Haskell Notes

基本概念部分

```
{- Zh 基本概念解释 -}{- 之后根据具体内容进行monad和do等方式的书写 -}
-- 数据 数据是各种领域中的对象或信息中抽象出来的表示对应属性的数值化符号表示。例如：二进制文本数据，体温，GDP等。

-- 程序 计算机程序是通过指令的顺序，使计算机能按所要求的功能进行精确执行的指令集。而在haskell中程序是输入数据类型到输出数据类型的函数。例如：应用软件，操作系统。

-- 计算机程序设计 为了完成一个特定计算任务而设计和编写计算机可执行程序的过程

-- 算法 算法是任何良定义的计算过程，该过程提取某个值或集合作为输入，根据表达式(expression)进行转化形成某个值或集合的输出的计算步骤的一个序列。例如：Strassen算法，Bellman-Ford算法，Dijkstra算法，或者就像Haskell中定义的函数(function)即是算法的一种缩影。

-- 数据类型 (data type) 类型是一系列同类(sort)的值(value)和运算的集合，这往往代表了对象具有的问题域(problem domain)或属性(attribute)。例如：Integer，Float，String。

-- 类(class) 在面向对象编程，类（英语：class）是一种面向对象计算机编程语言的构造，是创建对象的蓝图，描述了所创建的对象共同的属性和方法。类的更严格的定义是由某种特定的元数据所组成的内聚的包。它描述了一些对象的行为规则，而这些对象就被称为该类的实例。

-- 函数 函数是输入变量到输出变量的一种映射，而这种映射在计算机中往往以算法体现。例如：Haskell中的内置函数sum，take，drop。

-- 递归函数 接受自然数的有限元组并并返回一个单一自然数的偏函数。包括初始函数并闭合在复合、原始递归和μ算子下的最小的偏函数类。简单的意义上说一个函数由基例和递归模块组成且在内部直接或间接调用函数自身那么这个函数就是递归函数。

-- 高阶函数 将函数作为参数输入或输出的可能带有Curring化性质的函数属于高阶函数

-- 多态函数 函数的类型签名里包含类型变量/型别变数，即这个函数的某些参数可以是任意类型/一个变量，那么这种函数就是多态函数(f :: a -> a)

-- 重载 允许在同一作用域中的某个函数和运算符指定多个类型定义的具有class进行约束的函数、运算符或者类型，从而使得它们可以以相同的签名执行多种不同的功能 (f :: Num a => a -> a)

-- 自定义类型 在程序自带的标准库之外，用户通过构造函数自行定义的数据类型，在haskell中体现为type 或者data

-- 代数数据类型 在计算机编程，特别是函数编程和类型论中，代数数据类型是一种复合类型，即通过组合其他类型形成的类型。

-- monad单胞机/单子 In functional programming, a monad is a design pattern that allows structuring programs generically while automating away bookkeeping code needed by the program logic. 在函数式编程中，monad是一种设计模式，它允许通用地构造程序，同时自动化程序逻辑所需的簿记代码。

-- Functor函子 Functor是一种支持fmap函数，可以将操作基本数据类型的函数提升为可以操作这种数据结构的函数的数据结构
```

常用环境

```
{- module -}
module ToolkitForIris where

{- common used packages -}
-- check but don't submit
import Test.QuickCheck -- ghci quickCheck
import Test.HUnit -- testcase

-- about randomRIO
import System.Random

-- about List
import Data.List --

-- about Text Processing
import Data.Char
import Data.String

-- about string and print
import Text.Printf --
import Pictures

-- about record the processing time
import Data.Time

-- about IO
import System.IO.Unsafe
import System.IO
```

语法 & 函数使用

常用函数

```
{-Syntax sugars-}
$
do
>>=

{-数据类型转换-}
-- 从Int到Integer
toInteger
```

```haskell
-- 从Integer到Int
fromIntegral

-- 从Float到Integer（各种四舍五入）
ceiling
floor
round

-- 从Integer到float （单纯的格式转换加小数点）
fromIntegral
fromInteger

-- 从Char到Int
fromEnum cha

-- 从Int到Char
toEnum cha

{-随机数-}
mkStdGen：返回随机生成器
random，randoms：返回一个以及无限多个随机数（列表）
randomR，randomRs：返回某个区域内的一个以及无限多个随机数（列表）
getStdGen：返回系统全局的随机生成器
newStdGen：重置并返回系统全局的随机生成器
randomIO：返回一个随机数（使用全局随机生成器）
randomRIO（a，b）：返回某个区域内的一个随机数（使用全局随机生成器）


{-列表操作-}
--take
take int xs

--drop
drop int xs

--takeWhile
takeWhile lamdaT xs

--dropWhile
dropWhile lamdaT xs

--split
splitAt 3 [1,2,3,4,5] == ([1,2,3],[4,5])

--sum
sum xs

--and
and bools

--or
or bools
```

```haskell
--concat
concat [[1,2,3],[4,5]] == [1,2,3,4,5]

--group
group [1,2,3,4,5,6,6,6] == [[1],[2],[3],[4],[5],[6,6,6]]

-- sortBy
rank :: [(String,Int)] -> [(String,Int)]
rank zs = sortBy (\(a1,b1) (a2,b2) -> compare b2 b1) zs

{-特殊字符输入-}
tab '\t'
line '\n'
backshape '\\'
quote '\''

{-IO how to I/O-}
-- 如何将输入的str转化为Integer
getInte :: IO Integer
getInte = do
  x <- getLine
  return (read x :: Integer)

getIntes :: Integer -> IO [Integer]
getIntes n = do
  x <- getInte
  if n == 1
    then return [x]
    else
      do
        xs <- getIntes (n-1)
        return ([x] ++ xs)

getIntxs :: IO [Int]
getIntxs = do
  x <- getLine
  let xs = map (\x -> read x :: Int) (words x)
  return xs
 getIntsPositive ::  IO [Int]
getIntsPositive = do
  x <- getInt
  if x < 0
    then return []
    else
      do
        xs <- getIntsPositive
        return ([x] ++ xs)

getIntsPositiveSum ::  IO Int
getIntsPositiveSum = do
  x <- getInt
  if x < 0
    then return 0
```

```
    else
      do
        y <- getIntsPositiveSum
        return (x+y)

-- 如何读取文件/写入文件 - 基本main
main :: IO ()
main = do
   ls <- readFile "text.txt"
   let result = string2listofpairs ls
   let formatedResult = formatting result
   writeFile "answer.txt" (formatedResult) -- 覆写
   appendFile "answer.txt" "New edition!" -- 在文件后面添加内容
```

测试方法

quickCheck

```
ghci:
quickCheck xxx
```

Property 书写

```
propAboveBeside3Correct :: Picture -> Picture -> Property
propAboveBeside3Correct w e =
  (rectangular w && rectangular e && height w == height e) -- 这一部分是条件
  ==>
     (w `beside` e) `above` (w `beside` e)
         ==
     (w `above` w) `beside` (e `above` e)
```

测试集设定

```
-- 测试集的设定
testMax1 = TestCase (assertEqual "for: maxThree 6 4 1" 6 (maxThree 6 4 1))
testMax2 = TestCase (assertEqual "for: maxThree 6 6 6" 6 (maxThree 6 6 6))
testMax3 = TestCase (assertEqual "for: maxThree 2 6 6" 6 (maxThree 2 6 6))
testMax4 = TestCase (assertEqual "for: maxThree 2 2 6" 6 (maxThree 2 2 6))

testsMax = TestList [testMax1, testMax2, testMax3, testMax4]

ghci:
runTestTT testsMax
```

可复用代码块

奇偶判定

```
isOdd, isEven :: Int -> Bool

isOdd n
  | n<=0        = False
  | otherwise   = isEven (n-1)

isEven n
  | n<0         = False
  | n==0        = True
  | otherwise   = isOdd (n-1)
```

中位数

```
middleNumber :: Integer -> Integer -> Integer -> Integer
middleNumber x y z
  | between y x z      = x
  | between x y z      = y
  | otherwise          = z
```

列表最大值实现

```
-- 找到一列数中的最大值
mymax :: [Int] -> Int
mymax [] = 0
mymax xs = maximum xs

-- 找到某函数作用后的结果在0-n中的最大值
findTheMax2 :: (Integer -> Integer) -> Integer -> Integer
findTheMax2 f n
  | n == 1 = max (f 1) (f 0)
  | otherwise = max (f n) (findTheMax2 f (n-1))

--
```

列表元素查重

```haskell
rmCopies :: (Eq a) => [a] -> [a]
rmCopies [] = []
rmCopies (x:xs) = x : rmCopies [ y | y <-xs, y/=x ]
```

排序-sort

```haskell
-- 插入式排序
iSort :: [Integer] -> [Integer]
iSort []     = []
iSort (x:xs) = ins x (iSort xs)

ins :: Integer -> [Integer] -> [Integer]
ins x []    = [x]
ins x (y:ys)
  | x <= y      = x:(y:ys)
  | otherwise   = y : ins x ys

--逐个元素比较大小并分割区间进行快速排序
qSort :: [Integer] -> [Integer]
qSort [] = []
qSort (x:xs)
  = qSort [ y | y<-xs , y<=x] ++ [x] ++ qSort [ y | y<-xs , y>x]

--从大到小排序
qSortBack :: [Integer] -> [Integer]
qSortBack [] = []
qSortBack (x:xs)
  = qSortBack [ y | y<-xs , y>=x] ++ [x] ++ qSortBack [ y | y<-xs , y<x]

--查重后再进行排序
qSortRMC :: [Integer] -> [Integer]
qSortRMC xs = qSort (rmCopies xs)

qSortBackRMC :: [Integer] -> [Integer]
qSortBackRMC xs = qSortBack (rmCopies xs)
```

分组-group

```haskell
-- 分组 将所有相同的元素聚集在一个子分组中
groupi :: [String] -> [[String]]
groupi [] = []
groupi xs = group (sort xs)
```

三元系列实现

```haskell
-- 同时合并三个列表
zip3 :: [a] -> [b] -> [c] -> [(a,b,c)]
zip3 (a:as) (b:bs) (c:cs) = (a,b,c) : zip3 as bs cs
zip3 _      _      _      = []

-- 比较三者是否相等 引申为排序etc
threeEqual :: Integer -> Integer -> Integer -> Bool
threeEqual m n p = (m==n) && (n==p)

maxThree :: Integer -> Integer -> Integer -> Integer
maxThree x y z
  | (x >= y) && (x >= z)    = x
  | y >= z                  = y
  | otherwise               = z

maxThree2 :: Integer -> Integer -> Integer -> Integer
maxThree2 x y z = (x `max` y) `max` z

minThree :: Integer -> Integer -> Integer -> Integer
minThree x y z = (x `min` y) `min` z
```

字符转换定位

```haskell
-- 大小写转换实现
toUpper, toLower :: Char -> Char
toUpper ch = toEnum (fromEnum ch + offset)
toLower ch = toEnum (fromEnum ch - offset)
    where
    offset = fromEnum 'A' - fromEnum 'a'

-- 注意其中！！是一个提取从零开始计数的列表元素的operator
figureOut :: Int -> Picture
figureOut int
  | int <= ord 'Z' && int >= ord 'A' = alphabet !! (int - ord 'A')
  | int <= ord '9' && int >= ord '0' = numbers !! (int - ord '0')
```

字符右对齐（左对齐参考haskellStore)

```haskell
pushRight :: String -> String
pushRight xs
  | linelength - length xs <= 0 = xs
  | linelength - length xs /= 0 = pushRight (" " ++ xs)
    where
      linelength = 12
```

实践案例

质数 & 公因子List

```haskell
-- PrimeList Infinite: 1st must be a prime. And then try to filter out all non-prime of it. (艾拉托斯特尼筛法)
primes :: [Integer]
primes = filterPrime [2..]
  where filterPrime (p:xs) =
          p : filterPrime [x | x <- xs, x `mod` p /= 0]

-- 素数列表 有限
primes :: Int -> [Int]
primes n = [ x | x <- [2..n], isPrime x]

-- 素数判定 自带
import Math.NumberTheory.Prime
isPrime

--素数判定简单实现
isPrime :: Int -> Bool
isPrime n = pfactors n == [n]
    where
    pfactors :: Int -> [Int]
    pfactors n = [ i | i <- [2..n], n `mod` i == 0]
```

mod和div的实现

```haskell
remainder :: Integer -> Integer -> Integer
remainder m n
  | m<n         = m
  | otherwise   = remainder (m-n) n

divide    :: Integer -> Integer -> Integer
divide m n
  | m<n         = 0
  | otherwise   = 1 + divide (m-n) n
```

斐波那契数列

```
-- fastfib是找到第n个斐波那契数列数的快速方法
fibStep :: (Integer,Integer) -> (Integer,Integer)
fibStep (u,v) = (v,u+v)

fibPair :: Integer -> (Integer,Integer)
fibPair n
  | n==0        = (0,1)
  | otherwise   = fibStep (fibPair (n-1))

fastFib :: Integer -> Integer
fastFib = fst . fibPair

-- 输出斐波那契数列
onSeparateLines2 :: [String] -> String
onSeparateLines2 xss
  | n <= 2 = head xss ++ xss !! 1
  | otherwise = onSeparateLines2 (take (length xss - 2) xss) ++ "\n" ++ xss !! (n-1) ++
xss !! n
    where
      n = length xss - 1

finalSL2 :: [String] -> IO()
finalSL2 xss = putStr (onSeparateLines2 xss)

fibTable :: Integer -> IO()
fibTable x = finalSL2 (map pushRight (fibString x))

fibString :: Integer -> [String]
fibString x
  | x == 0 = ["n"] ++ ["fib n"]
  | otherwise = fibString (x-1) ++ [show x] ++ [show (fastFib x)]
```

分数运算 - 最大公因子、优先级

```
--module
module Myfraction where

--import
import Test.QuickCheck

--type
type Fraction = (Integer,Integer)

--1 functions
ratplus, ratminus, rattimes, ratdiv :: Fraction -> Fraction -> Fraction
ratfloor :: Fraction -> Integer
ratfloat :: Fraction -> Float
```

```
rateq :: Fraction -> Fraction -> Bool

neg :: Fraction -> Fraction
neg (a,b) =  (-a,b)

ratplus (x2,y2) (x1,y1) = (div (x2*y1+x1*y2) de, div (y1*y2) de)
  where
    de = gcd (x2*y1+x1*y2) (y2*y1)

ratminus (x2,y2) (x1,y1) = (div (x2*y1-x1*y2) de, div (y1*y2) de)
  where
    de = gcd (x2*y1-x1*y2) (y2*y1)

rattimes (x2,y2) (x1,y1) = (div (x2*x1) de, div (y1*y2) de)
  where
    de = gcd (x2*x1) (y1*y2)

ratdiv (x2,y2) (x1,y1) = (div (x2*y1) de, div (x1*y2) de)
  where
    de = gcd (x2*y1) (y2*x1)

ratfloor (x2,y2) = floor (ratfloat (x2,y2))

ratfloat (x2,y2) = fromInteger x2 / fromInteger y2

rateq (x2,y2) (x1,y1) = div x2 (gcd x2 y2) == div x1 (gcd x1 y1) && div y2 (gcd x2 y2)
== div y1 (gcd x1 y1)

--2 operators
infix 6 <+>
(<+>) :: Fraction -> Fraction -> Fraction
(<+>) (x1,y1) (x2,y2) = ratplus (x1,y1) (x2,y2)

infix 6 <->
(<->) :: Fraction -> Fraction -> Fraction
(<->) (x1,y1) (x2,y2) = ratminus (x1,y1) (x2,y2)

infix 7 <-*->
(<-*->) :: Fraction -> Fraction -> Fraction
(<-*->) (x1,y1) (x2,y2) = rattimes (x1,y1) (x2,y2)


infix 7 </>
(</>) :: Fraction -> Fraction -> Fraction
(</>) (x1,y1) (x2,y2) = ratdiv (x1,y1) (x2,y2)

infix 4 <==>
(<==>) :: Fraction -> Fraction -> Bool
(<==>) (x1,y1) (x2,y2) = rateq (x1,y1) (x2,y2)

--3 property
prop_ratplus_unit :: Fraction -> Property
prop_ratplus_unit (x,y) = y > 0 ==> (x,y) <+> (0,1) <==> (x,y)
```

```
prop_ratminus :: Fraction -> Property
prop_ratminus (x,y) = y > 0 ==> (x,y) <-> (0,1) <==> (x,y)

prop_rattimes :: Fraction -> Property
prop_rattimes (x,y) = y > 0 ==> (x,y) <-*-> (0,1) <==> (0,1)

prop_ratdiv :: Fraction -> Property
prop_ratdiv (x,y) = y > 0 ==> (x,y) </> (1,1) <==> (x,y)
```

最大公约数以及阶乘函数的实现 - Guards

```
--Module
module Test where

--import
import Test.QuickCheck

--1.非负整数最小公因子函数--辗转相除法
myGcd :: Integer -> Integer -> Integer
myGcd x y
  | x >= y = myGcdL x y
  | x <= y = myGcdL y x

myGcdL :: Integer -> Integer -> Integer
myGcdL x y
  | mod x y == 0 = y
  | mod x y /= 0 = myGcdL y (mod x y)

--1.非负整数最小公因子函数--辗转相减法
myGcdMinus :: Integer -> Integer -> Integer
myGcdMinus x y
  | abs x >= abs y = myGcdM (abs x) (abs y)
  | abs x <= abs y = myGcdM (abs y) (abs x)

myGcdM :: Integer -> Integer -> Integer
myGcdM x y
  | x - y == 0 = y
  | x /= 0 && y == 0 = x
  | x - y /= 0 && (x-y) >= y = myGcdM (x - y) y
  | x - y /= 0 && (x-y) <= y = myGcdM y (x-y)

--如果需要包含负数
myGcdNegative :: Integer -> Integer -> Integer
myGcdNegative x y
  | x >= 0 && y >= 0 = myGcdMinus x y
  | x <= 0 && y <= 0 = myGcdMinus (abs x) (abs y)
  | x <= 0 && y >= 0 = -myGcdMinus (abs x) y
```

```
  | y <= 0 && x >= 0 = -myGcdMinus (abs y) x

--简化版的负数 参考abs加成后的myGcdMinus

--Properties 其实比较怀疑gcd使用的是辗转相减法
{-prop_myGcd :: Integer -> Integer -> Bool
prop_myGcd x y = myGcd x y == gcd x y-}

prop_myGcdM :: Integer -> Integer -> Bool
prop_myGcdM x y = myGcdMinus x y == gcd x y

--2.阶乘函数
fac :: Integer -> Integer
fac n
  | n==0 = 1
  | n>0 = fac (n-1) * n
  | otherwise = error "fac only defined on natural numbers"

--3.阶乘累加函数
sumFacs :: Integer -> Integer
sumFacs n
  | n==0 = 1
  | n>0 = sumFacs (n-1) + fac n

--4.广义累加函数
sumFun :: (Integer -> Integer) -> Integer -> Integer
sumFun f n
  | n==0 = f 0
  | n>0 = sumFun f (n-1) + f n
```

牛顿拉夫森 - 我求平方根 - Guards

```
--Module
module Newton_Raphson where

--import
import Test.QuickCheck

--Functions
squareroot2 :: Float -> Integer -> Float
squareroot2 x n
  | n == 0 = x
  | n >= 0 = (squareroot2 x (n-1) + 2/squareroot2 x (n-1))/2

squareroot :: Float -> Float -> Integer -> Float
squareroot r x n
  | n == 0 = x
  | n >= 0 = (squareroot r x (n-1) + r/squareroot r x (n-1))/2
```

```
{-sqrtSeq :: Float -> Float -> [Float]
sqrtSeq x r = squareroot x r n : sqrtSeq x r
  where
    n = toInteger (length (sqrtSeq x r))
  -}

sqrtSeq :: Float -> Float -> [Float]
sqrtSeq r x = x : sqrtSeq r (squareroot r x 1)

squareroot' :: Float -> Float -> Float -> Float
squareroot' r x epsilon
    = head [a |(a,b) <- zip (sqrtSeq r x) (0 : sqrtSeq r x), abs (a - b)< epsilon]

--test
{-
prop_sqrt' :: Float -> Float -> Float -> Bool
prop_sqrt' r x epsilon
  | r > 0 && epsilon > 0 && epsilon < 1 && x > 0 =(squareroot' r x epsilon)**2 - r <
epsilon
  | otherwise = True

prop_sqrt :: Float -> Float -> Integer -> Bool
prop_sqrt r x n
  | r > 0 && n > 100 && x > 0 =(squareroot r x n)**2 - r < 1
  | otherwise = True

prop_sqrt2 :: Float -> Integer -> Bool
prop_sqrt2 x n
  | n > 100 && x > 0= (squareroot2 x n)**2 - 2 < 1
  | otherwise = True
-}
```

字符放大 - 应该不考

```
--module
module MyPicture where

--import
import Data.Char
import Data.List
import Test.QuickCheck
import Pictures

-- Data
alphabet :: [Picture]
alphabet = [picA, picB, picC, picD, picE, picF, picG,
            picH, picI, picJ, picK, picL, picM, picN,
            picO, picP, picQ, picR, picS, picT, picU,
```

```
            picV, picW, picX, picY, picZ]

numbers :: [Picture]
numbers = [pic0, pic1, pic2, pic3, pic4, pic5, pic6, pic7, pic8, pic9]

-- Function
sideBySide :: [Picture] -> Picture
sideBySide (pic:pics)
  | length pics == 1 = pic `beside` (concat pics)
  | otherwise = pic `beside` (sideBySide pics)

figureOut :: Int -> Picture
figureOut int
  | int <= ord 'Z' && int >= ord 'A' = alphabet !! (int - ord 'A')
  | int <= ord '9' && int >= ord '0' = numbers !! (int - ord '0')

say :: String -> Picture
say str = sideBySide [figureOut (ord (toUpper ch)) | ch <- str, isUpper ch || isLower
ch || isDigit ch]

sayIt :: String -> IO()
sayIt = printPicture.say

--Alphabet
picA, picB, picC, picD, picE, picF, picG :: Picture
picH, picI, picJ, picK, picL, picM, picN :: Picture
picO, picP, picQ, picR, picS, picT, picU :: Picture
picV, picW, picX, picY, picZ :: Picture
pic0, pic1, pic2, pic3, pic4, pic5, pic6, pic7, pic8, pic9 :: Picture

-- 没有补充具体字符画，不考，考就提供标准图
```

ASCII图形 - Picture系列

```
module Pictures where
import Test.QuickCheck


type Picture = [[Char]]

-- The example used in Craft2e: a polygon which looks like a horse. Here
-- taken to be a 16 by 12 rectangle.

horse :: Picture

horse = [".......##...",
         ".....##..#..",
         "...##.....#.",
         "..#.......#.",
         "..#...#...#.",
```

```
          "..#...###.#.",
          ".#....#..##.",
          "..#...#.....",
          "...#...#....",
          "....#..#....",
          ".....#.#....",
          "......##...."]

-- Completely white and black pictures.

white :: Picture

white = ["......",
          "......",
          "......",
          "......",
          "......",
          "......"]

black = ["######",
          "######",
          "######",
          "######",
          "######",
          "######"]

-- Getting a picture onto the screen.

printPicture :: Picture -> IO ()

printPicture = putStr . concat . map (++"\n")


-- Transformations of pictures.
-- ^^^^^^^^^^^^^^^^^^^^^^^^^^^^

-- 旋转 垂直翻转 Reflection in a vertical mirror.

flipV :: Picture -> Picture

flipV = map reverse

-- 旋转 水平翻转 Reflection in a horizontal mirror.

flipH :: Picture -> Picture

flipH = reverse

-- Rotation through 180 degrees, by composing vertical and horizontal
--旋转 90°旋转，其他相关的应该都是ok的，不过没有办法旋转一个1°这样的概念。
rotate90 :: Picture -> Picture
rotate90 x
  | length (concat x) == length x = [reverse (map head x)]
```

```haskell
  | otherwise = reverse (map head x) : rotate90 (map tail x)

--旋转 反向旋转90°
anticRotate90 :: Picture -> Picture
anticRotate90 x = rotate90 (rotate90 (rotate90 x))

-- reflection. Note that it can also be done by flipV.flipH, and that we
-- can prove equality of the two functions.

rotate :: Picture -> Picture

rotate = flipH . flipV

-- One picture above another. To maintain the rectangular property,
-- the pictures need to have the same width.

above :: Picture -> Picture -> Picture

above = (++)

-- One picture next to another. To maintain the rectangular property,
-- the pictures need to have the same height.

beside :: Picture -> Picture -> Picture

beside = zipWith (++)

-- Superimose one picture above another. Assume the pictures to be the same
-- size. The individual characters are combined using the combine function.

superimpose :: Picture -> Picture -> Picture

superimpose = zipWith (zipWith combine)

-- For the result to be '.' both components have to the '.'; otherwise
-- get the '#' character.

combine :: Char -> Char -> Char

combine topCh bottomCh
  = if (topCh == '.' && bottomCh == '.')
    then '.'
    else '#'

-- Inverting the colours in a picture; done pointwise by invert...

invertColour :: Picture -> Picture

invertColour = map (map invert)

-- ... which works by making the result '.' unless the input is '.'.

invert :: Char -> Char
```

```
invert ch = if ch == '.' then '#' else '.'

-- 4图像组合
fourPics1 :: Picture -> Picture
fourPics1 pic =
    left `beside` right
      where
        left  = pic `above` invertColour pic
        right = invertColour (flipV pic) `above` flipV pic

-- 如何输出图像
printPicture :: Picture -> IO ()
-- printPicture = putStr . concat . map (++"\n")
-- printPicture = putStr . concatMap (++"\n")
printPicture = putStr . unlines

--
```

Shape - 基本data定义以及instance

```
module Shape where

import Test.QuickCheck

--data Item = Itemize Name Amount Price

data Shape = Circle Float |
             Rectangle Float Float|
             Triangles Float Float Float
         deriving (Eq,Ord,Show,Read)

isRound :: Shape -> Bool
isRound (Circle _)      = True
isRound _ = False

--求周长
perimeter :: Shape -> Float
perimeter (Circle r) = 2*pi*r
perimeter (Rectangle h w) = 2*(h+w)
perimeter (Triangles a b c) = a + b + c

--求面积
area :: Shape -> Float
area (Circle r)      = pi*r*r
area (Rectangle h w) = h*w
area (Triangles a b c) = sqrt (s*(s-a)*(s-b)*(s-c))
  where
    s = 1/2 * (a + b + c)
```

```haskell
--确定是否是正三角形/正方形
isShapeRegular :: Shape -> Bool
isShapeRegular shape =
    case shape of
        (Triangles a b c) -> (a == b) && (b == c)
        (Circle r) -> True
        (Rectangle a b) -> a == b

-- 四季
data Season = Spring | Summer | Autumn | Winter
    deriving (Eq,Ord,Enum,Show,Read)

isHot :: Season -> Bool
isHot season =
    case season of
        Spring -> False
        Summer -> True
        Autumn -> False
        Winter -> False

nextSeason :: Season -> Season
nextSeason season =
    case season of
        Spring -> Summer
        Summer -> Autumn
        Autumn -> Winter
        Winter -> Spring
```

寻找毕达哥拉斯三元组 - List Comprehension

```haskell
module Triands where

--这个是直接输出结果的

triands :: Integer -> [(Integer, Integer, Integer)]
triands n = [(x,y,z) |  x <- [1..n], y <- [1..n], z <- [1..n],
            (fromInteger x)**2 + (fromInteger y)**2 == (fromInteger z)**2]

--这个可以避免输出重复的结果
triands2 :: Integer -> [(Integer, Integer, Integer)]
triands2 n = [(x,y,z) |  x <- [1..n], y <- [1..n], z <- [1..n],
            (fromInteger x)**2 + (fromInteger y)**2 == (fromInteger z)**2
            && x <= y]
```

购物小票打印 & 图书馆借阅 - list operation 输出格式化

```haskell
{-Here we don't use the method given by TA but some functions defined when learning
Picture.-}

--Module
module HaskellStore where

--import
import Test.QuickCheck
import Text.Printf

--type
type Picture = [String]

type Name = String
type Price = Float
type Amount = Float

type Items = [Item]
type Item = (Name, Amount, Price)

--function
titleRow :: String
titleRow = pushLeft "Name" ++ pushLeft "Amount"
        ++ pushLeft "Price" ++ pushLeft "Sum"

showPre :: Float -> String
showPre = printf"%.2f"

pushLeft :: String -> String
pushLeft xs
  | n - length xs <= 0 = xs
  | n - length xs /= 0 || xs == "" = pushLeft (xs ++ " ")
    where
      n = 12

fst3 :: (a, b, c) -> a
fst3 (a, _, _) = a

snd3 :: (a, b, c) -> b
snd3 (_, b, _) = b

third :: (a, b, c) -> c
third (_, _, c) = c

total1 :: Items -> Float
total1 items
  | null items = 0
  | length items == 1 = third (head items) * snd3 (head items)
  | otherwise = third (head items) * snd3 (head items) + total1 (tail items)

total2 :: Items -> Float
total2 items
```

```
  | null items = 0
  | otherwise = sum [snd3 item * third item | item <- items]

transItemsPic :: Items -> Picture
transItemsPic items = titleRow : [pushLeft (fst3 item) ++ pushLeft (showPre (snd3
item))
  ++ pushLeft (showPre (third item)) ++ pushLeft (showPre (third item * snd3 item)) |
item <- items]
  ++ ["Total  "++ replicate 27 '.' ++ "  " ++ pushLeft (showPre (total2 items))]

printPicture :: Picture -> IO ()
printPicture = putStr . unlines

rmItems :: Item -> Items -> Items
rmItems item items = [itemNext | itemNext <- items, fst3 itemNext /= fst3 item]

(<+>) :: Items -> Items
(<+>) xs = [(fst3 (head xs), sum (map snd3 xs), third (head xs))]

addSameItems :: Items -> Items
addSameItems items
  | length items == 0 = []
addSameItems (item:items) = (<+>) (item : [ item2 | item2 <- items, fst3 item == fst3
item2])
                            ++ addSameItems (rmItems item items)

printItems :: Items -> IO()
printItems items = printPicture (transItemsPic (addSameItems items))

myItems :: Items
myItems = [("huaq2", 12, 12),("huaq2", 2, 12),("huaq1",3,23),("huaq1",12,23),
("huaq2",2,12),("huaq3",12,2)]

--properties
prop_showPre :: Float -> Bool
prop_showPre x = length (showPre x) >= 2

prop_pushLeft :: String -> Bool
prop_pushLeft str = length (pushLeft str) >= 12

--之前提到过的精度问题，尝试更换Double解决
type Price' = Double
type Amount' = Double

type Items' = [Item']
type Item' = (Name, Amount', Price')

total1' :: Items' -> Double
total1' items
  | null items = 0
  | length items == 1 = third (head items) * snd3 (head items)
  | otherwise = third (head items) * snd3 (head items) + total1' (tail items)
```

```haskell
total2' :: Items' -> Double
total2' items
  | null items = 0
  | otherwise = sum [snd3 item * third item | item <- items]

prop_total12 :: Items' -> Bool
prop_total12 items = total1' items == total2' items

prop_transItemPic :: Items -> Bool
prop_transItemPic items = length items + 2 == length (transItemsPic items)
```

```haskell
-- A library database
-- ^^^^^^^^^^^^^^^^^^

-- Types


type Person = String
type Book   = String

type Database = [ (Person , Book) ]

-- An example database.

exampleBase :: Database
exampleBase
  = [ ("Alice" , "Tintin")   , ("Anna" , "Little Women") ,
      ("Alice" , "Asterix") , ("Rory" , "Tintin") ]

-- The books borrowed by a particular person in the given database.

books :: Database -> Person -> [Book]
books dBase findPerson
  = [ book | (person,book) <- dBase , person==findPerson ]

--more readable output
booksoutput :: Database -> Person -> String
booksoutput dBase findPerson
  | n == 0 = head xss
  | otherwise =  onSeparateLines1 (init xss) ++ " " ++ xss !! n
    where
      xss = books dBase findPerson
      n = length xss - 1

finalBkOp :: Database -> Person -> IO()
finalBkOp dBase findPerson
  = putStr ("Borrowed Books:" ++ booksoutput dBase findPerson ++ "\n")

-- The borrowers borrowing a particular book in the given database
borrowers :: Database -> Book -> [Person]
borrowers dBase findBook = [pers | (pers, book) <- dBase, book == findBook]
```

```haskell
-- Judge whether a book is borrowed
borrowed :: Database -> Book-> Bool
borrowed dBase borrowedBook
  | borrowers dBase borrowedBook == [] =False
  | otherwise = True

numBorrowed :: Database -> Person-> Int
numBorrowed dBase person = length (books dBase person)

-- Making a loan is done by adding a pair to the database.

makeLoan   :: Database -> Person -> Book -> Database
makeLoan dBase pers bk = [ (pers,bk) ] ++ dBase

-- To return a loan.

returnLoan   :: Database -> Person -> Book -> Database
returnLoan dBase pers bk = [ pair | pair <- dBase , pair /= (pers,bk) ]

-- Testing the database.

-- Commented out because borrowed is not defined here.

-- test1 :: Bool
-- test1 = borrowed exampleBase "Asterix"

test2 :: Database
test2 = makeLoan exampleBase "Alice" "Rotten Romans"

-- QuickCheck properties for the database

-- Check that bk is in the list of loaned books to pers
-- after making the loan of book to pers

prop_db1 :: Database -> Person -> Book -> Bool

prop_db1 dBase pers bk =
    elem bk loanedAfterLoan == True
         where
           afterLoan = makeLoan dBase pers bk
           loanedAfterLoan = books afterLoan pers

-- Check that bk is not in the list of loaned books to pers
-- after returning the loan of book to pers

prop_db2 :: Database -> Person -> Book -> Bool

prop_db2 dBase pers bk =
    elem bk loanedAfterReturn == False
         where
           afterReturn = returnLoan dBase pers bk
           loanedAfterReturn = books afterReturn pers
```

```haskell
prop_db3 :: Database -> Book -> Person -> Bool
prop_db3 dBase bk pers = [pair | pair <- dBase, pair == (pers, bk)] == []

--type tuple definiton fo database with a list of book for each person
type Database3 =[(Person, [Book])]

--data type definition of database
data Loan = Loan Person Book
type Database2 = [Loan]

exampleBase2 :: Database2
exampleBase2 = [Loan "Alice" "UnderWorld", Loan "Kitro" "StarBurst"]

books2 ::  Database2 -> Person -> [Book]
books2 dBase person = [book | Loan pers book <- dBase, pers == person]
```

购车数据统计-IO操作-str格式化

```haskell
-- module
module Car where

-- import
import Test.QuickCheck
import Data.List

-- function
test :: String
test = "1230197812061080       24596\n1230197210049047       11293\n1331197708184092
    26387\n1331198509014173       21991\n1331198702174176       13778"

-- 数据读取
extract :: String -> [(Int,Int)]
extract [] = []
extract xs = (read (take 16 xs) :: Int, read (take 5 (drop 23 xs)) :: Int) : extract
(drop 29 xs)

-- 数据重新格式化输出
refile1 :: [(Int,Int)] -> String
refile1 [] = []
refile1 ((x,y):xs) = show x ++ "       " ++ show y ++ " \n" ++ refile1 xs

--求特殊[(a,a)]结构的平均值
averagei :: [(Int,Int)] -> Float
averagei xs = fromIntegral (sum [ y | (x,y) <- xs]) / fromIntegral (length xs)

--最终format和输出
final :: IO ()
final = do
  xs <- readFile "bids_201711.txt"
  let zs = extract xs
```

```
  let os = sortBy (\(a1,b1) (a2,b2) -> compare b2 b1) zs
  let as = take 10 os
  let bs =
        "最高成交价: " ++ show (snd (head os)) ++
        "\n最低成交价: " ++ show (snd (last os)) ++
        "\n平均成交价: " ++ show (averagei os) ++
        "\n总共有" ++ show (length os) ++ "参与竞价" ++
        "\n成交名单: \n"
        ++ (refile1 as)
  writeFile "bidResults.txt" (bs)
```

石头剪刀布 - IO 多次输入与do if

```
--module
module Game where

--import
import System.Random
import Test.QuickCheck

--definition Random Hand主要是通过随机得到一个数确定是输出哪一种手势 randomRIO (Rock, Paper)在后续中可以使用的原因?  不过说到底其实我并不是十分清楚
data Hand = Rock | Scissor | Paper
  deriving (Show,Enum,Eq)
instance Random Hand where
    random g = case randomR (0,2) g of
            (r, g') -> (toEnum r, g')
    randomR (a,b) g = case randomR (fromEnum a, fromEnum b) g of
            (r, g') -> (toEnum r, g')

--function
-- 输入字符转换为可评估的手势
transCharHand :: Char -> Hand
transCharHand x
  | toLower x == 'r' = Rock
  | toLower x == 's' = Scissor
  | toLower x == 'p' = Paper

-- 输出手势转换为可以print的文字
transHandString :: Hand -> String
transHandString x
  | x == Rock = "石头"
  | x == Scissor = "剪刀"
  | x == Paper = "布"

-- 如何获得对应的手势
getHand :: IO Hand
getHand = do
  x <- getChar
  return (transCharHand x)
```

```haskell
-- 胜利条件
win :: Hand -> Hand -> Bool
win Rock Scissor = True
win Scissor Paper = True
win Paper Rock = True
win _ _ = False

{-eq :: Hand -> Hand -> Bool
eq Rock Rock = True
eq Scissor Scissor = True
eq Paper Paper = True
eq _ _ = False-}

-- 胜负平局以及输赢判断
winner :: Int -> Int -> IO ()
winner n1 n2 = do
  if (n1 == 3 || n2 == 3)
    then do
      if n1 == 3 then do putStrLn ("算您赢了这轮。")
                 else do putStrLn ("哈哈，我赢了！")
    else do
      putStr ("请您出手 (R)石头，(S)剪刀，(P)布")
      player1 <- getHand
      empty <- getHand
      player2 <- randomRIO (Rock, Paper)
      putStrLn("您出了" ++ transHandString player1 ++ ",我出了" ++ (transHandString
player2))
      if (win player1 player2)
        then do
                putStrLn ("您赢了这手")
                putStrLn ("我的得分: "++ show(n2) ++ "\n您的得分: "++ show(n1+1))
                winner (n1+1) n2
        else do
          if player1 == player2
            then do
                    putStrLn ("这一手平手")
                    putStrLn ("我的得分: "++ show(n2) ++ "\n您的得分: "++ show(n1))
                    winner n1 n2
            else do
                    putStrLn ("我赢了这手")
                    putStrLn ("我的得分: "++ show(n2+1) ++ "\n您的得分: "++ show(n1))
                    winner n1 (n2+1)

-- main 常用的预设参数方法判断
play :: IO()
play = winner 0 0
```

猜数游戏 - IO 多次输入与反馈

```
guessIt :: Int -> Int -> IO ()
guessIt x n = do
  y <- getInt
  if x == y
    then putStrLn ("Bingo and you have guessed "++ show(n) ++ "times")
    else do
      if x > y
        then do putStrLn ("Larger than yours and you have guessed "++ show(n) ++
"times")
                guessIt x (n+1)
        else do putStrLn ("Smaller than yours and you have guessed "++ show(n) ++
"times")
                guessIt x (n+1)

-- 给定
guessNum :: IO ()
guessNum = do
  x <- getInt
  guessIt x 1

-- 随机
guessRandomNum :: IO ()
guessRandomNum = do
  x <- randomRIO (0, 1000)
  guessIt x 1
```

随机事件 - 抛硬币 - 牛顿拉普森骰子问题 - IO 大量测试以及反馈

```
-- 抛硬币
flipCoin :: IO Int
flipCoin =  do
  x <- randomRIO (0, 1)
  return x

getCoins :: Int -> IO [Int]
getCoins n = do
  x <- flipCoin
  if n == 1
    then return [x]
    else
      do
        xs <- getCoins (n-1)
        return ([x] ++ xs)

countIt :: [Int] -> Int
countIt xs = length [x | x <- xs, x == 1 ]

flipManyTimes :: Int -> IO Float
flipManyTimes n = do
  xs <- getCoins n
  let y = countIt xs
```

```
  return ((fromIntegral y)/(fromIntegral n))

flipAverageList :: Int -> Int -> IO [Float]
flipAverageList n m = do
  x <- flipManyTimes n
  if m == 1
    then return [x]
    else
      do
        xs <- flipAverageList n (m-1)
        return ([x] ++ xs)

flipAverage :: IO Float
flipAverage = do
    n <- getInt
    m <- getInt
    xs <- flipAverageList n m
    return ((sum xs)/(fromIntegral m))

-- Average
computeAverage :: IO ()
computeAverage = do
  xs <- readFile "input.txt"
  let ys = [(read x :: Int) | x <- (words xs)]
  let y = average ys
  writeFile "output.txt" (show y)

average :: [Int] -> Float
average xs = fromIntegral.sum xs / length xs
```

```
-- Newton-Pepys problem 牛顿-拉普森 问题 骰子大法

--使用说明
{-
--模拟方法和过程
模拟方法的实现过程如下:
1)dice首先模拟了一次投掷骰子的点数
2)dices然后模拟了同时投掷n个投掷骰子所得到的点数的列表
3)countIt对dices中提供的列表进行6点数的元素计数
4)manyDices形成进行count次dices中6的点数出现次数的列表
5)countThem对manyDices中的6的点数出现次数中大于y的元素计数
6)provement最终集合了上述步骤，对同时投掷n枚骰子中出现至少y次6的事件重复count次进行了
概率计算

--原理以及使用说明
在使用这个对Newton-Pepys problem进行求解的方式是：调用provement函数进行测试。

在provement函数中，provement n y count代表了n个正常的骰子独立投掷，至少出现y个6的
事件在进行count次实验后所得到的频率,当count足够大时（经过测试，大概10000次以上
是比较有准确的），可以认为其为事件发生的概率，因此在对
1)provement 6 1 10000
2)provement 12 2 10000
```

```
3)provement 18 3 10000
进行相应的测试之后，对比其计算出的频率的大小可以大致地估计其概率大小，从而可以验证哪一种
情况发生的概率较大。

最终的结果是第一种情况出现的概率最大。

PS.当然，其实这就是一个概率论问题，在wiki上已经给出了概率解，因此我们不考虑实现这种较为
简单的概率运算，选择了重复试验的方法进行验证。
-}

--module
module NewtonPepysProblem where

--import
import System.Random
import Test.QuickCheck

--type


--function
dice :: IO Int
dice =  do
  x <- randomRIO (1, 6)
  return x

dices :: Int -> IO [Int]
dices n = do
  x <- dice
  if n == 1
    then return [x]
    else
      do
        xs <- dices (n-1)
        return ([x] ++ xs)

countIt :: [Int] -> Int
countIt xs = length [x | x <- xs, x == 6 ]

manyDices :: Int -> Int -> IO [Int]
manyDices n count = do
  xs <- dices n
  if count == 1
    then return ([countIt xs])
    else
      do
        counts <- manyDices n (count-1)
        return ([countIt xs] ++ counts)

countThem :: [Int] -> Int -> Int
countThem xs y = length [x | x <- xs, x >= y ]

provement :: Int -> Int -> Int -> IO Float
```

```
provement n y count = do
  xs <- manyDices n count
  let z = countThem xs y
  return ((fromIntegral z)/(fromIntegral count))
```

电话号码确认 - IO

```
```

确认信用卡号 - IO

```
module Main where

-- 综合函数 判断标准
isValid :: Integer -> Bool
isValid x = if mod (sum (splitNumGtTen (doubleSecondDigit (formARevList x)))) 10 == 0
then True else False

-- 计算符合条件的卡号数
numValid :: [Integer] -> Integer
numValid xs = sum . map (\_ -> 1) $ filter isValid xs

-- 判断标准实现
--找10的倍数列表元素
formARevList :: Integer -> [Integer]
formARevList x
    | x <= 9 = [x]
    | otherwise = formARevList (mod x 10) ++ formARevList (div x 10)

-- 二倍
doubleSecondDigit :: [Integer] -> [Integer]
doubleSecondDigit [x] = [x]
doubleSecondDigit (x:[y]) = [x] ++ [2*y]
doubleSecondDigit (x:y:xs) = [x] ++ [2*y] ++ doubleSecondDigit xs

-- 分割
splitNumGtTen :: [Integer] -> [Integer]
splitNumGtTen [x]
    | x <= 9 = [x]
    | otherwise = [div x 10] ++ [mod x 10]
splitNumGtTen (x:xs)
    | x <= 9 = [x] ++ splitNumGtTen xs
    | otherwise = [div x 10] ++ [mod x 10] ++ splitNumGtTen xs

-- main 最终输出
testcard :: IO ()
```

```
testcard = do
    xs <- readFile "cards200.txt"
    let ys = [(read x :: Integer) | x <- (words xs)]
    let y = numValid ys
    putStr (show y ++ show (length ys))

-- main 指定个数的信用卡中有多少个是真实的
getInte :: IO Integer
getInte = do
  x <- getLine
  return (read x :: Integer)

getIntes :: Integer -> IO [Integer]
getIntes n = do
  x <- getInte
  if n == 1
    then return [x]
    else
      do
        xs <- getIntes (n-1)
        return ([x] ++ xs)

formatting :: [Integer] ->String
formatting [] = []
formatting (x:xs) = show x ++ " \n" ++ formatting xs

main :: IO ()
main = do
    n <- getInte
    xs <- getIntes n
    let ys = [y | y <- xs, isValid y]
    putStr (formatting ys)
```

词频统计 - Parser- IO 操作

```
-- Module
module Main where

-- Import
import Data.List
import Data.Char
import Data.String


-- Function
{-补充内容
```

```
--如何取得单个单词
getWord :: String -> String
getWord []    = []
getWord (x:xs)
  | elem x whitespace   = []
  | otherwise           = x : getWord xs

--如何取得特定单词数的行
type Line = [Word]
getLine :: Int -> [Word] -> Line
getLine len []     = []
getLine len (w:ws)
  | length w <= len     = w : restOfLine
  | otherwise           = []
    where
    newlen      = len - (length w + 1)
    restOfLine  = getLine newlen ws
-}

-- 将字符串转换为小写
toLowerStr :: String -> String
toLowerStr str = [ toLower x | x <- str]

-- 筛选 去除groupi之后中的空元素
groupSp :: [[String]] -> [[String]]
groupSp xss = [xs | xs <- xss , not (null (head xs)) ]

-- 分组 将所有相同的元素聚集在一个子分组中
groupi :: [String] -> [[String]]
groupi [] = []
groupi xs = group (sort xs)

-- 计数 对每一个子分组的元素个数进行计数，同时作为元组的一部分进行保存
count :: [[String]] -> [(String,Int)]
count xss = [ (head xs, length xs) | xs <- xss]

-- 排名 比较元组
rank :: [(String,Int)] -> [(String,Int)]
rank zs = sortBy (\(a1,b1) (a2,b2) -> compare b2 b1) zs

-- 替换 对于特定元素进行替换，其中比较重要的是其中有两个元素的时候基例的设定以及递归的写法
replaceSp :: String -> String
replaceSp [] = []
replaceSp [wd] = [wd]
replaceSp (wd1:wd2:ls)
  | (wd1 == '\37413' && wd2 == '\27290') || (wd1 == '\37413' && wd2 == '\27054') || wd1
== ''' || wd1 == '‘'
        = '\'' : replaceSp ls
  | (wd1 == '.') || (wd1 == ',') || (wd1 == '\"') || (wd1 == '/') || (wd1 == '(') ||
(wd1 == ')')  = ' ' : replaceSp (wd2 :ls)
  | otherwise =  wd1 : replaceSp (wd2 :ls)

-- 过滤 可定制过滤元件，如果没有成功的话
```

```haskell
wordFilter :: [String] -> [String]
wordFilter [] = []
wordFilter (str:strs)
    | str == "warm-up:" = "warm-up" : wordFilter strs
    | elem '\'' str = (takeWhile (\x -> x /= '\'') str) : wordFilter strs
    | elem '-' str = str : wordFilter strs
    | null str = wordFilter strs
    | and (map (\x -> isAlpha x || isDigit x) str) = str : wordFilter strs
    | otherwise = filter (\x -> isAlpha x || isDigit x) str : wordFilter strs

-- 综合函数 注意使用的顺序即可
string2listofpairs :: String ->[(String, Int)]
string2listofpairs xs = rank $ count $ groupSp $ groupi $ wordFilter $ words $
replaceSp $ toLowerStr xs

-- 对结果进行格式化str输出
formatting :: [(String, Int)] ->String
formatting [] = []
formatting ((x,y):xs) = x ++ " " ++ show y ++ "\n" ++ formatting xs

-- main函数 读入与书写
main :: IO ()
main = do
   ls <- readFile "text.txt"
   let result = string2listofpairs ls
   let formatedResult = formatting result
   writeFile "answer.txt" (formatedResult)

-- detect 确认其中出错的内容（尤其是最开始出现错误的内容是什么）
format :: [(String, String)] -> String
format [] =[]
format ((x,y):xs) = x ++ ".." ++ y ++ "\n" ++ format xs

check :: IO ()
check = do
   ls1 <- readFile "answer.txt"
   ls2 <- readFile "ansb2.txt"
   let ys = [ (x, y) |(x, y) <- zip (lines ls1) (lines ls2), x /= y ]
   writeFile "checklist.txt" (format ys)
```

卡牌比赛 - 计分 - 基本Data Type 定义

```haskell
odule CardPlayer where

--import
-- import Test.QuickCheck


--type
type Team = (Player,Player)
```

```haskell
data Player = East | South | West | North | None
  deriving (Eq,Show,Ord)

data Suit = Spladespades | Hearts | Diamonds| Clubs
  deriving (Eq,Show,Ord)
type Value = Int
type Deck = (Suit, Value)

type Hand = ([Deck],Player)
type Hands = [Hand]

type Trick = [(Deck,Player)]

--functions
myTrick1 :: Trick
myTrick1 = [((Hearts,2),East),((Hearts,6),South),((Hearts,4),West),((Hearts,5),North)]

myTrick2 :: Trick
myTrick2 = [((Hearts,2),East),((Hearts,6),South),((Hearts,4),West),((Clubs,5),North)]

myTricks :: [Trick]
myTricks = [myTrick1, myTrick2]

myHands :: Hands
myHands = [([(Hearts,2),(Hearts,5)],East),([(Hearts,3)],South),([(Hearts,4)],West),
([(Hearts,5)],North)]

teamNS :: Team
teamNS = (North, South)

teamEW :: Team
teamEW = (East, West)

teamNone :: Team
teamNone = (None,None)

winNT :: Trick -> [Player]
winNT trick = [snd step | step <- trick, snd (fst step) == maximum (map (snd.fst)
trick)]

winT :: Suit -> Trick -> [Player]
winT suit trick
  | suit `elem` map (fst.fst) trick = winNT [step | step <- trick, (fst.fst) step ==
suit]
  | otherwise = winNT trick

checkPlay :: Hands -> Trick -> Bool
checkPlay hands trick
  | length hands == 1 =
          null [deck | deck <- [fst step | step <- trick, snd step == snd (head
hands)],
          deck `notElem` fst (head hands)]
  | null [deck | deck <- [fst step | step <- trick, snd step == snd (head hands)],
```

```
          deck `notElem` fst (head hands)] && checkPlay (tail hands) trick
          && null [step | step <-trick, (fst.fst) step /= fst (fst (head trick))] =
True
  | otherwise = False

winNS :: [Trick] -> Int
winNS tricks = length [ winner | winner <- map winNT tricks, North `elem` winner ||
South `elem` winner]

winEW :: [Trick] -> Int
winEW tricks = length [ winner | winner <- map winNT tricks,  East `elem` winner ||
West `elem` winner]

winnerNT :: [Trick] -> Team
winnerNT tricks
  | winEW tricks > winNS tricks = teamEW
  | winNS tricks > winEW tricks = teamNS
  | otherwise = teamNone

rmTrick :: Player -> Trick -> Trick
rmTrick player trick = [itemNext | itemNext <- trick, snd itemNext /= player]

checkPlays :: [Trick] -> Bool
checkPlays tricks = checkPlay (hands1 (concat tricks)) (concat tricks)
  where
    hands1 :: Trick -> Hands
    hands1 trick
      | length trick == 0 = []
      | otherwise = ([fst step | step <- trick, snd step == snd (head trick)], snd
(head trick))
        : hands1 (rmTrick (snd (head trick)) trick)
```

算数表达式求值练习 - 代数类型 - Maybe - do系列

```
module CCal where

-- 主要目的是实现所有的计算方程，主要是关注于其中出现可能的报错问题并使用Maybe(构造函数)来对这些问题进
行解决。
-- 其中>>=就是monad。

data Exp = Con Int
    | Var String
    | Add Exp Exp
    | Mul Exp Exp
    | Div Exp Exp
    deriving (Show, Eq)

-- 用于判断是否需要加括号
data Context = Multi | Divi| AnyOther
    deriving (Show, Eq)
```

```haskell
{-代数类型实现 & 进行简单多次测试（参考猜数部分）-}
instance Show Exp Context where
--    show :: Exp -> Context -> String
    show (Con int) _ = show int
    show (Var str) _ = string
    show (Add exp1 exp2) Multi = "(" ++ show exp1 ++ "+" ++ show exp2  ++ ")"
    show (Add exp1 exp2) Divi = "(" ++ show exp1 ++ "+" ++ show exp2  ++ ")"
    show (Add exp1 exp2) AnyOther = "(" ++ show exp1 ++ "+" ++ show exp2  ++ ")"
    show (Mul exp1 exp2) _ = show exp1 ++ "*" ++ show exp2
    show (Div exp1 exp2) _ = show exp1 ++ "/" ++ show exp2

eval :: Exp -> Int
eval (Con i) = i
eval (Add e1 e2) = e1 + e2
eval (Mul e1 e2) = e1 * e2
eval (Div e1 e2) = e1 / e2

expGen :: Int -> Exp
expGen x =
    | x == 1 = Con (randomRIO (1,100))
    | x == 2 = Add (randomRIO (1,100)) (randomRIO (1,100))
    | x == 3 = Mul (randomRIO (1,100)) (randomRIO (1,100))
    | x == 4 = Div (randomRIO (1,100)) (randomRIO (1,100))


genExp :: IO Exp
genExp = do
    decision <- randomRIO  (1,4)
    return (expGen decision)


guessIt :: Int -> IO ()
guessIt x = do
  y <- getInt
  if (eval x) == y
    then
        putStrLn ("Bingo")
    else do
        putStrLn ("False Try Again")
        guessIt

main :: IO ()
main = do
      x <- genExp
      putStrLn $ show x
      guessIt x

{-eval的Maybe实现 模式匹配-}
eval :: Exp -> Maybe Int
eval (Con i) = Just i
eval (Var str) = Just str
eval (Mul e1 e2) = case (eval e1) of
```

```
    Nothing -> Nothing
    Just i1 -> case (eval e2) of
        Nothing -> Nothing
        Just i2 -> Just (i1 * i2)
eval (Div e1 e2) = case (eval e1) of
    Nothing -> Nothing
    Just i1 -> case (eval e2) of
        Nothing -> Nothing
        Just i2 -> if i2 ==0 then Nothing else Just (i1 `div` i2)
eval (Add e1 e2) = case (eval e1) of
    Nothing -> Nothing
    Just i1 -> case (eval e2) of
        Nothing -> Nothing
        Just i2 -> Just (i1 + i2)


safeDiv :: Maybe Int -> Maybe Int -> Maybe Int
safeDiv x Nothing = Nothing
saftDiv Nothing y = Nothing
saftDiv (Just x) (Just y) = if y == 0 then Nothing else Just (div x y)

{-monad实现-}
m_eval1 :: Exp -> Maybe Int
m_eval1 (Con i) = return i
m_eval1 (Div e1 e2) =
    m_eval1 e1 >>= (\i1 ->
        m_eval1 e2 >>= (\i2 ->
            if i2 == 0 then Nothing
                else return (i1 `div` i2)))
m_eval1 (Add e1 e2) =
    m_eval1 e1 >>= \i1 ->
        m_eval1 e2 >>= \i2 ->
            return (i1 + i2)

(//), (///) :: Maybe Int -> Maybe Int -> Maybe Int
x // y = x >>= (\a -> y >>= (\b -> if b == 0 then Nothing else Just (div a b)))
x /// y = do
    a <- x
    b <- y
    if b == 0 then Nothing else Just (div a b)

{-do实现-}
m_evalDO :: Exp -> Maybe Int
m_evalDO (Con i) = return i
m_evalDO (Div e1 e2) = do
    i1 <- m_eval1 e1
    i2 <- m_eval1 e2
    if i2 == 0 then Nothing
        else return (i1 `div`i2)
m_evalDO (Add e1 e2) = do
    i1 <- m_eval1 e1
    i2 <- m_eval1 e2
    return (i1 + i2)
```

```haskell
{-输出结果-}
run_m_eval :: Exp -> IO ()
run_m_eval exp = do
    case m_eval1 exp of
        Nothing -> putStrLn "Something Wrong!"
        Just i -> putStrLn $ show i
```

逻辑判断实现 - instance data 重载函数 -其余同上

```haskell
module Lab3 where

--tips: nub union unions fromList toList
import Data.List

--定义Prop与Prop对应的Show符号
data Prop = Const Bool
    | Var Char
    | Not Prop
    | And Prop Prop
    | Or Prop Prop
    | Imply Prop Prop
        deriving Eq

instance Show Prop where
--    show :: Prop -> String
    show (Const True) = "True"
    show (Const False) = "False"
    show (Var char) = [char]
    show (Not exp1) = "~" ++ show exp1
    show (And exp1 exp2) = show exp1 ++ "&&" ++ show exp2
    show (Or exp1 exp2) = show exp1 ++ "||" ++ show exp2
    show (Imply exp1 exp2) = show exp1 ++ "=>" ++ show exp2

-- 待测试样品
p1, p2, p3, p4 :: Prop
p1 = And (Var 'A') (Not (Var 'A'))
p2 = Or (Var 'A') (Not (Var 'A'))
p3 = Imply (Var 'A') (And (Var 'A') (Var 'B'))
p4 = Imply (Var 'p') (Imply (Var 'q')(Var 'p'))

-- eval 所有的运算的结果instances
type Subst = [(Char, Bool)]

sub1 = [('A', True),('B', False)]

eval :: Subst -> Prop -> Bool
eval sub (Const b) = b
eval sub (Var x) =  head [b | (a, b) <- sub, a == x]
eval sub (Not e1) = not (eval sub e1)
```

```haskell
eval sub (And e1 e2) = eval sub e1 && eval sub e2
eval sub (Or e1 e2) = eval sub e1 || eval sub e2
eval sub (Imply e1 e2) = if (eval sub e1 == True) then eval sub e2
                                                  else True

eval2 :: Prop -> Subst -> Bool
eval2 p sub = eval sub p

--尚未实现。。
--evalMaybe :: Subst -> Prop -> Maybe Bool

-- varc + vars 是找TF对象用的
rmCopies :: (Eq a) => [a] -> [a]
rmCopies [] = []
rmCopies (x:xs) = x : rmCopies [ y | y <-xs, y/=x ]

varc :: Prop -> [Char]
varc (Const True) = ['T']
varc (Const False) = ['F']
varc (Var x) = [x]
varc (And e1 e2) = varc e1 ++ varc e2
varc (Not e1) = varc e1
varc (Or e1 e2) = varc e1 ++ varc e2
varc (Imply e1 e2) = varc e1 ++ varc e2

vars :: Prop -> [Char]
vars p = rmCopies (varc p)

-- 非常巧妙的方法，可以对列表中所有的TF对象组合形成的真值表进行枚举
allsubsts :: [Char] -> [Subst]
allsubsts ['T'] = [[('T', True)]]
allsubsts ['F'] = [[('F', False)]]
allsubsts [x] = [[(x, True)]] ++ [[(x, False)]]
allsubsts (x:xs) = [(x, True):s | s <- ss] ++ [(x, False):s | s <- ss]
    where
        ss = allsubsts xs

-- 集合vars和allsubsts形成完整的真值表
substs :: Prop -> [Subst]
substs p = allsubsts (vars p)

-- 单体永真式检测
isTaut :: Prop -> Bool
isTaut p = and $ map (eval2 p) (substs p)

-- 多永真式检测法 就是先分好类然后进行组合使得结果应该是True，如果出现了False则一般认为是eval的
isTaut出错了
check_it :: IO ()
check_it = do
    let a = and (map isTaut someTauts) -- all are True
    let b = or (map isTaut nonTauts)  -- all are False
    let  c = a && (not b) -- should be True
    print c
```

（备用）列表原理部分讲解

```
{- [a]的元素:

1. [] 是[a]的空列表
2. 如果x::a, xs::[a], 那么x:xs是[a]的列表

[Int]的元素: -}

-- 列表构造原理
[], 1:[] = [1],  2:(1:[]) = [2,1]

sum :: [Int] -> Int
sum []  = 0
sum (x:xs) = x +  sum xs

-- 列表概括
[2*x | x<- [1..3], mod x == 0]


-- 无限列表
[1..]

-- 如何得到斐波那契数列

-- 第n位的斐波那契数
fib :: Integer -> Integer
fib n
  | n==0        = 0
  | n==1        = 1
  | n>1         = fib (n-2) + fib (n-1)

-- 通过斐波那契数列以及与之有一位之差的数字来计算后续的斐波那契梳理
-- fibs      = 1,1,2,3,5,8,...
-- tail fibs = 1,2,3,5,8,,...

fibs :: [Int]
fibs = 1:1: [  x+y | (x,y) <- zip fibs (tail fibs) ]

-- 计算两点之间直线长度
-- find the distance between two points on the plane

type Point = (Float, Float)

distance :: Point -> Point -> Float
distance (x1,y1) (x2,y2) = sqrt ((x1-x2)^2+(y1-y2)^2)

-- 计算多个点之间的路径连接长度
-- find the length of a path
```

```
 [P,    Q,    R,S]

type Path = [Point]
pathLength :: Path -> Float
pathLength [p1,p2] = distance p1 p2
pathLength (p1:p2:ps) = distance p1 p2 + pathLength (p2:ps)

pathDistance :: [Point] -> Float
pathDistance xs = sum [distance x y | (x, y) <- zip (init xs) (tail xs)]

{- 步骤分解input path: p = [P, Q,R,S]
first step :  get the list [(P,Q),(Q,R),(R,S)] (zip [P,Q,R] [Q,R,S], zip (init p) (tail
p))
second step: [distance P Q, distance Q R, distance R S]
last step: sum [distance P Q, distance Q R, distance R S] -}


-- 找偶数
[x |x<-[1..], mod x 2 ==1]

-- 计算平方根
squareroot r x0 0 = x0
squareroot r x0 n = (x' + r/x')/2
   where
   x' = squareroot r x0 (n-1)

-- 平方根序列
sqrtSeq r x0 = x0 : sqrtSeq r (x0+r/x0)/2

-- epsilon-N 语言 极限相关
withIn ::[Float] -> Float -> Float
withIn (x1:x2:xs)  epsilon
   | abs (x1-x2) < epsilon = x2
   | otherwise = withIn (x2:xs) epsilon

--判定是否逼近极限
squareroot' r x0 epsilon = withIn (sqrtSeq r x0) epsilon
```

（备用）高阶函数base实现

```
--从应用上来说，foldr是从list的最后一个元素开始与基例进行操作，而foldl则是从list的第一个元素开始与基例进行操作，从而得到最终的结果。如果想用用大脑计算结果，不妨从这里开始逐层计算。
```

（备用）Monad课堂指南

```haskell
module FunctorMonad where

import CCal
-- 课程记录 结合CCal进行阅读
-- Functor & instances Part
fmapi :: (a -> b) -> Maybe a -> Maybe b
fmapi f Nothing = Nothing
fmapi f (Just x) = Just (f x)

fmapIO :: (a -> b) -> IO a -> IO b
fmapIO f k = do
    x <- k
    return (f x)

fmapxs :: (a -> b) -> [a] -> [b]
fmapxs f [] = []
fmapxs f (x:xs) = f x : fmapxs f xs

--Monad Part参考CCal 通过这种机制可以将不同的函数连接起来，尝试提高每一个function的可复用性
-- a simple case about Monad
getInt :: IO Int
getInt = do
    x <- getLine
    return (read x :: Int)

doubleprint :: Int -> IO ()
doubleprint x = putStrLn (show (2*x))

combine :: IO ()
combine = getInt >>= doubleprint

-- Haskell wiki
For Functor, the fmap function moves inside the Just constructor and is identity on the
Nothing constructor.

For Monad, the bind operation passes through Just, while Nothing will force the result
to always be Nothing.

--Monad MaybeSource#

(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b Source#

(>>) :: Maybe a -> Maybe b -> Maybe b Source#

return :: a -> Maybe a Source#

fail :: String -> Maybe a

-- Maybe type
data Maybe a = Just a | Nothing
     deriving (Eq, Ord)
```

（备用）Merge实现（也可以直接调用data.list中的merge）

```
--Module
module Merge where

--Import
import Test.QuickCheck
import Data.List

--Function
merge :: Ord a => (a -> a -> Ording) [a] -> [a] -> [a]
merge f [] ys = ys
merge f xs [] = xs
merge f (x:xs) (y:ys)
    |

mergeBy :: (a -> a -> Ordering) -> [a] -> [a] -> [a]
mergeBy _cmp [] ys = ys
mergeBy _cmp xs [] = xs
mergeBy cmp (allx@(x:xs)) (ally@(y:ys))
        -- Ordering derives Eq, Ord, so the comparison below is valid.
        -- Explanation left as an exercise for the reader.
        -- Someone please put this code out of its misery.
    | (x `cmp` y) <= EQ = x : mergeBy cmp xs ally
    | otherwise = y : mergeBy cmp allx ys

qSort :: [Integer] -> [Integer]
qSort [] = []
qSort (x:xs)
  = qSort [ y | y<-xs , y<=x] ++ [x] ++ qSort [ y | y<-xs , y>x]

--课程指南
-- 多态 polymorphism

takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x:xs)
   | p x    = x : takeWhile p xs
   | otherwise = []


  -- brain-computer
isLetter :: Char -> Bool
isLetter c = isAlpha c || c == '-'

getWord :: [Char] -> [Char]
getWord ws = takeWhile (\x -> isAlpha x  || c =='-')   ws


作业过程：
```

```
readFile : String
String -> [String] (words)
[String] ->  [String]  (map)
[String] -> [String] (sort)
[String] -> [[String]] (groupBy)
[[String]] -> [(String, Int)] (map ..)
[(String,Int)] -> [(String,Int)] (sort)

[(String,Int)] -> [String] -> String
```

最后writeFile

```
-- 重载 overloading


mergesort ::Ord a =>  [a] -> [a]
mergesort [] = []
mergesort [x] = [x]
mergesort xs  = merge ys1 ys2
   where
   n = length xs
   xs1 = take (div n 2) xs   -- xs的前半部分
   xs2 = drop (dive n 2) xs   -- xs的后半部分
   ys1 = mergesort xs1
   ys2 = mergesort xs2

merge :: Ord a =>  [a] ->  [a] -> [a]
merge [] ys  = ys
merge xs []  = xs
merge (x:xs) ( y:ys)
   | x<= y    = x : merge xs *y:ys)
   |otherwise = y : merge (x:xs) ys
```

归并排序:

```
[3,21,3,2,5,23,12]
```

中间分开，分成两个子列表-> [3,21,3] , [2,5,23,21]

对两个子列表分别排序:

前一个: [3, 3, 21]

后一个: [2, 5, 21,23]

将这两个有序序列合并为一个有序序列:

```
[2,3,3,5,21,21,23]
```

```
-- 按照指定次序排序

mergesort :: Ord a => (a -> a -> Ordering) ->[a] ->  [a]

merge :: Ord a => (a -> a -> Ordering) -> [a] -> [a] -> [a]
merge f [] ys = ys
merge f xs [] = xs
merge f (x:xs) (y:ys)
 | f x y == LT = x : merge f xs (y:ys)    -- x <= y
```

```
  | othrewise  = y : merge f (x:xs) ys


mergesort (\x y -> compare x y) [2,3,1,3,21] -- 从小到大

mergesort (\x y -> compare y x) [2,3,1,3,21] -- 从大到小
```