

Сколько задач надо решить, чтобы понять монады?



Михаил Симуни
simuni@mail.ru

Замечания

- ▣ Почти только задачи и почти все *очень* простые
- ▣ Вообще, видимо, за час про монады рассказать нельзя :(
(Примерно как объяснить за час интегралы)
 - Во первых, времени маловато. Во вторых, все равно, надо порешать задачи, чтобы что-то понять
 - Но мы попробуем. В любом случае точно можно получить общее впечатление.
- ▣ Если непонятно, спрашивайте!

В таких рамках
дальше будут
приведены
доп.задачи, если
вдруг кто-то захочет
что-то решить сам. В
основном задачи
простые или очень
простые

Что желательно знать про Haskell



Композиция, карринг, секции, map

Композиция

- $\sin \circ \cos$ - сокращение для $x \rightarrow \sin(\cos x)$

Карринг – задаем *часть* параметров

- $f \ x \ y$
 $f \ 1$ - сокращение для $x \rightarrow f \ 1 \ x$

Кстати, строго говоря, это не сокращение, как раз $f \ x \ y$ - сокращение

Секция

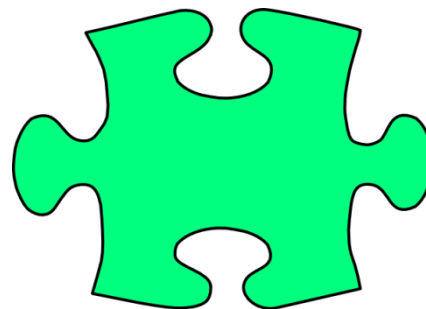
- $(2+)$ – сокращение для $x \rightarrow 2 + x$

map

- $\text{map } \sin [1, 2, 3] \rightarrow [\sin 1, \sin 2, \sin 3]$

Если вы не очень понимаете, что значат слова на этом и паре следующих слайдов, имеет смысл, может быть, начать с сайта msimuni.wikidot.com/fp4 (курс матмека прошлого года)

Задачи про КОМПОЗИЦИЮ



Простые примеры

- ▣ Найти сумму синусов для списка

`f xs = sum (map sin xs)`

Как это можно записать короче?

- ▣ `f n = 10 * n + 7`

Как это можно записать короче?

Простые примеры - ответы

▣ `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)`

При такой форме записи иногда получается коротко и понятно, а довольно часто – это коротко и непонятно :(Для меня лично грань где-то между этими примерами, уже второй – не очень понятен.

Написать функцию, параметр которой – список списков целых чисел. Функция должна вернуть их сумму. И при этом функция должна быть написана в таком стиле, с помощью композиции

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*. А обобщение в теории категорий называется *codensity monad*

1. Написать функцию `addUp n xs`, которая приписывает к списку числа от 1 до `n`.
Например, `addUp 3 [10,11]`
– это `[1,2,3,10,11]`
2. Попробуйте написать эту функцию с помощью композиции

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 = add 10 1 xs
          xs2 = add 10 2 xs
          xs3 = add 10 3 xs
          in xs3
```

Как записать короче?

Ответ

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

1. `rev xs` – переставляет числа в обратном порядке. Надо написать без использования `++` (тут надо знать, что такое `++`)
2. `flatten t` - для дерева вернуть список его элементов, без использования `++`. (Тут надо знать, как описать деревья и с ними работать)
3. * Реализовать `rev` с помощью `foldr` (тут надо знать, что такое `foldr`)



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

Какой бы вы предложили интерфейс?

1. Выполнить первую просьбу начальника – написать функцию, которая ищет в списке первый элемент, удовлетворяющий условию. Тут считаем, что элемент точно всегда найдется.
2. Попробуйте написать эту функцию с помощью стандартных функций `filter` и `head`
3. И с помощью композиции

Поиск в списке – интерфейс «с хвостом»

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.

Это теперь совсем
простая задача:
напишите find с таким
интерфейсом

Вопрос

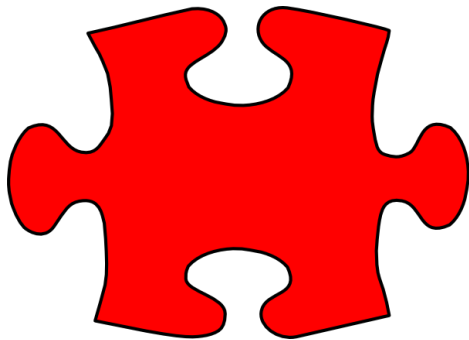
Придумать какой-нибудь способ соединять такие find, примерно как мы соединяли add.

$f = \text{find}(>3) \text{ ??? } \text{find}(>5)$

Ответ будет в конце, и желающие могут попробовать сами его придумать.



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

1. Напишите с помощью `map` функцию, которая заменяет в строке все `'?'` на `'!'` и наоборот. (Строка в Haskell – это список символов).
2. Напишите функцию `f n`, которая возвращает список длины `n`, в котором чередуются 4 и 5.
3. Если получится, двумя способами, один с помощью `map` и еще какой-нибудь.

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]
```

Опишите функцию,
которая создает
«крестик» из 0 и 1
`cross 5 =`

```
  [[1,0,0,0,1],  
   [0,1,0,1,0],  
   [0,0,1,0,0],  
   [0,1,0,1,0],  
   [1,0,0,0,1]]
```


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.

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

Н. L. Mencken (переделанная цитата)



Вдали уже
показались
монады...

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...

Это очень просто, но попробуйте, может, сами написать такой supermap.

>>=

>>=

- 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...»

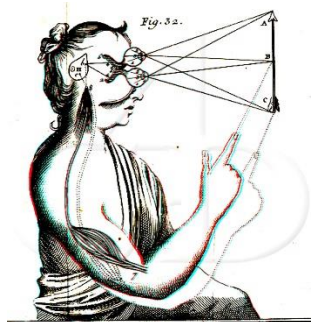
Что то вроде тела цикла

Простое упражнение: С помощью >>= описать функцию, которая в данном списке после каждого четного числа дописывает число, большее на 1. Например, [1,4,3,10] → [1,4,5,3,10,11]

Вложенные $>>=$

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

Попробуйте написать его, не заглядывая в следующий слайд?



См. <https://www.inf.ed.ac.uk/teaching/courses/inf1/fp/lectures/2015/lect15.pdf>
про то, как эта картинка связана с Декартом и монадами

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)
```

▣ `do` нотация

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

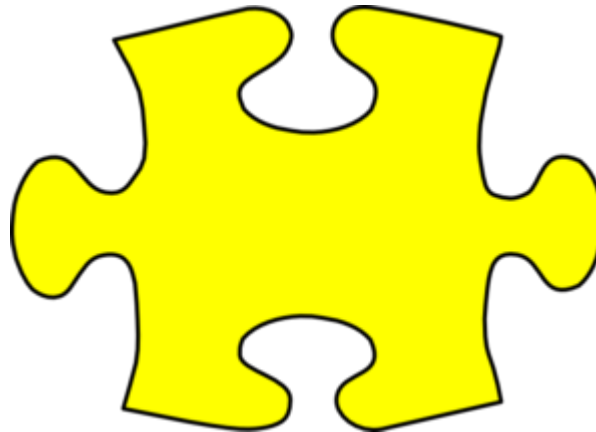
Зачем??

Абстракция!

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

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

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.



Варианты решения:

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

Еще интересный вариант, кстати - failure continuation

1. Простое упражнение: опишите такую функцию для варианта с [x] и []
2. Еще один вариант – это возвращать (True, x) или (False, что-то). Но что написать на месте «что-то»? Если написать например, 0, то получится не полиморфно, find будет работать только для чисел. А как написать полиморфно?

Задача о трех поисках

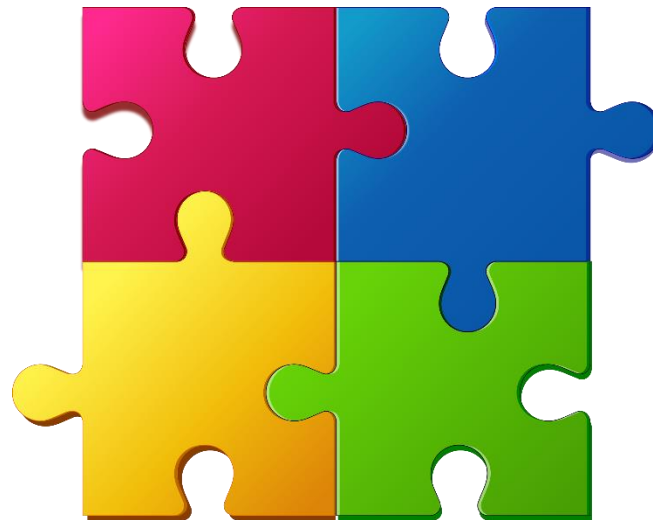
Пусть у нас уже написан `find`, который возвращает `[x]` или `[]`. Я хочу в списке найти:

- ▣ 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**

1. Просто для упражнения, напишите то же с помощью >>=
2. Напишите это же с помощью list comprehension (возможно, придется узнать, что это такое). Так получится совсем коротко.

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: 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)
```

Для студентов это просто балл в подарок, надо просто переписать решение предыдущей задачи. Ну и хорошо, пусть запомнят этот счастливый день, когда они прошли монады :)

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):

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

- На самом деле, не так сложно объяснить, откуда они берутся, но в этот раз времени не хватило.
- Но на практике они не очень то и нужны, писать и использовать монады вполне можно не вникая в законы.

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

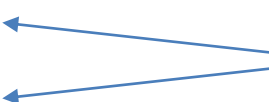
Сейчас будет несколько главных слайдов, где написано главное, что, мне кажется, надо понять. И это первый. И тут главное: «Монады – это про слово «потом»»

«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



Тут было верное замечание, что на самом деле монады не только соединяют действия, они еще передают результат от первого действия ко второму.

На самом деле правильнее сказать, что монада – это программируемый оператор `let`. Но с точкой с запятой как то получается не совсем точно, зато более доходчиво :)

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.

И это второе, мне кажется, главное. Монады позволяют отдельно задать последовательность действий и отдельно – что при выполнении этой последовательности надо делать что-то дополнительное. И это очень интересно. (Хотя и не то, видимо, что надо каждый день простому программисту...)

Абстракция в иллюстрациях



Мультфильм «Головоломка» (Inside Out).
Внутри человека живут разные эмоции и воспоминания. Слоник – это воспоминание, и он случайно попал в область, где вещи абстрагируются.



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.

Я бы очень советовал попробовать написать `>>>` не заглядывая не следующий слайд! Это важно!

«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?

>>>=

Надо написать оператор, который позволит писать как то так:

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

То есть, это похоже на >>>, но только:

- второй find вызывается не прямо, а внутри лямбда выражения
- И через это лямбда выражение мы сможем добиться того, чтобы второй find мог использовать результаты первого.

И тут попробуйте, пожалуйста, написать >>>= сами. Это важно! (Но, если вы были на лекции, то, наверное, запомнили решение, там надо буквально исправить несколько букв из решения предыдущей задачи)

И это третий главный слайд, наверное самый главный. Если вы поймете, почему >>>= решает нашу проблему, значит вы совсем поняли, что такое монада. Подумайте! Совет – разберитесь с типами, что тут какого типа?

> > > =

```
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.

И осталось только разобраться с return

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)
```

Попробуйте написать такой return1. Это попроще, чем >>>=, у вас должно получиться!)

Совет: не уверен, но может быть будет понятнее, если расставить скобки:

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

То есть это у нас лямбда выражение, в которое вложено другое лямбда выражение и т.д. (Похоже на вложенные map в задаче про таблицу умножения).

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 (точнее добьемся, когда напишем настоящую монаду)

```
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

Мне кажется, то раз поэтому, это все довольно трудно объяснить. Появляются одновременно две непривычные вещи, а это всегда очень трудно понять.

Вопрос от воображаемого слушателя: Что-то сложновато?

После часа умственных усилий мы получили сокращенную программу.

```
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 одного параметра?

▣ Не знаю, может быть



Что мы не прошли

- Еще одна часть паззла – монада IO (ввод-вывод)
Но это вы сами можете почитать, теперь это уже должно быть не очень сложно. Если в двух словах, это похоже на последний пример с `find`. Только там мы прятали списки-параметры, а тут – файлы-параметры.
- 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:)

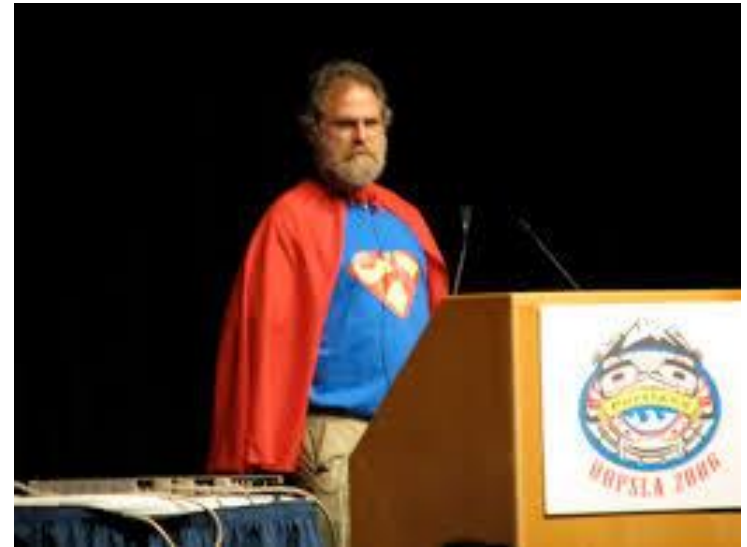
Некоторые ссылки

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

Кто это все придумал?



Eugenio Moggi



Philip Wadler

Пишите вопросы, пожалуйста

- Если есть вопросы по этим слайдам, или вам кажется, что в них ошибка, напишите, пожалуйста! Адрес simuni@mail.ru
- Если плохо понятно, но хочется разобраться – напишите, я бы рад что-то объяснить. Только напишите, пожалуйста, что именно непонятно, или, по крайней мере, с какого слайда.