На листке

Одна известная последовательность

Решето Эратосфена

- Решето:

filter (\t -> t `mod` x /= 0)

Описать sieve:

список □ список

- первое число переносит в результат
- просеивает по первому числу
- и рекурсия

primes = sieve [2..]

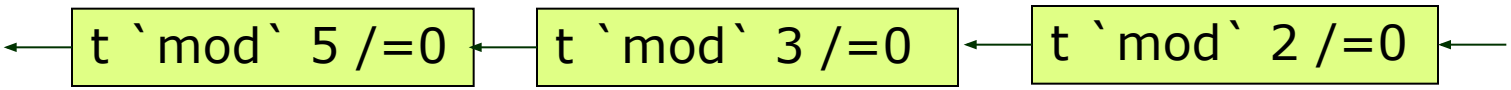
	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

Prime numbers

Решение

```
sieve (x:xs) =
    x:
       sieve
          (filter (\t -> t `mod` x /= 0) xs)

primes = sieve [2..]
```



Придумал Douglas McIlroy, 1968
http://www.cs.dartmouth.edu/~doug/sieve

Д.з.

Тип foldr

foldr f e (x:xs) = f x (foldr f e xs)
foldr f e [] = e

x1 -> x2 -> x3 -> x4

- Этап 1: Выясняем, что некоторые x на самом деле – сложные типы

x1 = (x5->x6->x7)

- Потому что у **f** тип x1 и f – это функция (видно из **f x (foldr…)**)

x3 = [x8]

- Потому что у **(x:xs)** тип x3

- Т.е. получили:

(x5->x6->x7) -> x2 -> [x8] -> x4

- Этап 2: Выясняем, что некоторые x, на самом деле, совпадают

x2 == x4

- Потому что у **e** тип x4 (так как это аргумент в **foldr**) и тип x5 (так как это результат **foldr**)

x7 == x4

- Потому что у **f x (foldr…)** тип x7 (так как это результат f) и тип x5 (так как это результат **foldr**)

x6 == x4

- Потому что у **(foldr f e xs)** тип x4 (так как это результат fodlr) и тип x7 (так как это парамер **f x (foldr…)**)

Тип foldr - продолжение

x5 == x8

- Потому что у **x** тип x5 (так как это аргумент в **f x ...**) и тип x8 (так как выражение **(x:xs)** имеет тип x8

Итого получаем:

(x5->x4->x4)->x4->[x5]->x4

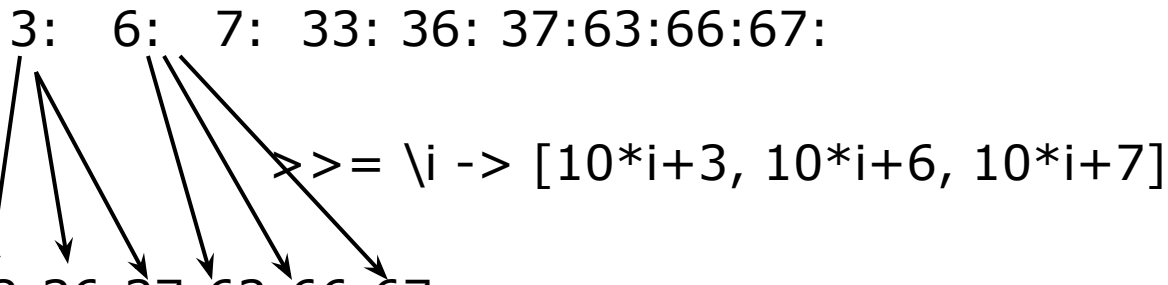
Или, с более обычными именами:
(a->b->b)->b->[a]->b

Еще про >>=.
do нотация

doubleOdd

```
doubleOdd xs = xs >>= \x ->     -- для всех элементов x из xs …
    if x `mod` 2 == 1
    then [x,x]
    else [x]
```

lst367



lst367 = 3:6:7: (lst367 >>= \i -> [10*i+3, 10*i+6, 10*i+7])

или

lst367 = 0 : lst367 >>= \i -> [10*i+3, 10*i+6, 10*i+7]

или

lst367 = 3:6:7:[10*i+j | i <- lst367, j <- [3,6,7]]

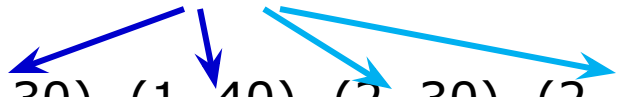
Декарт

Cogito ergo sum

Кристина, королева Швеции

cartesian

cartesian [1, 2] [30, 40] □



[(1,30), (1, 40), (2, 30), (2, 40)]

caretsian xs ys = xs >>= \x ->

приписать x ко всем элементам ys

1□ [(1,30),(1,40)]

2->[(2,30),(2,40)]

- Как приписать x ко всем элементам ys?
  - Можно map
  - Красивее еще раз >>=

    ys >>= \y->

    [(x, y)]
- Итого

cartesian xs ys = xs >>= \x -> ys >>= \y -> [(x, y)]

return

- Есть стандартная функция return
  return x= [x]

```
cartesian xs ys = xs >>= \x -> ys >>= \y -> return (x, y)
```

- Зачем?
  - Тогда можно определить >>= и return и для других типов и писать в таком стиле не только для списков
  - И это будет называться «monadic style»

do нотация

```
cartesian xs ys =
    xs >>= \x ->
       ys >>= \y ->
          return (x, y)
```

- **xs >>= \x ->** можно понимать как "Для каждого x из списка xs …"

- Есть сокращенная запись:

```
do x <- xs
   y <- ys
   return (x,y)
```

- То есть автоматически преобразуется:

```
x <- xs
   □
xs >>= \x ->
```

- Двумерный синтаксис, как в let

Классы

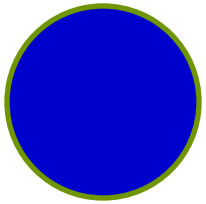
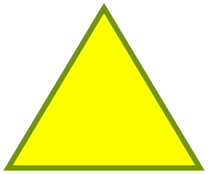
Какой тип у sort?

- sort [3,1,2,4] ☐ sort [1,2,3,4]
  - [Int] -> [Int] ?
- Но м.б. sort "apple" ☐ "aaelpp"
  - [a] -> [a] ?
- Но списки функций, например, мы сортировать не можем
  - [a] -> [a] если мы умеем сравнивать a – как это сказать?

- sum [1,2,3] ☐ 6
- sum [2.71, 3.14, 1.41] ☐ [7.26]
  - [a] -> a, если мы умеем складывать a

- Как сочетать generic и типы
  - В обычных языках по разному (м.б. потом обсудим)
  - Например, в C++ generic (templates) типы, в общем то не используются

Фигуры

- тип "Прямоугольник"

```
data Rect = Rect Double Double

area (Rect x y) = x*y

perim (Rect x y) = 2*(x+y)
```

- тип "Круг"

```
data Circle = Circle Double

area (Circle r) = 3.14*r*r

perim(Circle r) = 2*3.14*r
```

- Ошибка компиляции!
  - Какой тип area?

Классы

- Описываем то, что должны уметь все фигуры

```
class Shape a where
    area:: a -> Double
    perim:: a -> Double
```

- Описываем Rect, как реализацию Shape

```
instance Shape Rect where
    area (Rect x y) = x*y
    perim (Rect x y) = 2*(x+y)
```

- И то же Circle

```
instance Shape Circle where
    area (Circle r) = 3.14*r*r
    perim (Circle r) = 2*3.14*r
```

- Теперь все компилируется!

Классы – продолжение 1

- Класс указывается в типе функции:
  У area тип
  Shape a => a -> Double

- Можно определять функции для всех фигур

```
f x = area x / perim x
```

  - Полиморфизм

- Разумные сообщения об ошибках
  area "abc"
  - Сообщение вроде
    "String не принадлежит классу Shape"
    - Не то что в C++ ☹

Классы - продолжение

- Это, конечно, похоже на классы обычных языков
  - Но не совсем

- Класс не обязательно в параметре

```
class Shape a where
    …
    createSample:: Double -> a

instance Shape Rect where
    …
    createSample x =
            Rect x (1.6*x)

instance Shape Circle where
    …
    createSample x = Circle x
```

- Любые функции

  inflate:: a -> Double -> a

  areDisjoint:: [a] -> Bool

  … и т.д. …

Стандартные классы

Комплексные числа

```
data Complex = C Double Double

(C re1 im1) + (C re2 im2) =
        C (re1+re2) (im1+im2)
```

- Так не скомпилируется

- Стандартный класс Num
  - +
  - -
  - *

```
instance Num Complex where
   (C re1 im1) + (C re2 im2) =
        C (re1+re2) (im1+im2)
```

- И сразу получаем возможность использовать все, что написано для Num!

```
sum [C 3 6, C 1 0, C 2 2] □
    C 6 8
```

- Тип sum

```
sum :: Num a => [a] -> a
```

Еще стандартные классы

- Ord
  - <, <=, >, >=
- Eq
  - =, /=
- Show
  - show

```
instance Eq Complex where
    C re1 im1 == C re2 im2 =
      re1 == re2 &&
      im1 == im2
```

```
instance Show Complex where
    show (C re im) =
        show re ++ "+" ++
        show im ++ "*i"
```

Прием «Представление множества с помощью логической функции»

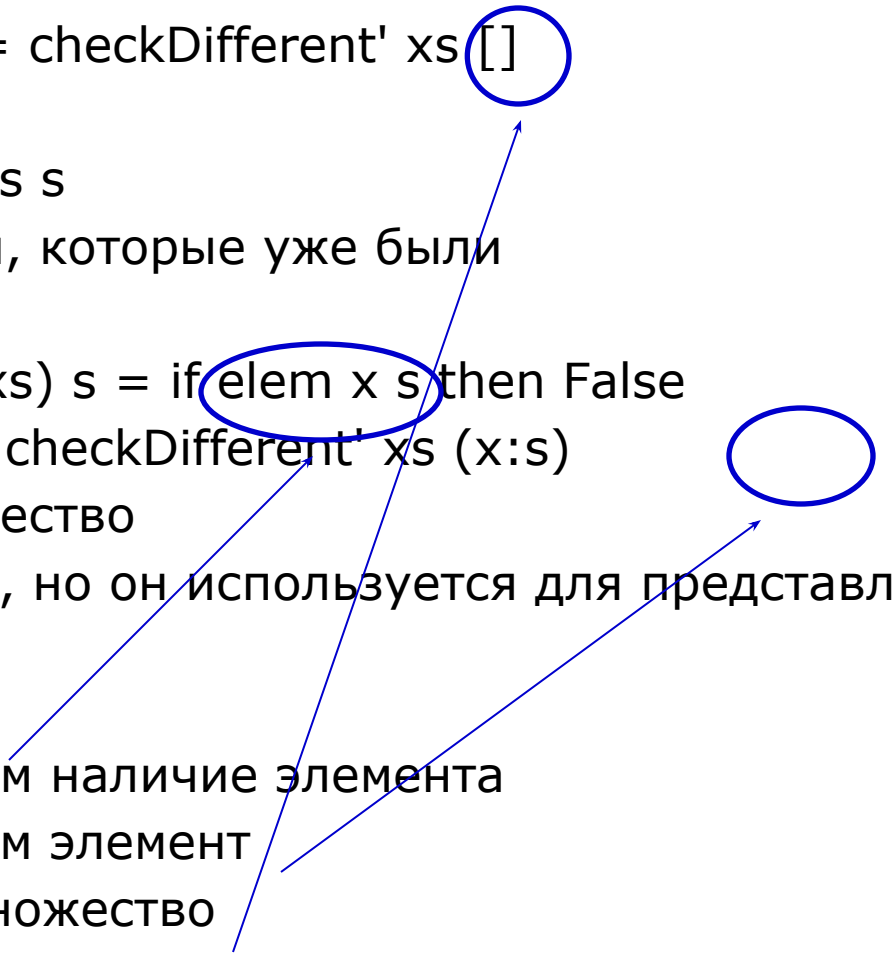
Снова checkDifferent

```
checkDiffferent xs = checkDifferent' xs []
```

- checkDifferent' xs s
  - s – элементы, которые уже были

```
checkDiffenent' (x:xs) s = if elem x s then False
                           else checkDifferent' xs (x:s)
```

- s – как бы множество
  - Т.е. s список, но он используется для представление множества
  - Операции:
    - Проверяем наличие элемента
    - Добавляем элемент
    - Пустое множество



Как еще можно представить множество?

- Можно переделать для другого представления данных:
  - Tree
  - Data.Set
  - функция, которая проверяет, было ли уже значение, или нет. (характеристическая функция)

{1,2,3} - представляем, как
\t -> t == 1 || t == 2 || t == 3
{1..1000} – представляем, как
\t -> t >= 1 && t <= 1000
пустое множество – представляем, как
\t -> False

- или
  const False

Решение с помощью характеристической функции

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

Пустое множество

- checkDifferent' xs cond
  - cond – множество уже просмотренных чисел, представленное, как функция

Проверка, было ли число

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

Добавить в условие еще одну проверку

Пример работы

checkDifferent [2,3,5,3,8] □
  checkDifferent’ [2,3,5,3,8] (\t -> False) □
    checkDifferent’ [3,5,3,8] (\t -> t == 2) □
      checkDifferent’ [5,3,8] (\t -> t == 2 || t == 3) □
        checkDifferent’ [3,8] (\t -> t == 2 || t == 3 || t == 5) □
          False

Про некоторые доп. задачи

cantor

- По диагоналям

```
[(x, y) | sum <- [2..], x <- [1..sum-1], let y = sum – x]
```

generalizedCantor

- Мне кажется, Кантору бы оно понравилось вот это:

- Для простоты будем рассматривать пары с 0 тоже

```
cantorList = [[x, y] | s <- [2..], x <- [1..s-1], let y = s – x]
cantorFunction k = cantorList !! (k-1)
```

  - число □ пара чисел
  - или могли бы прямо тут выписать функцию, которую нашел Кантор

```
generalizedCantor 2 = cantorList
generalizedCantor n = [x1:x2:xs | (x:xs) <- generalizedCantor (n-1),
                                  let [x1, x2]  = cantorFunction x]
```

zeroDigits

- zeroDigits(a, 2)
  563, 5643, 76796 □ 500, 5600, 76700

```
static IEnumerable<int> ZeroDigits(IEnumerable<int>a, int n)
{
    return a.Select(x => x - x % (int) Math.Pow(10,n));
```

- Лишняя работа!
  - Вычисляется для каждого элемента массива!

```
    int m = Math.Pow(10, n);
    return a.Select(x => x – x % m);
}
```

- Переменные замыкания лучше вычислять заранее!

pascal

```
pascal = [[1],
          [1,1],
          [1,2,1],
          [1,3,3,1], …

pascal = [1] : map getNext pascal
getNext xs =
    [1] ++
    zipWith (+) xs (tail xs)
    ++ [1]
```

- Или, без вспомогательной функции:

```
pascal =
    [1] : map (\xs -> [1] ++zipWith (+) xs (tail xs)++ [1]) pascal
```

Задачи на листках

Эти задачи только для тех, кто был на занятии. Но, может остальным интересно просто почитать.

1. Какой тип у оператора (.) (композиции)?

2. Опишите функцию, имеющую тип (a->a)->a->a
(если получиться, опишите, пожалуйста, два примера таких функций).

Те, кто был на занятии, но не решил задачу 2, могут прислать ее решение по почте, и получить еще 1 балл.