Переписывание

- 1 декабря на 4 паре, место будет уточнено позже

Задачи на листке

Задачи на листочке

- Тип (.)

```
sin . cos =
    \x -> sin (cos x)
```

```
(   ->   ) -> (   ->   ) -> (   ->   )
```

```
(b->c) -> (a->b) -> (a->c)
```

- (a->a)->a->a

```
func f x = f (f x)
func f x = f (f (f x))
```

Д.3.

allDiffLists

- Вариант 1:

```
allDiffLists n k = filter checkDifferent (allLists n k)
```

  - Очень неэффективно ☹

- Вариант 2:

Было:

```
allLists n 0 = [[]]
allLists n k = [x:xs | x<-[1..n], xs<-allLists n (k-1)]
```

```
allDiffLists n 0 = [[]]
allDiffLists n k = [x:xs | x<-[1..n], xs<-allDiffLists n (k-1),
                           not elem x xs]
```

  - elem – стандартная функция
  - Точно так же неэффективно ☹
    - Все равно получится, что перебираем все наборы
  - Надо бы как-то проверку *до* рекурсивного вызова

allDiffLists - продолжение

- Вариант 3:

```
allDiffLists n k = allDiffLists' n k []
```

- allDiffLists' n k s
  - s – те элементы, которые мы уже включили

```
allDiffLists' n 0 _ = [[]]
allDiffLists' n k s = [x:xs | x<-[1..n], not (elem x s),
                      allDiffLists' n k (x:s)]
```

  - Теперь эффективно!

- (Есть и другие хорошие решения)

allNondivisible

- прием "представление множества с помощью логической функции"

  - Что тут все-таки требовалось?
    Вместо списка, в который добавляем элементы –
    логическая функция, в которую добавляем новые условия

- [6,10,8,25,3]
  1. Сначала проверяем, что не делится для 6
  2. Потом проверяем, что не делится для 6 и 10
  3. Потом проверяем, что не делится для 6, 10 и 8
  4. Потом проверяем, что не делится для 6, 10, 8 и 25

allNondivisible - код

```
allNondivisible xs = allNondivisible' xs (\t -> False)

allNondivisible' [] _ = True

allNondivisible' (x:xs) cond =
    if cond x
    then False
    else allNondivisible' xs
            (\t -> cond t || mod x t ==0 || mod t x == 0)
```

- или можно короче

```
... = not cond x && allNondivisible' xs
            (\t -> cond t || mod x t ==0 || mod t x == 0)
```

triangle1, triangle2

triangle 3 □
[1, 1,4, 1,4,9]
i: 1 2 3

```
triangle1 n =
  [1..n] >>= \i ->          -- Для каждого i от 1 до n
    [1..i] >>= \j ->        --   Для каждого j от 1 до i
      return (j*j)          --     добавить в результат j*j

triangle2 n = do
  i <- [1..n]
  j <- [1..i]
  return (j*j)
```

Shape – типичные ошибки

```
contains (Circle r x y) a b = if (sqrt ((x-a)^2+(y-b)^2)) <= r)
                              then True
                              else False
```

- sqrt □
  (x-a)^2+(y-b)^2) <= r^2
- if …условие… then True else False
  □ …условие…
- лишние скобки

Shape

```
data Circle = Circle Double Double Double
data Rect = Rect Double Double Double Double

class Shape a where
   area:: a -> Double
   perim:: a -> Double
   contains:: a -> Double -> Double -> Bool

instance Shape Circle where
   contains (Circle r x0 y0) x y = (x-x0)^2+(y-y0)^2) <= r^2

instance Shape Rect where
   contains (Rect w h x0 y0) x y = abs(x-x0)<=w/2 && abs(y-y0)<=h/2
```

… и определения area и perim …

Дроби

```
data Ration  = Rat  Integer Integer

instance Ord Ration where
 Rat n1 d1 < Rat n2 d2 = n1*d2 < n2*d1
      --- Вспомните 6 класс! ---
 Rat n1 d1 < Rat n2 d2 = if d1*d2 > 0 then n1*d2 < n2*d1 else n1*d2 > n2*d1

instance Eq Ration where
 Rat n1 d1 == Rat n2 d2 = n1*d2 == n2*d1

instance Num Ration where
 Rat n1 d1 + Rat n2 d2 = (n1*d2 + n2*d1) / (d1*d2)

instance Show Ration where
  show (Rat n d) = show n ++ "/" ++ show d
```

Еще про классы

deriving

data Abc = … deriving Show

- Определить show автоматически (как-то)
- Еще
  - Ord
  - Eq
- Можно писать несколько классов
  data Abc = … deriving (Show, Ord, Eq)
- См. также *Datatype-generic programming*
  http://www.haskell.org/haskellwiki/Generics

Как сообщать о неудаче?

findSame – варианты решения

1. Число для сообщения об ошибке

   [1,2,3,2] ☐ 2

   [1,2,3] ☐ -1

   - Проблема: -1 может быть и "хорошим" ответом

   - На самом деле, на практике это вполне хороший подход, только возвращать лучше не -1, а что-то более странное:

   notFound = 2674386582678295 7

   [1,2,3] ☐ notFound

findSame- еще варианты

2. Возвращаем строку

[1,2,3] -> "Not found"

[1,2,3,2] □ "2"

- Ваш интерфейс не будет пользоваться успехом, он не очень удобный ☹
  - Наверняка ведь пользователь захочет с ответом еще что-то сделать (возвести в квадрат, например)
    - Придется парсировать, неудобно..

findSame – еще варианты

3. Вернуть пару (значение, код)

[1,2,3,2] ☐ (True, 2)
[1,2,3] ☐ (False, 0)

- Проблема: в списке могут быть и не числа, тогда 0 приведет к ошибке
  - Т.е. решение получается не generic
  - Как исправить, не очень понятно…

findSame- еще варианты

4. [] или [x]

[1,2,3,2] -> [2]
[1,2,3] -> []

5. Список *всех* повторяющихся

[1,2,3,2,3] -> [2,3]
[1,2,3] -> []

- Вопрос начальника: Разве мы не будем делать лишнюю работу? Я же просил *один* ответ.
  - Мы: Никакой лишней работы!
    - (Ленивые вычисления…)

Maybe

6. Специальный тип
   - Например: data Result a = Found a | NotFound

7. Есть стандартный тип, лучше использовать его:
   data Maybe a = Just a | Nothing

   [1,2,3,2] -> Just 2
   [1,2,3] -> Nothing

```
findSame [] = Nothing
findSame (x:xs) = if elem x xs
                  then Just x
                  else findSame xs
```

Failure continuations (продолжения при ошибке)

Очередной функциональный фокус…

find

- find условие список
  - Вернуть элемент, удовлетворяющий условию
  - Тоже проблема, как сообщить об ошибке

- Идея – в качестве параметра будем передавать *значение, которое надо возвращать при ошибке*

- Примеры

```
find (>0) xs (-1)
find odd xs 0
```

- Определение

```
find f [] err = err
find f (x:xs) err = if f x then x
                    else find f xs err
```

find с failure continuation

- А можно считать, что err – это *то, что надо сделать при ошибке*

- Пример:
  - Найти в xs число > 0 а если его нет, то найти число >0 в ys, а если и его нет, вернуть 0.

```
find (>0) xs
   (find (>0) ys 0)
```

  - Так можно, потому что ленивые вычисления
  - Получается, что find не совсем функция, а что-то вроде оператора:

```
find условие список
        код-который-надо-выполнить-если-не-нашли
```

- Называется *failure continuation* (продолжение при ошибке)

findSame с failure continuation

```
findsame [] err = err
findsame (x:xs) err =
   find (==x) xs
        (
          findsame xs err
        )
```

Мы бы могли написать так:

```
   findsame (x:xs) err = let
            res = find (==x) xs err
        in if res != err
        then res
        else findsame xs err
```

Но можно короче

else часть для поиска встроена прямо в find

- Написали findSame без if
  - Потому что find – это тоже что-то вроде if
  - Иногда удобно, но особого смысла нет, просто фокус

- Кстати, в вычислительных пакетах всегда было что-то похожее – например, при решении системы уравнений передаем функцию, которую надо вызвать, если у системы нет решений

Еще пример

- Найти первое число, большее 1000, а если его нет, то первое число большее 500, а если и его нет, то первое число большее 100 (а если и его нет, вернуть 0):

```
find (>1000) xs
   (find (>500) xs
       (find (>100) xs 0)
```

К следующему д.з.: Комбинируем функции

Как вернуть позицию в списке?

- Когда мы ищем что-то в списке, хотелось бы вернуть не просто значение, а еще как-то и то место, на котором мы его нашли.

Например, хотелось бы написать find, чтобы его можно было вызвать три раза и найти третий элемент в списке, удовлетворяющий данному условию

- Какой должен быть интерфейс у такого find?

- Распостраненный вариант: возвращать пару
  (значение, еще не просмотренный хвост списка)

find, который возвращает пару

- В этой теме давайте для простоты временно считать, что мы всегда точно найдем элемент, удовлетворяющий условию.

Тогда find можно написать так:

```
find cond (x:xs) =
    if cond x
    then (x, xs)
    else find cond xs
```

Пример вызова:

```
find (>3) [1, 3, 5, 2, 20, 25, 2]
```

- (5, [2, 20, 25, 2])

Моя мечта – композиция для таких функций

- Пусть я хочу как-то сочетать функции, которые берут список и возвращают пару *(значение, список)*
  - Примерно так же, как это делает композиция

    sin . cos
  - То есть я бы хотел писать так:
    f = find (>3) . find (>5)
    чтобы получилась функция, которая сначала ищет число больше 3 а потом, с того места где остановился первый поиск – число больше 5

Но только (.) тут, конечно не подходит :(

Задание на дом: >>>

Давайте напишем что-то похожее на композицию, но для функций вида

список -> (результат, список)

- Т.е. задача (для д.з.):
Определить такой оператор, назовем его >>>, чтобы можно было писать так:

```
f = find (>3) >>> find (>5)
   -- f - функция, которая ищет в списке элемент, больший 3,
   -- а потом, с этого места, элемент больший 5.

f [1, 3, 5, 2, 20, 25, 2]
   -- Должно получиться (20, [25, 2])>
```

К следующему д.з.: Символьные вычисления

Как представлять выражения, чтобы можно обрабатывать в программе

- Пока будем рассматривать выражения, которые состоят из целых чисел, переменной X и операции сложения и умножения

```
data Expr = Num Integer | X | Add Expr Expr | Mult Expr Expr
                          deriving Show
```

  - или м.б. deriving (Show, Eq)

- Примеры:

```
Add X  (Num 1)
```

  - Так мы записываем x+1

- Как записать x*(1+x*x) ?

```
Mult X (Add (Num 1) (Mult X X))
```

Про некоторые доп. задачи

Lst367 на C#

```
static IEnumberable<int> Lst367()
{
    yield return 3;
    yield return 6;
    yield return 7;
    foreach (var i in Lst367()) {
        yield return 10*i + 3;
        yield return 10*i + 6;
        yield return 10*i + 7;
    }
}
```

countDifferentVars

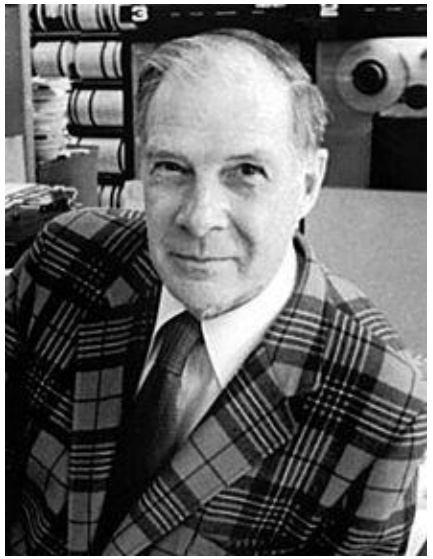
1. Понять, какие переменные на самом деле разные, а какие одинаковые
2. Посчитать разные переменные во втором параметре

- Представление данных?
  - Список списков
  - Список пар
    - Disjoint Set Structure?

http://en.wikipedia.org/wiki/Disjoint-set_data_structure

ham

- Richard Hamming: 2^i * 3^j * 5^k



1, 3, 9, 10, 27, 30, 81, 90, 100, 243, …

map (*3) ham ☐
  3, 9, 27, 30, 81, 90, 243, …
  - Не хватает только 1, 10, 100, 1000, …
  - Как добавить?
    - merge!

```
ham = merge (map (*3) ham) ([10^n | i<-[0..] ])
```

- Или, та же идея, но другой вариант:

```
ham = 1 : merge (map (*3) ham) (map (*10) ham)
```