Haskell资料

Sort

```haskell
    -- 带判断的插入排序
    insert :: (a -> a -> Ordering) -> a -> [a] -> [a]
    insert f x [] = [x]
    insert f x (y : xs) = if (f x y) /= GT then x : y : xs else y : insert f x
xs

    insertionSort :: (a -> a -> Ordering) -> [a] -> [a]
    insertionSort f [] = []
    insertionSort f [x] = [x]
    insertionSort f (x : xs) = insert f x (insertionSort f xs)

    -- 带判断的快排
    partition ::  (a -> a -> Ordering) -> [a] -> ([a], [a])
    partition f x [] = ([], [])
    partition f x xs = ([t | t <- xs, f t x /= GT], [t | t <- xs, f t x == GT])

    quickSort ::  (a -> a -> Ordering) -> [a] -> [a]
    quickSort f [] = []
    quickSort f [x] = [x]
    quickSort f (x:xs) = (quickSort f (fst y)) ++ [x] ++ (quickSort f (snd y))
        where y = partition f x xs


    -- 带判断的归并
    merge :: (a -> a -> Ordering) -> [a] -> [a] -> [a]
    merge f [] [] = []
    merge f [] y = y
    merge f x [] = x
    merge f x y = if f (head x) (head y) == LT then (head x) : merge f (tail x)
y else (head y) : merge f x (tail y)

    mergeSort ::   (a -> a -> Ordering) ->  [a] -> [a]
    mergeSort f [] = [];
    mergeSort f [x] = [x]
    mergeSort f x = merge f (mergeSort f (take y x)) (mergeSort f (drop y x))
        where y = div (length x) 2

    -- 归并
    merge ::  Ord a => [a] -> [a] -> [a]
    merge [] [] = []
    merge [] y = y
    merge x [] = x
    merge x y = if (head x) < (head y) then (head x) : merge (tail x) y else
(head y) : merge x (tail y)

    mergeSort ::  Ord a => [a] -> [a]
    mergeSort [] = []
    mergeSort [x] = [x]
    mergeSort x = merge (mergeSort (take y x)) (mergeSort (drop y x))
```

```haskell
        where y = div (length x) 2

    -- 快排
    partition ::  Ord a => a -> [a] -> ([a], [a])
    partition x [] = ([], [])
    partition x xs = ([t | t <- xs, t <= x], [t | t <- xs, t > x])

    quickSort ::  Ord a => [a] -> [a]
    quickSort [] = []
    quickSort [x] = [x]
    quickSort (x:xs) = (quickSort (fst y)) ++ [x] ++ (quickSort (snd y))
        where y = partition x xs

    -- 选择排序
    erase :: Ord a => a -> [a] -> [a]
    erase x [] = []
    erase x (y:ys) = if x == y then ys else y : (erase x ys)

    selectMin ::  Ord a => [a] -> (a, [a])
    selectMin [x] = (x, [])
    selectMin x = (y, erase y x)
        where y = minimum x

    selectionSort ::  Ord a => [a] -> [a]
    selectionSort [] = []
    selectionSort [x] = [x]
    selectionSort x = fst y : selectionSort (snd y)
        where y = selectMin x

    -- 插入排序
    insert :: Ord a => a -> [a] -> [a]
    insert x [] = [x]
    insert x (y : xs) = if x <= y then x : y : xs else y : insert x xs

    insertionSort :: Ord a => [a] -> [a]
    insertionSort [] = []
    insertionSort [x] = [x]
    insertionSort (x : xs) = insert x (insertionSort xs)
```

MyFraction

```haskell
module MyFraction where
    import Test.QuickCheck

    type Fraction = (Integer, Integer) -- 类型定义

    -- matrix 题目中的化简函数
    normalize :: Fraction -> Fraction
    normalize (a, b) = ((signum b) * (div a x), (signum b) * (div b x))
        where x = gcd a b

    -- 化为浮点数
    ratfloat :: Fraction -> Float
    ratfloat (a, b) = fromInteger(a) / fromInteger(b)

    -- 判断两个分数是否相等
```

```haskell
    rateq :: Fraction -> Fraction -> Bool
    rateq (a, b) (c, d) =  normalize(a, b) == normalize(c, d)

    ratfloor :: Fraction -> Integer
    ratfloor (a, b) = div a b

-- 运算法则
    -- 加法
    ratplus :: Fraction -> Fraction -> Fraction
    ratplus (a, b) (c, d) = normalize(a * d + b * c, b * d)

    -- 减法
    ratminus :: Fraction -> Fraction -> Fraction
    ratminus (a, b) (c, d) = normalize(a * d - b * c, b * d)

    -- 乘法
    rattimes :: Fraction -> Fraction -> Fraction
    rattimes (a, b) (c, d) = normalize(a * c, b * d)

    -- 除法
    ratdiv :: Fraction -> Fraction -> Fraction
    ratdiv (a, b) (c, d) = normalize(a * d, b * c)

-- 运算符定义
-- infix 似乎是优先级相关，参数越高优先级越高

    -- 加法
    infix 5 <+>
    (<+>) :: Fraction -> Fraction -> Fraction
    (<+>) (a, b) (c, d) = ratplus (a, b) (c, d)

    -- 减法
    infix 5 <->
    (<->) :: Fraction -> Fraction -> Fraction
    (<->) (a, b) (c, d) = ratminus (a, b) (c, d)

    -- 乘法
    infix 6 <-*->
    (<-*->) :: Fraction -> Fraction -> Fraction
    (<-*->) (a, b) (c, d) = rattimes (a, b) (c, d)

    -- 除法
    infix 6 </>
    (</>) :: Fraction -> Fraction -> Fraction
    (</>) (a, b) (c, d) = ratdiv (a, b) (c, d)

    -- 判断等于
    infix 4 <==>
    (<==>) :: Fraction -> Fraction -> Bool
    (<==>) (a, b) (c, d) = rateq (a, b) (c, d)
```

Lab3

```haskell
module Lab3(
    Prop(..),
    isTaut -- :: Prop -> Bool
```

```
) where

    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 x) = if x == True then "T" else "F"
        show (Var x) = [x]
        show (And x y) = x_ ++ " && " ++ y_
            where x_ = if length (show x) > 2 then "(" ++ show x ++ ")" else
show x
                  y_ = if length (show y) > 2 then "(" ++ show y ++ ")" else
show y

        show (Not x) = "~" ++ x_ where x_ = if length (show x) > 1 then "(" ++
show x ++ ")" else show x
        show (Or x y) = x_ ++ " || " ++ y_
            where x_ = if length (show x) > 2 then "(" ++ show x ++ ")" else
show x
                  y_ = if length (show y) > 2 then "(" ++ show y ++ ")" else
show y

        show (Imply x y) = x_ ++ " => " ++ y_
            where x_ = if length (show x) > 2 then "(" ++ show x ++ ")" else
show x
                  y_ = if length (show y) > 2 then "(" ++ show y ++ ")" else
show y

    p1, p2, p3 :: Prop
    p1 = And (Var 'A') (Not (Var 'A'))
    p2 = Or (Var 'A') (Not (Var 'A'))
    p3 = Imply (Var 'A') (And (Var 'A') (Var 'B'))

    -- Var :: Subst -> Prop -> Bool
    -- Var set x = if elem x (fst (unzip set)) then

    type Subst = [(Char, Bool)]
    eval :: Subst -> Prop -> Bool
    eval sub (Const x) = if x == True then True else False
    eval sub (Var x) = head [b | (a, b) <- sub, x == a]
    eval sub (And x y) = (eval sub x) && (eval sub y)
    eval sub (Or x y) = (eval sub x) || (eval sub y)
    eval sub (Not x) = not (eval sub x)
    eval sub (Imply x y) = if (eval sub x) == False then True else (eval sub y)

    vars :: Prop -> [Char]
    vars (Const x) = if x == True then "T" else "F"
    vars (Var x) = [x]
    vars (Not x) = vars x
    vars (And x y) = [t | t <- vars x] ++ [t | t <- vars y, not (elem t (vars
x))]
    vars (Or x y) = [t | t <- vars x] ++ [t | t <- vars y, not (elem t (vars
x))]
    vars (Imply x y) = [t | t <- vars x] ++ [t | t <- vars y, not (elem t (vars
x))]

    substs :: Prop -> [Subst]
```

```
    substs (Const x) = [[(t, x)]] where t = if x == True then 'T' else 'F'
    substs (Var x) = [[(x, True)], [(x, False)]]
    substs (Not x) = substs x
    -- substs tar = [substs x | x <- vars tar]
    substs tar = merge [substs (Var t) | t <- vars tar]
    -- substs (Or x y) = merge [head (substs (Var t)) | t <- vars (Or x y)]
    -- substs (Imply x y) = merge [head (substs (Var t)) | t <- vars (Imply x
y)]

    merge :: [[Subst]] -> [Subst]
    merge [] = []
    merge [x] = x
    merge (x : xs) = [a ++ [c] | a <- x, b <- merge xs, c <- b]

    isTaut :: Prop -> Bool
    isTaut func = not (elem False [eval sub func | sub <- substs func])
```

Expr (about Maybe)

```
    j2i :: Maybe Int -> Int
    j2i (Just a) = a

    type Subst = [(Char, Int)]
    eval :: Expr -> Subst -> Maybe Int
    eval (Const n) sub = Just n
    eval (Var x) [] = Nothing
    eval (Var x) (s : sub)
        | (fst s) == x = (Just (snd s))
        | otherwise = eval (Var x) sub
    eval (Add x y) sub
        | eval x sub == Nothing || eval y sub == Nothing = Nothing
        | otherwise = Just ((j2i (eval x sub)) + (j2i (eval y sub)))
    eval (Mul x y) sub
        | eval x sub == Nothing || eval y sub == Nothing = Nothing
        | otherwise = Just ((j2i (eval x sub)) * (j2i (eval y sub)))
    eval (Div x y) sub
        | eval x sub == Nothing || eval y sub == Nothing = Nothing
        | otherwise = Just ((j2i (eval x sub)) `div` (j2i (eval y sub)))
```