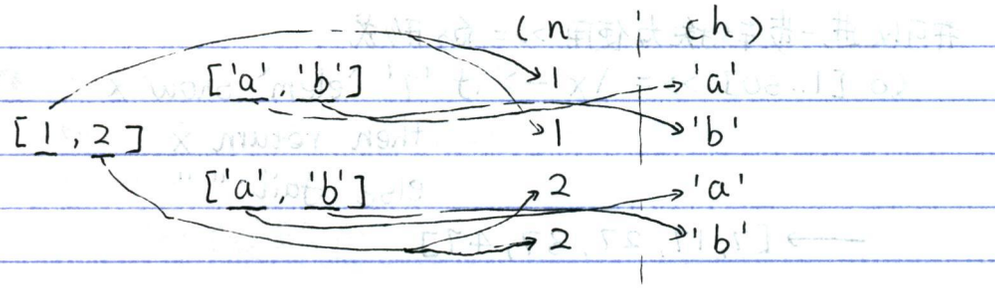
如 `[1,2,3] >>= \x -> [x,-x]`
⟶ [1, -1, 2, -2, 3, -3]

fail : non-determinism also support for failure
empty list [] 类似于 Maybe 中的 Nothing
signify absence of result

如 `[] >>= \x -> "bad"` ⟶ ""
`[1,2,3] >>= \x -> []` ⟶ []

与 Maybe 类似，可以连续地使用 >>=

如 `[1,2] >>= \n -> ['a','b'] >>= \ch -> return (n,ch)`
⟶ [(1,'a'),(1,'b'),(2,'a'),(2,'b')]



Haskell - P64

Monad

对于 [1,2] >>= \n -> ['a','b'] >>= \ch -> return (n,ch)
⟶ [(1,'a'),(1,'b'),(2,'a'),(2,'b')]

转换为 do notation 的形式有

```
listOfTuple :: [(Int, Char)]
listOfTuple = do
  n <- [1,2]
  ch <- ['a','b']
  return (n, ch)
```

listOfTuple ⟶ [(1,'a'),(1,'b'),(2,'a'),(2,'b')]

注意到这个 do notation 类似于列表生成式 (list comprehension)

即有 [(n,ch) | n <- [1,2], ch <- ['a','b']]
⟶ [(1,'a'),(1,'b'),(2,'a'),(2,'b')]

实际上列表生成式是使用 Monad [] 的语法糖
syntactic sugar for using list as monad

对于列表生成式中使用筛选条件 (filter)

如 [x | x <- [1..50], '7' `elem` show x]
⟶ [7,17,27,37,47]

同样可以转换为 do notation 的形式

注意对于不满足筛选条件的情况，可以调用 fail 函数以生成 []

```
listOfSeven :: [Int]
listOfSeven = do
  n <- [1..50]
  if '7' `elem` show n then
    return n
  else
    fail ""
```

listOfSeven ⟶ [7,17,27,37,47]

并可以进一步转换为使用 >>= 的形式

```
如 [1..50] >>= \x -> if '7' `elem` show x
                     then return x
                     else fail ""
```

⟶ [7,17,27,37,47]

Haskell - P65

MonadPlus 在Haskell中提供3 MonadPlus type class

for Monad can also act as Monoid

在GHC.Base中对 type class MonadPlus 的定义为

infixl 3 <|>

```
class (Applicative f) => Alternative f where
    empty :: f a                    : identity of '<|>'
    (<|>) :: f a -> f a -> f a      : associative binary operation
    some :: f a -> f [a]
    some v = some_v
      where many_v = some_v <|> pure []
            some_v = liftA2 (:) v many_v
    many :: f a -> f [a]
    many v = many_v
      where many_v = some_v <|> pure []
            some_v = liftA2 (:) v many_v
```

one or more

zero or more

```
class (Alternative m, Monad m) => MonadPlus m where
    mzero :: m a                    ] default definition of
    mzero = empty                   ]   identity of `mplus`
    mplus :: m a -> m a -> m a      ] default definition of
    mplus = (<|>)                   ]   associative operation
```

其中 mzero synonymous to mempty for Monoid type class

represent non-deterministic computation has no result at all (failed)

mplus synonymous to mappend for Monoid type class

join two non-deterministic values into one

对于 Maybe 类型有 Alternative 的实现为

```
instance Alternative Maybe where        ] instance MonadPlus Maybe
    empty = Nothing                     ]   采用在Alternative中的实现
    Nothing <|> r = r
    l <|> _ = l
```

对于 [] 有Alternative 的实现为

```
instance Alternative [] where           ] instance MonadPlus []
    empty = []                          ]   采用在Alternative中的实现
    (<|>) = (++)
```

Haskell - P66

MonadPlus

在Haskell中，Control.Monad中提供3与Monad相关的类和函数

如 guard :: (MonadPlus m) => Bool -> m ()

guard True = return ()

guard False = mzero

当传入Bool值为True时，将空元组()装入 minimal default context

而传入Bool值为False时，返回 failed monadic value

如 guard (5 > 1) :: Maybe () ⟶ Just ()

guard (5 > 11) :: Maybe () ⟶ Nothing

guard (5 > 1) :: [()] ⟶ [()]

guard (5 > 11) :: [()] ⟶ []

当 guard 与 >> 组合使用时，>> ignore empty tuple () and present other result

如果 guard fail，则其后的操作也会失败

即向 >>= 传入一个failed monadic value 也会返回一个failed monadic value

如果 guard succeed，则 make successful value has dummy result of () inside

all allow computation to continue

如 guard (5 > 1) >> return "TRUE" :: [String] ⟶ ["TRUE"]

guard (5 > 11) >> return "TRUE" :: [String] ⟶ []

再考虑列表生成式中使用筛选条件，可以使用guard进行实现

如 [x | x <- [1..50], '7' `elem` show x] ⟶ [7,17,27,37,47]

⟶ [7,17,27,37,47]

[1..50] >>= (\x -> guard ('7' `elem` show x) >> return x)

可以进一步地用guard实现 do notation 的形式

如 sevenOnly :: [Int]

sevenOnly = do

x <- [1..50]

guard ('7' `elem` show x)

return x

sevenOnly ⟶ [7,17,27,37,47]

注意在这个过程中，由于guard函数支抛出的failed monadic value 阻断计算

所以实际上相当于一个 if-then-eles 的结构

并在失败时调用 fail 函数

Haskell - P67

Knight's quest，考虑一个国际象棋棋盘(chess board)且仅有一个骑士(knight)

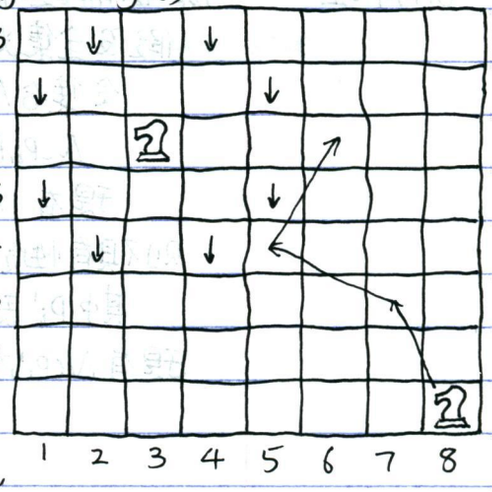
对于给定 knight 的起始位置和一个目标位置

考虑是否可以用恰好3次移动到达

令 type KnightPos = (Int, Int)

其中第一个元素表示 knight 所在的列

第二个元素表示 knight 所在的行



可以通常地定义一个使用 filter 的函数

```
moveKnight :: KnightPos -> [KnightPos]
moveKnight (c, r) = filter onBoard
    [(c+2, r+1), (c+1, r+2), (c-1, r+2), (c-2, r+1),
     (c-2, r-1), (c-1, r-2), (c+1, r-2), (c+2, r-1)]
    where onBoard (c', r') = c' `elem` [1..8] && r' `elem` [1..8]
```

其中 onBoard 函数用于保证输出位置均在棋盘内部

则 moveKnight 也可以转换为等价的 do notation 形式

```
moveKnight :: KnightPos -> [KnightPos]
moveKnight (c, r) = do
    (c', r') <- [(c+2, r+1), (c+1, r+2), (c-1, r+2), (c-2, r+1),
                 (c-2, r-1), (c-1, r-2), (c+1, r-2), (c+2, r-1)]
    guard (c' `elem` [1..8] && r' `elem` [1..8])
    return (c', r')
```

如 moveKnight (7,7) ⟶ [(5,8), (5,6), (6,5), (8,5)]

可以定义函数 at3 以返回对给定起始位置可以用恰好3次移动到达的位置

```
at3 :: KnightPos -> [KnightPos]
at3 start = do
    first <- moveKnight start
    second <- moveKnight first
    moveKnight second
```

等价于 at3 start = return start >>= moveKnight >>= moveKnight >>= moveKnight

再定义函数 canReachAt3 用于判断

```
canReachAt3 :: KnightPos -> KnightPos -> Bool
canReachAt3 start end = end `elem` at3 start
```

如 (8,1) `canReachAt3` (7,7) ⟶ True

Haskell - P68

Monad

与functor类似，Monad也需要符合一些规则，即monad law

注意在Haskell中，对于Monad type class的instance并不要求满足monad law

所以需要在实现instance Monad的时候特别注意

left identity: return x >>= f 等价于 f x

feed monadic value to function f :: (Monad m) => a -> m b

not different than apply function to normal value

如 return "wow" >>= (\x -> [x, x, x])

⟶ ["wow", "wow", "wow"]

(\x -> [x, x, x]) "wow"

⟶ ["wow", "wow", "wow"]

right identity: m >>= return 等价于 m

注意 return :: (Monad m) => a -> m a

如对于 instance Monad [] 中对 >>= 的实现

有 xs >>= f = concat (map f xs)

于是 [1,2,3] >>= return ⟶ concat [[1],[2],[3]] ⟶ [1,2,3]

associativity: (m >>= f) >>= g 等价于 m >>= (\x -> f x >>= g)

chain of monadic function application with >>= not matter how nested

类似于函数复合运算符 (.) :: (b -> c) -> (a -> b) -> (a -> c)

f . g = (\x -> f (g x))

有 (<=<) :: (Monad m) => (b -> m c) -> (a -> m b) -> (a -> m c)

f <=< g = (\x -> g x >>= f)

如 let f x = [x, -x]

let g x = [x*3, x*2]

let h = f <=< g

h 1 ⟶ [3, -3, 2, -2]

结合Monad law 则有：对于任意 monadic function f :: (Monad m) => a -> m b

有 f <=< return 等价于 f

return <=< f 等价于 f

(f <=< g) <=< h 等价于 f <=< (g <=< h)

其中 g :: (Monad m) => c -> m a , h :: (Monad m) => d -> m c