1 Давайте поговорим о паттерн матчинге на листах

1.1 Введение

1.1.1 Определения

Тотальная функция - определена на всей области возможных значений
Нетотальная - на каких-то значениях падает.
Давайте теперь рассмотрим примеры функций на списках, доступных в хаскеле, и приведем их реализации на паттрен матчинге

1.2 Стандартные функции на листах

1.2.1 Функция head

Пример - функция *head*, которая возвращает голову листа, упадет с ошибкой при передачи пустого листа. Т.е. она нетотальрная

```
1 head :: [a] -> a
2 head (x:_) = x
3 head _ = error "empty list"
```

1.2.2 Функция tail

Еще есть функция *tail* - она вообще говоря в *Prelude* падает на пустом списке, т.е. она нетотальна.
Но мы можем реализовать ее посвоему. Если нет даже головы или есть элемент всего один,то результат один и тот же - пустой список. Т.е. можно определить тотальную функцию *totalTail*

```
1 totalTail :: [a] -> [a]
2 totalTail (_: xs) = xs
3 totalTail _ = []
```

1.2.3 Функция take

Еще есть функция *take*. Она возвращает какое-то (заданное) количество элементов

```
1 take :: Int -> [a] -> [a]
2 take 0 _ = []
3 take n (x: xs) = x : take (n-1) xs
4 take _ [] = []
```

1.2.4 Функция map

Преобазует $a->b$ при помощи функции

```
1 map :: (a -> b) -> [a] -> [b]
2 map _ [] = []
3 map f (x: xs) = f x : map f xs
```

1.2.5 Функция filter

Фильтрует значения из списка
Вариант №1 (обычный)

```
1 filter :: (a -> Bool) -> [a] -> [a]
2 filter _ [] = []
3 filter p (x:xs) =
4     if p x
5     then x : filter p xs
6     else filter p xs
```

Вариант 2 (модный)

```
1 filter :: (a -> Bool) [a] -> [a]
2 filter _ [] = []
3 filter p (x: xs)
4     | p x       =   x : rest
5     | otherwise =       rest
6     where
7         rest = filter p xs
```

1.2.6 Функция zip

Которая объединяет 2 списка в список пар

```
zip :: [a] -> [b] -> [(a,b)]
zip _      [] = []
zip []     _  = []
zip (x:xs) (y:ys) = (x,y): zip xs ys
```

1.2.7 Функция zipWith

Которая объединяет 2 списка в третий при помощи вашей любой функции

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ _ [] = []
zipWith _ [] _ = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

Это упражнение можно продолжать до zipWith3, например, которая объеденит уже 3

```
zipWith3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
zipWith3 _ _ _ [] = []
zipWith3 _ _ [] _ = []
zipWith3 _ [] _ _ = []
zipWith3 f (x:xs) (y:ys) (z:zs) = f x y z : zipWith3 f xs ys zs
```

А дальше можно продолжать в принципе до бесконечности

1.2.8 Функции takeWhile и dropWhile

takeWhile берет элементы, пока не найдет первый false, как найден -> возвращает все элементы для этого
dropWhile наоборот дропает, пока не найдет первый false, дальше возвращает весь остаток (+ тот элемент, который первый "не подошел")
В хаскеле можно определять одинковые сигнатуры вот так:

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

dropWhile _ [] = []
dropWhile p (x:xs)
  | p x       = dropWhile p xs
  | otherwise = x : xs
```

1.2.9 Функция свертки она же fold

Свертка - это взять все элементы некого листа и каким-то образом их объеденить (т.е. "свернуть"все элементы в 1 элемент).
Сворачивать в общем-то можно слева направо или справа налево. Для + например, разницы нет. А вот для умножения квадратных матриц очень даже влияет.
В примере ниже сверху левая свертка, снизу правая свертка.

$$((1+2)+3)+4$$
$$1+(2+(3+4))$$

А вот и определения сверток

```
foldl :: (acc -> b -> acc) -> acc -> [b] -> acc
foldr :: (b -> acc -> acc) -> acc -> [b] -> acc

foldl _ acc []     =  acc
foldl f acc (x:xs) =  foldl f (f acc x) xs

foldr _ acc []     =  acc
foldr f acc (x:xs) =  f x (foldr f acc xs)
```

На основе сверток (и на основе того, что в хаскеле из сигнатуры можно вернуть любую часть этой сигнатуры) можно очень коротко писать всякие другие классные функции. Например сумму и умножение

```
sum     = foldl (+) 0
product = foldl (*) 1
```

1.2.10 Бесконечный поток чисел Фибоначчи!

Над следующим пунктом можно помедетировать. Это бесконечный список Фибоначчи. Его бесконечность нас не сильно волнует, так как в хаскеле все ленивое.

```
fibs :: [Int]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

1.2.11 Конкатенация списков

Конкатенация делается вообще чудесно

```
(++) :: [a] -> [a] -> [a]
(x:xs) ++ ys = x : (xs ++ ys)
[]     ++ ys =             ys
```

Тут написано, что мы в первом матчинге складываем первый элемент первого списка с остатком первого списка и полный вторым списком.
А во втором матчинге, если первый список закончился - просто добавляем сзади весь второй список.

1.3 Всякое разное

1.3.1 Оператор (.)

Есть такое оператор со следующей сигнатурой

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

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

pipe :: (b -> c) -> (a -> b) -> [a] -> [a]
pipe f = map f . map g
```

Последние две строчки - это η **редукция** (читается как ета-редукция). Такая нотация еще называется "бесточечная хотя в хаскеле здесь как раз-таки и используется точка...
Это напоминает суперпозицию функций из математики.

1.3.2 Задачка про воду

Есть некоторая местность. Её рельеф задан высотой столбиков. Идет дождь. Дождь прошел, а там где какие-то ямы образовались лужи. Нужно посчитать, сколько туда накапало воды...
Давайте её решим

```

r = [1, 5, 3, 8, 3, 2, 3, 7, 4]

-
-----
---**
--------
---****
--*****
---****
-------
----
```

Звездочками отмечено, где есть вода. Палочками - высота рельефа.
Есть такие функции *scanl* и *snanr*. Это как *fold*, только он возвращает значение аккумулятора на каждом из шагов

```
 main = print $ scanl (+) 0 [1,2,3,4,5] -- yeild us [0,1,3,6,10,15]
```

Мы можем сделать 2 скана. 1 слева на максимум. Другой справа на максимум.

```
r = [1, 5, 3, 8, 3, 2, 3, 7, 4]
main = do
  print $ r
  print $ tail $ scanl max 0 r
  print $ init $ scanr max 0 r

[1, 5, 3, 8, 3, 2, 3, 7, 4]
[1, 5, 5, 8, 8, 8, 8, 8, 8]
[8, 8, 8, 8, 7, 7, 7, 7, 4]
```

Значение сверху - это "дно". Значения снизу (2 ряда) - это одна и вторая стенка и её "тень". По этим трем массивам видно, что нужно вычесть значение от минимума из двух стенок значение "дна"(По выводу видно, что что-то отличное от нуля у нас будет в позиции 3, а также 5,6,7).
Т.е. у нас следующий алгорим

1. Вычислить границы, считая с левой стороны (т.е. запомнить последний максимум и повторять его, пока не найдем еще 1 больший максимум). Это будет что-то вроде "тени"
2. Вычислить границы, считая с правой стороны
3. Берем минимум от двух пар из пунктов 1 и 2
4. Вычитаем из значение этого минимума значение в точке
5. Складываем сумму этих результатов

При реализации на хаскеле все будет записао как бы наоборот

```
1  r = [1, 5, 3, 8, 3, 2, 3, 7, 4]
2  main = do
3      print r
4      print x
5      where
6          x = foldl (+) 0 ws
7          ws = zipWith (-) bs r
8          bs = zipWith min bl br
9          bl = tail $ scanl max 0 r
10         br = init $ scanr max 0 r
```