

Сколько задач надо решить, чтобы понять монады?



Михаил Симуни
simuni@mail.ru

Замечания

- Почти только задачи и почти все *очень* простые
 - Вообще то я думаю, что за час про монады рассказать нельзя :(
- (Примерно как объяснить за час интегралы)
- Если непонятно, спрашивайте!

Handwritten mathematical notes on a chalkboard:

- $\iint xy(x^2+y^2)^5 dx dy$,
of the circle $x^2+y^2 = R^2$
- i) $\int_0^{2a} \int_{-\sqrt{2ax-x^2}}^{\sqrt{2ax-x^2}} (x^2+y^2) dx dy$
- ii) $\int_0^\infty \int_0^\infty e^{-(x^2+y^2)} dx dy$
- use the result in part (iii) +

FOTOFSEARCH®

K5251537 www.fotosearch.com ©

Что желательно знать про Haskell



Композиция, карринг, секции, map

Композиция

- ▣ `sin . cos` - сокращение для `\x -> sin (cos x)`

Карринг – задаем *часть* параметров

- ▣ `f x y`
`f 1` - сокращение для `\x -> f 1 x`

Секция

- ▣ `(2+)` – сокращение для `\x -> 2 + x`

map

- ▣ `map sin [1, 2, 3] → [sin 1, sin 2, sin 3]`

Композиция, карринг, секции, map

Композиция

- ▣ `sin . cos` - сокращение для `\x -> sin (cos x)`

Карринг – задаем *часть* параметров

- ▣ `f x y`
`f 1` ← - сокращение для `\x -> f 1 x`

Строго говоря, это
не сокращение

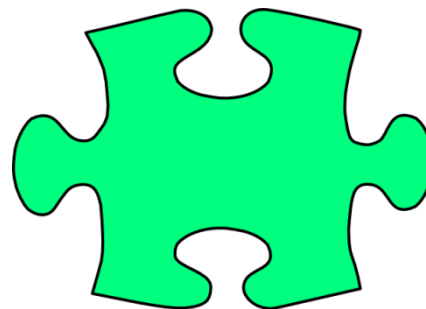
Секция

- ▣ `(2+)` – сокращение для `\x -> 2 + x`

map

- ▣ `map sin [1, 2, 3] → [sin 1, sin 2, sin 3]`

Задачи про КОМПОЗИЦИЮ



Простые примеры

- ▣ Найти сумму синусов для списка

```
f xs = sum (map sin xs)
```

Как это можно записать короче?

```
f = sum . map sin
```

- ▣ $f\ n = 10 * n + 7$

Как это можно записать короче?

Простые примеры

- ▣ Найти сумму синусов для списка

`f xs = sum (map sin xs)`

Как это можно записать короче?

`f = sum . map sin`

- ▣ `f n = 10 * n + 7`

Как это можно записать короче?

`f = (+7).(*10)`

Пример: функции, дописывающие в список

- ▣ `add n x xs` - добавить в начало списка `xs` `n` чисел `x`.

Например, `add 3 7 xs`

– дописать `[7,7,7` в начало `xs`

`add 0 x xs = xs`

`add n x xs = x: add (n-1) x xs`

Называется *difference list*

Оказывается, такие функции часто удобнее, чем функции, которые создают список.

Пример: функции, дописывающие в список

- ▣ `add n x xs` - добавить в начало списка `xs` `n` чисел `x`.

Например, `add 3 7 xs`

– дописать `[7,7,7` в начало `xs`

`add 0 x xs = xs`

`add n x xs = x: add (n-1) x xs`

Оказывается, такие функции часто удобнее, чем функции, которые создают список.

Называется *difference list*

Задачи на дом:)

- ▣ Числа в обратном порядке без использования `++`
- ▣ Для дерева построить список вершин, без использования `++`
- ▣ * числа в обратном порядке с помощью `foldr`

Композиция для функций, дописывающих в список

Почему такие функции удобные?

Одна из причин – удобно применять композицию

▣ `f xs` - дописать сначала 10 раз 1, потом 10 раз 2, потом 10 раз 3

```
f n xs = let xs1 = f 10 1 xs
          xs2 = f 10 2 xs
          xs3 = f 10 3 xs
          in xs3
```

Как записать короче?

Композиция для функций, дописывающих в список

Почему такие функции удобные?

Одна из причин – удобно применять композицию

- `f xs` - дописать сначала 10 раз 1, потом 10 раз 2, потом 10 раз 3

```
f n xs = let xs1 = f 10 1 xs
          xs2 = f 10 2 xs
          xs3 = f 10 3 xs
          in xs3
```

Как записать короче?

```
f = (f 10 3).(f 10 2).(f 10 1)
```

Поиск в списке

- Начальник попросил нас написать функцию `find`, которая по списку и условию ищет элемент в списке.

`find (>3) [1, 4, -7, 8, -2, 2] → 4`

- Потом добавил условие: надо, чтобы можно было, например, найти, первое число, большее 3, и после него первое число, большее 5



Поиск в списке

- Начальник попросил нас написать функцию `find`, которая по списку и условию ищет элемент в списке.

```
find (>3) [1, 4, -7, 8, -2, 2] → 4
```

- Потом добавил условие: надо, чтобы можно было, например, найти, первое число, большее 3, и после него первое число, большее 5

Типичное решение: возвращать пару
(*найденный элемент, хвост*)

```
find (>3) [1,4,7,8,1] → (4, [7,8,1])
```

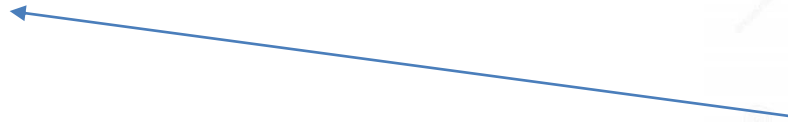


Что
возвращать,
если такого
элемента нет?
Это мы обсудим
позже.

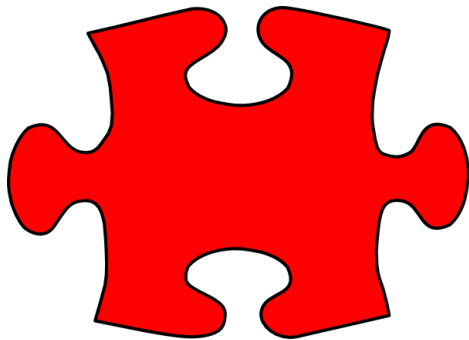
Вопрос

Придумать какой-нибудь способ соединять такие find, примерно как мы соединяли add.

`f = find (>3) ??? find (>5)`



Задачи про тар и т.д.



map в чем то похож на foreach в обычных языках

▣ `f xs` – заменить в списке все 2 на 5

```
map (\x ->
    if x == 2
    then 5
    else x
) xs
```

Что то вроде заголовка цикла:
«для каждого элемента xs...»

Что то вроде тела цикла

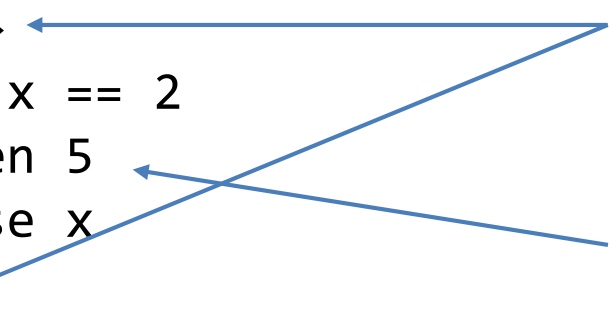


Таблица умножения

- `multTable n ->` список `n` списков по `n` элементов

```
multTable 3 = [[1,2,3],  
               [2,4,6],  
               [3,6,9]]
```

```
multTable n =
```

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

Доп. задача –
«крестик»

```
cross 4 =  
  [[1,0,0,1],  
   [0,1,1,0],  
   [0,1,1,0],  
   [1,0,0,1]]
```

Цепочки вложенных функций высшего порядка

```
map (\i ->  
    map (\j ->  
        ...
```

Такие вложенные цепочки из функций высшего порядка и лямбда выражений – это, в общем то, почти монады.



Цепочки вложенных функций высшего порядка

```
map (\i ->  
    map (\j ->  
        ...
```



Такие вложенные цепочки из функций высшего порядка и лямбда выражений – это, в общем то, почти монады.

«Для любого, самого сложного понятия, всегда существует объяснение простое, понятное и неправильное»

Н. Л. Mencken (переделанная цитата)

supermap

Написать supermap – почти такой же, как обычный map, но чтобы он умел заменять элемент на **несколько** элементов.

```
supermap (\x -> [sin x, cos x]) [1,2,3] →  
    [sin 1, cos 1, sin 2, cos 2, sin 3, cos 3]
```

Такая функция уже есть, и это...

>>=

>>=

- читается "bind"

```
f xs = xs >>= \x -> [sin x, cos x]
```

- Пример: в списке раздвоить все положительные числа и убрать все остальные

```
f xs = xs >>= \x ->  
    if x > 0  
    then [x, x]  
    else []
```

Что то вроде заголовка цикла:
«для каждого элемента xs...»

Что то вроде тела цикла

Вложенные $>>=$

- ▣ `decartes xs ys` – декартово произведение двух списков



Вложенные $>>=$

- ▣ `decartes xs ys` – декартово произведение двух списков

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

- ▣ $>>=$ как бы связывает `x` и `xs`

```
decartes xs ys =  
  xs >>= \x ->
```



return и do нотация

▣ `return x = [x]`

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

Зачем??

return и do нотация

□ `return x = [x]`

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

□ `do` нотация

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

Зачем??

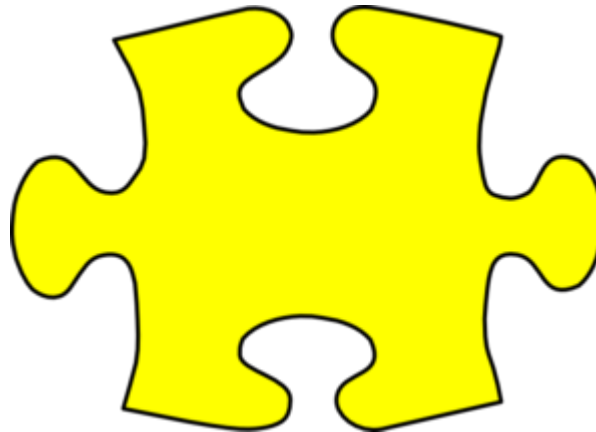
Абстракция!

Так у нас нет никаких упоминания о том, что мы вообще работаем со списками

Простая задача:

С помощью `>>=` и `return` и с помощью `do` нотации создать список всех троек чисел от 1 до `n`, сумма которых `<= n`.

Задачи про неудачу



Как сообщить о неудаче?

Начальник: Я хочу, чтобы find сообщал о том, что функция ничего не нашла.

Я: А как сообщать?

Начальник: Все равно как, но после вызова пользователь должен как-то понять, нашел он что-то или нет.



Как сообщить о неудаче?

Начальник: Я хочу, чтобы find сообщал о том, что функция ничего не нашла.

Я: А как сообщать?

Начальник: Все равно как, но после вызова пользователь должен как-то понять, нашел он что-то или нет.



Варианты решения:

- ❑ ~~x или error "Ошибка"~~
- ❑ ~~x или -1~~
- ❑ Just x или Nothing (тип Maybe)
- ❑ [x] или []
- ❑ ...

Еще интересный вариант, кстати - failure continuation

Задача о трех поисках

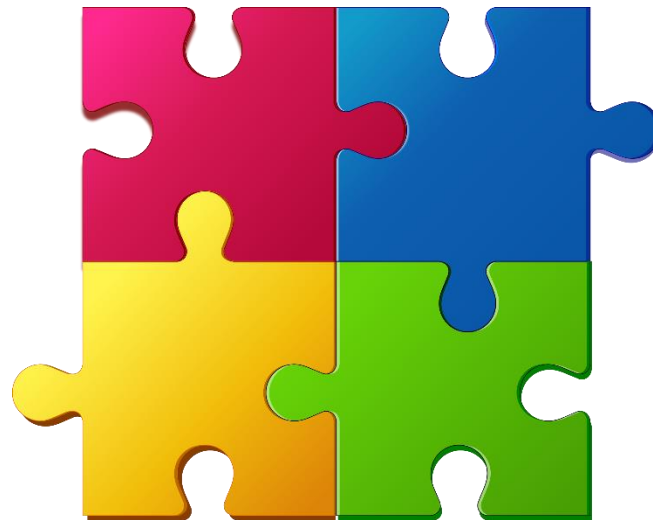
Пусть у нас уже написан `find`, который возвращает `[x]` или `[]`. Я хочу в списке найти:

- ▣ Первое число, большее 0
- ▣ Первое число, меньшее 3
- ▣ Первое число, не равное 1

И хочу вернуть

- ▣ *[сумма этих чисел]*
- ▣ или `[]` если хотя бы один из поисков завершился неудачей.

Соединяем все вместе



Решение

```
f xs = do
  x <- find (>0) xs
  y <- find (<3) xs
  z <- find (/=1) xs
  return (x+y+z)
```

Тут >>= обеспечивает **выполнение до первой неудачи**

То же для Maybe

Для Maybe (т.е. Just x или Nothing) определены

- ▣ >>= - смысл, как для [x] и [] - «выполнять до первой неудачи»
- ▣ return - return x = Just x

Задача (она же балл в подарок): Пусть у нас уже написан find, который возвращает Just x или Nothing. Я хочу в списке найти:

- ▣ Первое число, большее 0
- ▣ Первое число, меньшее 3
- ▣ Первое число, не равное 1

И хочу вернуть:

- ▣ Just *сумма этих чисел*
- ▣ или Nothing если хотя бы один из поисков завершился неудачей.

Решение

```
f xs = do
  x <- find (>0) xs
  y <- find (<3) xs
  z <- find (/=1) xs
  return (x+y+z)
```

Что такое монады, формально

Монада – это тип, для которого определены операции

- $>>=$
- `return`
- Уже знаем два примера монад:
 - `List`
 - `Maybe`

Операции еще должны удовлетворять некоторым правилам (Monadic laws):

$$\text{return } a \gg= f \equiv f a$$
$$m \gg= \text{return} \equiv m$$
$$(m \gg= f) \gg= g \equiv m \gg= (\lambda x \rightarrow f x \gg= g)$$

Монады – это про слово «ПОТОМ»

```
f xs = do
```

```
  x <- find (<5) xs
```

```
  y <- find (>10) xs
```

```
  z <- find (/=7) xs
```

```
  return (x+y+z)
```

Вычислить find (<5) xs, **ПОТОМ**
вычислить find (>10) xs, **ПОТОМ**
вычислить find (/=7) xs

И, кроме этого выполнять
дополнительные действия
(проверять, удачно ли завершились
вызовы)

Т.е. мы описываем некоторые действия

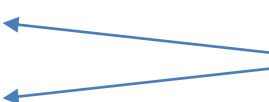
- Которые должны выполняться одно за другим
- И при этом должно автоматически происходить что-то дополнительное

«Программируемая точка с запятой»

Примерно как если бы в обычном языке мы могли бы переопределить точку с запятой

```
x = f(1);  
y = g(x);  
z = x + y;
```

И тут автоматически еще
что-то происходит



Монады обеспечивают абстракцию вычисления

```
f xs = do
  x <- find (<5) xs
  y <- find (>10) xs
  z <- find (/=7) xs
  return (x+y+z)
```

Что делает эта функция?

Монады обеспечивают абстракцию вычисления

```
f xs = do
  x <- find (<5) xs
  y <- find (>10) xs
  z <- find (/=7) xs
  return (x+y+z)
```

Что делает эта функция?

- ❑ Вообще-то мы не можем это сказать!
- ❑ Выполняем три поиска и **делаем что-то еще**
- ❑ Что именно еще – нельзя сказать по f. Это зависит от того, какой тип возвращают функции find.



Снова задача про композицию find



Напомним условие

Пусть у нас `find` возвращает пару (*найденный элемент, хвост*)

`find (>3) [1, 4, -7, 8, -2, 2] → (4, [-7, 8, -2, 2])`

Придумать какой-нибудь способ соединять такие `find` примерно как мы соединяли `add`.

`f = find (>3) ??? find (>5)`

▣ Вариант 1 – «супер композиция»

`f = find (>3) >>> find (>5)`

В этой части давайте считать, что неудач нет, мы всегда все находим.

«Суперкомпозиция» и почему ее недостаточно

```
(f >>> g) xs = let  
    (_, xs1) = f xs  
    (x2, xs2) = g xs1  
in (x2, xs2)
```

Какие проблемы?

«Суперкомпозиция» и почему ее недостаточно

```
(f >>> g) xs = let  
    (_, xs1) = f xs  
    (x2, xs2) = g xs1  
in (x2, xs2)
```

Какие проблемы?

- Как найти число, большее 3, а потом – число, большее того, которое мы нашли в первый раз?
- Как найти число, большее 3, потом число, большее 5 и вернуть их сумму?

Решение?

>>>= !!!

```
f = find (>3) >>>= \x -> find (>x)
```

```
(f >>>= g) xs = let  
    (x1, xs1) = f xs  
    (x2, xs2) = g x1 xs1  
in (x2, xs2)
```

Совет: Follow the types!
Надо понять, что там какого типа, и тогда все просто.

Но что же делать с второй задачей:
«Найти число, большее 3, потом число, большее 5 и вернуть их сумму» ?

```
f = find (>3) >>>= \x ->  
    find (>5) >>>= \y ->  
    return1 (x+y)
```

Задачи на дом

- ▣ Написать `return1`
- ▣ * Сделать так, чтобы мы могли использовать не `>>=` и `return1`, а стандартные `>>=` и `return` (и, как следствие, `do` нотацию)

Тогда мы могли бы писать:

```
f = do x <- find (>3)
      y <- find (>5)
      return (x+y)
```

Но придется еще довольно много узнать про Haskell

- Потому что для функций нельзя переопределять `>>=`
- Придется разобраться со словами `type`, `class`, `Control.Monad`

Чего мы добились

```
f = do x <- find (>3)
      y <- find (>5)
      return (x+y)
```

- «а потом» применяется к функциям высшего порядка
- Примерно как если бы в обычном языке мы могли написать
sin ; cos ; ln

Что-то сложновато?

и 40



Что-то сложновато?

```
f xs = (x, xs1) = find (>3) xs  
      (y, xs2) = find (>5) xs1  
      return (x+y, xs1)
```

→

```
f = do x <- find (>3)  
      y <- find (>5)  
      return (x+y)
```

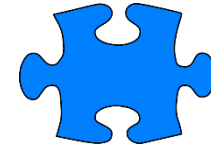


Не слишком ли сложно для того, чтобы избавиться от от одного параметра?

▣ Не знаю, может быть

Что мы не прошли

- Еще одна часть паззла – монада IO (ввод-вывод)
- Связь монад и классов Functor, Applicative и Monoid.
- Откуда берутся законы монад



И все таки, сколько нужно решить задач, чтобы понять монады?

1. Про композицию
`f xs = sum (map sin xs)`
2. Про дописывание в список, например по дереву получить список элементов
3. `find`, который возвращает пару (элемент, хвост)
4. `map` – заменить 2 на 5
5. Вложенный `map` – например, крестик
6. “`supermap`”
7. Упражнение на использование `>>=`
8. `decartes`
9. Упражнение на вложенные `>>=` - с `do` и без него
10. `find`, который сообщает об ошибке
11. Какая-нибудь задача про `Maybe`
12. «Три поиска»
13. «Три поиска» с `Maybe`
14. `>>>`
15. `>>>=`
16. `return1`
17. * `>>=` и `return` вместо `>>>=` и `return1`
18. Что-нибудь про ввод-вывод (монада `IO`)

Немало:)

Некоторые ссылки

- ▣ The Monad Challenges
<http://mightybyte.github.io/monad-challenges/>

Кто это все придумал?



Eugenio Moggi



Philip Wadler