Основы функционального программирования на языке Haskell.

Автор: доцент каф. ВМиК УГАТУ, к.т.н., Макеев Григорий Анатольевич, grigorym на гмейл

От автора

> …My Dear Frodo, you asked me once if I had told you everything there was to know about my adventures. While I can honestly say I have told you the truth, I may not have told you all of it…

Этот текст вырос из курса лекций по функциональному программированию, читавшегося студентам специальности 010503 "Математическое обеспечение и администрирование информационных систем" очной формы обучения на факультете Информатики и Робототехники Уфимского Государственного Авиационного Технического Университета до 2012 года.

Обучение функциональному программированию за 8 лекций - серьезный вызов. За несколько лет преподавания у автора выработалось определенное понимание того, каким образом необходимо объяснять те или иные понятия и концепции функционального программирования. Не претендуя на единственную правильность используемого подхода, автор все-таки надеется, что предлагаемый текст пополнит все еще довольно скудную коллекцию русскоязычной литературы по языку Haskell и функциональному программированию в целом.

Язык Haskell не относится ни к старым и заслуженным, ни к популярным и известным каждому школьнику языкам программирования. Аудитория его пользователей и почитателей (вторая, кстати, как мне кажется, намного шире – по крайней мере, автор принадлежит только к ней) довольно ограничена. Поэтому первое достаточно полное русское издание, посвященное программированию на языке Haskell, книга "Функциональное программирование на языке Haskell" [1] появилась в печати только в 2007 году. Заслуживающий безусловного уважения монументальный труд Романа Душкина, разумеется, занял свое место на полках у ценителей функционального программирования. Но лица моих студентов очень быстро кисли при первом взгляде на оглавление и толщину книги. Зато один из двух или трех моих экземпляров книги надолго прописался у коллеги, преподающего лямбда-исчисление и теорию комбинаторов.

Наконец, ситуация с русскоязычной литературой получила качественное изменение. Буквально только что, в 2012 году, в том же издательстве ДМК Пресс вышел перевод книги Мирана Липовачи "Изучай Haskell во имя добра!" [2] (она же – "книжка с синим слоном"), книги яркой и веселой, переведенной целым коллективом авторов и под редакцией того же Романа Душкина. На обе эти книги я и старался равняться, периодически беззастенчиво используя удачные аналогии – периодически находя свои, как мне кажется, более пробивные объяснения. Получилось или нет – решать моим студентам после получения зачета, и всем остальным, кто будет читать этот текст не по велению неумолимого учебного процесса, а по движению души.

Структура

После небольшой демонстрации, необходимой для привлечения внимания и произведения "вау-эффекта", последует большая теоретическая часть.

В ней я сначала рассматриваю основные понятия – через призму новой предметной области, показываю основные типы данных языка Haskell и простейшие стандартные функции для манипуляции ими.
Затем объясняются основные способы написания функций – тела программ на функциональных языках, объясняется такая неимоверно удобная технология, как сопоставление с образцом. Затем описываются некоторые особенности языка, и дальше мы переходим к самому главному – объяснению, что же, собственно, представляет собой рекурсивно-функциональный способ

решения задач.
Самое главное, чему я стараюсь научить студентов – это идея функций высших порядков и те потрясающие возможности, которые дает умение их использовать. Функциям высших порядков посвящена, наверное, самая большая и подробная глава.
Следующая глава посвящена созданию собственных типов данных, в том числе рекурсивных и потенциально бесконечных – что является неожиданным для студентов, которые привыкли писать на C++ и бесконечность видеть только в неправильной работе с указателями, приводящей к структурам данных, указывающих друг на друга.
Предпоследняя глава теоретического раздела посвящена организации ввода-вывода в Haskell, который хоть и реализуется с помощью такого ускользающего понятия, как монада, но на деле не представляет никаких особых трудностей – если не заглядывать слишком глубоко в эту кроличью нору.
И в последней главе рассматривается пример решения на Haskell всем известной задачки про волка, козу и капусту. Предлагается два решения, отличающиеся моделью данных и кое-какими учитываемыми или не учитываемыми деталями.

Вторая часть представляет собой задачник, разбитый на 5 глав, по количеству лабораторных работ, которые студентам необходимо выполнить при изучении курса функционального программирования. Задачи приведены разные, от простых - к задачам средней сложности. Прорешав все задачи, студент готов сдавать зачет, на котором ему будет предложено без компьютера написать на бумаге решение определенной функциональной задачи.

В приложении 1 описана установка и простейшие приемы использования интерпретатора WinHugs, с помощью которого можно проверять все примеры, приведенные в этом тексте и решать задачи, представленные в задачнике. В приложении 2 приводится список рекомендуемой литературы.

Очень часто в процессе работы над текстом я упирался в пределы своих знаний, и старался не писать о том, что сам плохо понимаю, маскируя незнание туманными и многозначительными намеками. Какие-то детали я намеренно опускал, ведь у студентов было всего 8 лекций. Поэтому простите мне недосказанность (см. эпиграф), ведь область функционального программирования и теории вычислений в целом - просто неисчерпаема. А если встретите фактическую ошибку - напишите мне, я исправлю.

Пояснения и обозначения

`Вот таким шрифтом обозначаются куски кода, имена функций и сигнатуры типа.`

Вот таким образом оформлены замечания, дополнения и заметки на полях.

Демонстрация кунг-фу

Классика жанра. В качестве мотивации студентов к изучению Haskell (кроме, конечно, демонстрации пряника зачетных единиц и кнута в виде списка отчисленных за прошлый семестр) обычно приводят небольшие кусочки кода, которые должны поразить своей красотой, краткостью и мощностью. С краткостью и мощностью обычно проблем нет. Вот быстрая сортировка любого списка:

```
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y <= x]
               ++ [x] ++
```

```
            qsort [y | y <- xs, y > x]
```

Вот бесконечный список чисел Фибоначчи:

```
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

или так:

```
fibbs = 1 : 1 : next fibbs where
    next (x:y:xs) = x + y : next (y:xs)
```

или так:

```
fibbbs = next 0 1 where
    next x y = y : next y (x+y)
```

Вот простые числа:

```
primes = map head $ iterate sieve [2..] where
    sieve (x:xs) = [y | y <-xs, y `mod` x /= 0]
```

или так:

```
primes = 2 :
    filter (\x -> all (\p -> x `mod` p /= 0) $
                  takeWhile (\p -> p^2 <= x) primes)
           [3..]
```

А вот создание идеально сбалансированного бинарного поискового дерева:

```
data Ord a => SearchTree a = Empty | Branches a (SearchTree a) (SearchTree a)
                           deriving Show

putVal x Empty = Branches x Empty Empty
putVal x (Branches v l r)
      | x > v     =     Branches v l (putVal x r)
      | otherwise =     Branches v (putVal x l) r

list2tree xs = foldl (flip putVal) Empty xs

height Empty = 0
height (Branches v l r) = max (height l) (height r) + 1

stepr (Branches v l@(Branches lv ll lr) r)
      | height lr > height ll  = stepr (Branches v (stepl l) r)
      | otherwise              = (Branches lv ll (Branches v lr r))
stepl (Branches v l r@(Branches rv rl rr))
      | height rl > height rr  = stepl (Branches v l (stepr r))
      | otherwise              = (Branches rv (Branches v l rl) rr)

balance Empty = Empty
balance t@(Branches v l r) = balance' (Branches v (balance l) (balance r)) where
      balance' t@(Branches v l r)
            | height l - height r > 1      = balance $ stepr (Branches v l r)
            | height r - height l > 1      = balance $ stepl (Branches v l r)
```

```
        | otherwise = t
```

Так как студенты обычно решают эти задачи на первом-втором курсе в рамках изучения программирования и стандартных алгоритмов, они помнят, сколько кода сортировка или простые числа требуют на C++, и сколько мучений с поворотами и указателями при работе с идеально сбалансированными деревьями. Может быть, они даже помнят, как приходилось думать, сколько памяти выделять под массив простых чисел, и поэтому впечатляются бесконечным списком чисел Фибоначчи или списком простых чисел.

Однако настоящую красоту функционального программирования, красоту идей и смыслов, невозможно продемонстрировать кодом на неизвестном языке. И я больше всего радуюсь именно тогда, когда удается найти такое объяснение, благодаря которому еще у кого-нибудь загораются восхищением глаза.

Теория

Основные понятия и типы данных

Начнем с того, что определимся с основными терминами, которые будем использовать в дальнейшем. Дело в том, что многие понятия, с которыми вы сталкивались раньше, в курсе функционального программирования не имеют применения, а некоторые, наоборот, получают расширительное толкование.

Первое понятие, которое чуть позже заиграет новыми неожиданными красками - это "значение". **Значение** - кусочек простейших данных, который можно понимать как ответ на какой-то вопрос, как результат вычисления, и который нельзя дальше "упростить" или "вычислить". Например, значениями являются:

```
5
True
'a'
(1,True)
[1,2,3,4,5]
```

Очевидно, что сложно назвать значением следующие конструкции:

```
5+6
x-1
Integer
```

Если рассмотреть различные пары значений, то можно обнаружить, что некоторые из них нам кажутся интуитивно более "похожими" друг на друга, чем другие. Например, значения `1` и `2` гораздо более похожи друг на друга, чем значения `1` и `"one"`. Любой программист сразу скажет, что значения `1` и `2` принадлежат одному и тому же **типу** `Integer`. **Типом** можно назвать множество значений (как очень маленькое, так и потенциально бесконечное), которые в некотором смысле похожи все друг на друга, как братья.
А если более конкретно? Чем значение `1` похоже на значение `2` и отличается от значения `"one"`? Да тем, что к значению `1` можно прибавить `10`, а к значению `"one"` - нельзя. Чуть позже мы увидим, как это проявляется формально, а пока просто констатируем: **тип** - это множество значений одной природы, похожих тем, какие операции к ним всем применять можно, а какие нельзя.
Для любого значения мы будем указывать, какому типу оно принадлежит, и использовать для этого двойное двоеточие. Вот примеры значений для простых типов данных языка Haskell:

```
5 :: Integer
5 :: Int
5 :: Float
True :: Bool
'a' :: Char
```

Типы `Integer` и `Int` отличаются длиной: значения типа `Int` имеют ограниченную длину, а вот тип `Integer` может содержать целое число с любым количеством цифр. Если вы так и не поняли, к какому типу принадлежит число `5`, то я вам скажу – ко всем этим трем сразу. А еще оно является, к тому же, рациональным и комплексным.

Следующее очень важное понятие - это **переменная**. В обычных языках программирования под переменной понимают определенную область памяти, имеющую определенное имя и текущее

значение, которое может меняться в процессе выполнения программы. В математике вместо слова "переменная" иногда используют слово "неизвестная": например, в уравнении `sin(x) = x`, говоря о неизвестном "`x`" имеют в виду, что вместо `x` можно подставлять разные числа, и какие-то из них удовлетворяют уравнению, а какие-то нет.
В языке Haskell нет переменных, и если `x` получает, или имеет, значение `1`, то это значение не изменяется на всем протяжении жизни этого `x`. Правильнее в этом случае говорить просто об "имени" `x`, которое имеет определенное неизменное значение. Раз значение имени `x` не меняется на протяжении жизни этого `x`, а любое значение имеет тип, то и любое имя `x` имеет неизменный тип, например:

```
x :: Int
```

Давайте сразу договоримся: когда мы будем произносить слово "переменная" в применении к языку Haskell, мы будем иметь в виду именно эти неизменные `x`, `y`, `z` - за которыми скрывается какое-то значение, изменить которое мы не в состоянии.

> Заметки на полях: Haskell довольно строго регламентирует регистр имен: все они - и "переменные", и функции (а по сути это одно и то же) должны начинаться с строчных букв, и могут содержать буквы, цифры и одинарную кавычку. Прописные зарегистрированы за типами данных и за специфическими значениями вроде `True`, `False` - но об этом потом.

Когда имя `x` получает свое значение? Некоторые имена получают свои значения, можно сказать, при ~~сотворении мира~~ запуске программы. Если у вас в программе написано: `pi=3.1415`, то имя `pi` получает свое значение однажды и навсегда. А если у вас есть функция `tg x = sin x / cos x`, то имя `x` рождается и получает значение каждый раз при вызове этой функции (уничтожаясь при завершении функции), и каждый раз оно может быть разным - но изменить его внутри функции мы не можем.

Выражение, неформально говоря - это какая-то конструкция, которую в отличие от значения можно "упростить", "дальше вычислить", или вот еще термин - "редуцировать". Например, выражение `4+5` можно редуцировать до значения `9`, а выражение `(4+5)*2` можно упростить до выражения `9*2`, которое, в свою очередь, упростить до значения `18`. Этот факт мы будем отображать следующим образом:

```
4+5 → 9
(4+5)*2 → 9*2 → 18
(x-1)^2 + (x+1)^2 → x^2+2*x+1 + x^2-2*x+1 → 2*x^2+2
```

Как видим, выражение может включать в себя как константы, так и ~~переменные~~ имена. Очевидно, что выражение, так же как и значение, имеет определенный неизменный тип. Каким конкретно образом вычислять выражение, в каком порядке вычислять в сложном выражении его части - это совершенно отдельная проблема, в которую мы лучше пока не будем залезать.

Из простых типов данных конструируются сложные, к которым относят **список**, **кортеж** и **функцию**.

Списки

Список – одна из главных структур данных в функциональных языках. Список – это потенциально бесконечный упорядоченный набор однотипных элементов. Взяли набор однотипных элементов, записали через запятую, взяли в квадратные скобки - получили список. Вот пример,

соответственно, списка целых чисел, списка символов и списка списков целых чисел.

```
[1,3,4,2] :: [Integer]
['h','e','l','l','o'] :: [Char]
[[1,2],[],[3,1]] :: [[Integer]]
```

Обратите внимание, из типа выражения сразу видно, что это список. Если тип какого-то выражения есть `[SomeType]`, то мы имеем дело со списком значений типа `SomeType`, даже если `SomeType` в свою очередь есть что-то сложное – опять список, кортеж, список кортежей, кортеж списков или вообще функции:

```
[sin, cos] :: [Float -> Float]
[sin 5, cos 5] :: [Float]
```

Обратите внимание, в первом списке мы имеем дело со списком функций, в котором лежат синус и косинус, сами по себе, как функции. А во втором случае у нас банальный список значений типа `Float`.

Но вот сложить в один список значения различных типов – не получится. Зато их там может быть как угодно много.
Строки являются обычными списками символов. Причем не обязательно записывать каждый символ в одинарных кавычках - можно записать всю строку в двойных. С другими списками, правда, такой фокус не выйдет – это привилегия строк.

```
"hello" :: String
"hello" == ['h','e','l','l','o']
```

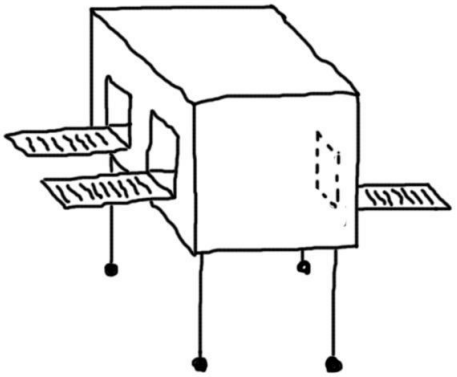
Кортежи

Кортежи, в отличие от списков, могут содержать разнотипные элементы, но их обязательно фиксированное число. Кортеж является, по сути, аналогом записи в императивных языках, только без названий у полей:

```
(1,'a') :: (Integer, Char)
(True,1) :: (Bool, Integer)
("Vladimir",12,[5,5,5]) :: (String, Integer, [Integer])
```

Функции, операторы

Что такое функция? В привычном для нас всех смысле, функция - это механизм, который умеет преобразовывать одни данные в другие. Это может быть удивительным для тех, кто привык к императивным языкам, но функции тоже относят к данным, потому что ими можно манипулировать, как любыми другими данными (если встретите трудноперводимое на русский язык выражение "a function is a first-class citizen" - оно именно об этом).
Удобно представлять себе функции как большие передвижные ящики-преобразователи на колесиках, у которых есть одно или несколько входных отверстий и одно выходное.



Большую часть времени ящик простаивает, но если ему во входные отверстия вложить требуемые материалы, то ящик заработает и выкинет в выходное отверстие что-то новое, что является результатом преобразования исходных материалов.
В такой аналогии становится понятно, почему с функциями можно обращаться так же как с любыми другим данными: складывать в списки и кортежи, передавать в другие функции и получать как результат из третьих функций.
Функция - это специфическое значение. Значение - потому что ее нельзя дальше упростить, вычислить в нечто более простое (напоминаю, функция - это ящик с отверстиями). А специфическое - потому что в отличие от, скажем, числа, которое нельзя ни для чего особенного применить (только принять к сведению как ответ), функцию можно **применить** к значениям для получения других значений. В функцию можно **передать** значения, чтобы получить другие значения, вот так:

```
mod 5 3 → 2
```

Обратите внимание, вызов функции не требует никаких скобок, даже если этих аргументов несколько. Более того, если одиночный аргумент можно и взять в скобки `sin (5)` , то в случае двух аргументов что `mod (5 3)`, что `mod (5,3)` будет ошибкой.

Но вернемся к самой функции. Раз функция - это значение, а любое значение имеет тип, то мы должны говорить о типе функции. Раз функция принимает значения, имеющие тип, то мы должны говорить о типе параметров функции. Собственно, вместо всех этих слов можно было просто сказать, что тип функции - это совокупность типов всех ее параметров и типа результата:

```
sin :: Float -> Float
ord :: Char -> Int
```

Что нам говорит эта запись? Нечто, скрывающееся под именем `sin` - это функция, принимающая значение типа `Float`, и возвращающая значение типа `Float`. Заметим, что тип этой функции не говорит нам ничего о том, **что** эта функция делает. Несмотря на это, чаще всего по имени функции и по типу ее параметров можно догадаться о ее назначении. Так, `ord` - это функция, принимающая значение типа `Char` и возвращающая значение типа `Int`, а название говорит нам о том, что скорее всего эта функция возвращает порядковый номер символа (конечно же, в таблице ASCII).

```
(+) :: Float -> Float -> Float
```

Несколько стрелочек говорят нам о том, что перед нами функция нескольких параметров. В данном случае `(+)` - это функция, берущая два значения типа `Float`, и возвращающая значение типа `Float`.

Последняя стрелка в данном случае указывает на тип результата, а все предыдущие стрелки разделяют входные параметры.

> Правда, "логично"? Стрелка одна и та же, но в одном случае она означает "входной параметр", а в другом случае "результат"? На самом деле в языке Haskell все очень строго и логично, без всяких кавычек, и эту кажущуюся несуразность мы очень скоро ликвидируем.

Полиморфные типы данных

Давайте вернемся к функции сложения и применим к ней метод внимательного взгляда.

```
(+) :: Float -> Float -> Float
```

А если надо сложить два целых числа? Какую-то другую функцию использовать? А как насчет функции `length`, находящей длину списка символов?

```
length :: [Char] -> Int
```

А если надо найти длину списка целых чисел? Или длину списка значений типа `Bool`? Или длину списка списков `Integer`? Каждый раз разные функции использовать?

Давайте заглянем вперед и узнаем, как в действительности написана функция `length`:

```
length [] = 0
length xs = 1 + length (tail xs)
```

Код этой функции описывает сам себя: длина пустого списка считается нулевой, а длина любого другого списка считается как увеличенная на единицу длина исходного списка без первого элемента (функция `tail` возвращает все элементы списка, кроме первого).

Попробуйте найти, где в этой функции есть указание на то, со списком чего она должна работать? Нигде! И действительно, эта функция работает с любыми списками - и со списками чисел, и со списками символов, и со списками строк, и со списками списков кортежей, - и даже со списками функций!

Но как же тогда описать ее тип? Для этого используется понятие "переменная типа":

```
length :: [a] -> Int
```

Здесь `a` - это как раз переменная типа, то есть нечто, под чем может скрываться абсолютно любой тип. И фразу эту мы тогда прочитаем так: `length` - это функция, берущая список значений произвольного типа `a` и возвращающая значение типа `Int`.

Вернемся в третий раз к функции сложения и запишем ее аналогично функции `length`:

```
(+) :: a -> a -> a
```

Но разве можно применять числовое сложение, например, к кортежам? Или, например, к фунциям? Разве имеет, например, смысл выражение `sin + cos`? Обратите внимание, что мы пытаемся складывать две функции, а не два значения `Float`, как было бы в случае выражения `sin`

`5 + cos 5`, которое вполне осмысленно.
Получается, с одной стороны, мы не хотим ограничивать операцию сложения каким-то одним типом, а с другой стороны, вседозволенность этой функции точно не присуща. С какими типами должна оперировать функция сложения? Ответ на самом деле прост - с числовыми! Самый правильный тип этой функции выглядит так:

```
(+) :: Num a => a -> a -> a
```

И читается эта запись следующим образом: функция сложения берет значение какого-то типа `a`, затем берет еще одно значение того же типа `a` и возвращает значение того же самого типа `a`, где тип `a` принадлежит **классу** численных типов `Num`.
Вот оно что: оказывается, можно **типы** данных сгруппировать в **классы** типов по тому, какие операции к ним применимы!

А как насчет операции сравнения? Должна она позволять сравнивать только числа? А может быть, только символы?

```
(>) :: Integer -> Integer -> Bool    ?
(>) :: Float -> Float -> Bool        ?
(>) :: Char -> Char -> Bool          ?
```

С какими типами она должна работать? Может быть, с любыми?

```
(>) :: a -> a -> Bool                ?
```

Я думаю, вы уже догадались: она должна работать только с такими типами данных, которые допускают сравнение значений на больше-меньше, и этот факт отражается в использовании класса типов `Ord`:

```
(>) :: Ord a => a -> a -> Bool
```

> Не спутайте этот класс `Ord` с функцией `ord :: Char -> Int`, которая нам уже попадалась раньше. Они никак не связаны, это просто совпадение. Класс `Ord` – это класс таких типов, которые допускают сравнение на больше-меньше.

Аналогичным образом, операция проверки равенства двух значений должна оперировать только такими типами, которые допускают проверку равенства:

```
(==) :: Eq a => a -> a -> Bool
```

Чтение сигнатур типов

Теперь мы знаем достаточно, чтобы уметь читать любую сигнатуру типа. Или, другими словами, разбираться в типе любых значений, выражений и функций, которые нам могут встретиться. Давайте потренируемся в построении сложносочиненных и сложноподчиненных выражений русского языка. Как только поймете, что от скобок и стрелок кружится голова - переходите к следующему разделу: в конце концов, совсем уж сложные сигнатуры типов встречаются редко.

```
foo :: (a -> b) -> [a] -> [b]
```

Значение `foo` - это функция, берущая функцию, берущую значения типа `a` и возвращающую значение типа `b`, берущая также список значений типа `a` и возвращающая список значений типа `b`.

Громоздко звучит? Давайте подсократим и используем синонимы: `foo` - это функция, берущая функцию, отображающую тип `a` в тип `b`, берущая список типа `a` и возвращающая список типа `b`.

```
boo :: ((a,b) -> b) -> [b -> [a] -> b]
```

Значение `boo` - это функция, которая:

- принимает функцию `((a,b) -> b)`, берущую кортеж (пару) из значения типа `a` и значения типа `b` и возвращающую значение типа `b`;
- возвращает аж список функций `[b -> [a] -> b]`, берущих значение типа `b` и список типа `a` и возвращающих значения типа `b`.

Простейшие функции и операторы

Ладно, с простыми и составными типами данных мы разобрались. Самое время разобраться, какие же элементарные функции нам предоставляет язык Haskell для работы с ними.

Арифметические функции

Ну, тут все просто. Арифметика обязана быть в любом языке, и Haskell тут не исключение. Может быть, вас, разве что, удивят типы некоторых операторов, - ну так самое время проверить, как вы научились читать типы данных, на примере следующих операторов и функций:

```
(+), (-), (*), (/), div, mod, (^), sum, product, max, min, maximum, minimum, even, odd, gcd, lcm
```

Кстати, **операторами** мы будем называть бинарные функции, название которых состоит из сплошных знаков пунктуации. В остальном, оператор - точно такая же функция, просто используемая чаще в инфиксной нотации, чем в префиксной. Обычные же бинарные функции, состоящие из букв и цифр, чаще всего используются в префиксной нотации, хотя можно и в инфиксной.

Короче говоря:

- можно написать `5+3`, а можно `(+) 5 3`, и это будет одним и тем же;
- можно написать `mod 5 3`, а можно `5 `mod` 3`, и это тоже будет одним и тем же (обратите внимание, это обратные апострофы, а не обычные кавычки).

Логические функции

С логическими функциями, то есть с функциями, возвращающими значения типа `Bool`, тоже все просто. Сравнения на больше-меньше, на равенство и неравенство, комбинация нескольких условий с помощью операций "и", "или" и "не". Немного на особицу стоят функции (именно функции, а не операторы) `and` и `or`, принимающие сразу список значений `Bool`:

```
(>), (<), (==), (/=), (>=), (<=), (&&), (||), not, and, or
```

Списочные функции

Простейшие списочные функции позволяют выполнять только несколько самых-самых простейших операций по синтезу и анализу списков. Начнем с анализа:

```
head :: [a] -> a
```

Функция `head` берет список и возвращает его первый элемент. Да, вот так просто. Например:

```
head [3,2,1] → 3
head [[3,2],[1,2]] → [3,2]
head "hello" → 'h'
```

Обратите внимание на тип функции `head` - да, она не накладывает вообще никаких ограничений на то, что может храниться в списке. Там могут лежать, в том числе, и функции:

```
head [sin,cos] → sin
```

Второй функцией для разбора списков на составные части является функция `tail`:

```
tail :: [a] -> [a]
```

Эта функция берет список и возвращает "хвост" списка, под которым понимаются все элементы, кроме первого ("головы"). Да, вы правы, хвост начинается сразу от головы. Это действительно удобно. Функции `tail` тоже все равно, что лежит в списке, она не трогает сами элементы:

```
tail [3,2,1] → [2,1]
tail [[3,2],[1,2]] → [[1,2]]
tail "hello" → "ello"
```

Может быть, вы ждете функцию, которая возвращает последний элемент списка? А нет такой! Ну ладно, ладно - есть такая функция. Но если в нее заглянуть, то ничего кроме `head` и `tail` мы не увидим. Потому что списки в Haskell действительно устроены так, что, имея список, можно обратиться только или к первому элементу (к голове), или к той части списка, что идет сразу за ним - к хвосту то бишь. А как же добраться до последней буквы в слове `"hello"`? Ответ прост - несколько раз применить `head` и `tail`:

```
head (tail (tail (tail (tail "hello")))) → 'o'
```

Не ошибитесь в количестве вызовов функций `tail` и `head` - если вызвать их от пустого списка, вы получите ошибку. Проверить, является ли список пустым, можно с помощью третьей списочной функции:

```
null :: [a] -> Bool
```

Функция `null` берет список и возвращает `True`, если этот список пуст, и `False` в любом другом случае.

Хорошо, разбирать список на составные части мы научились. Пора научиться и создавать списки заново:

```
(:) :: a -> [a] -> [a]
```

Встречайте операцию создания списков! Она берет элемент и уже готовый список, и возвращает новый список, добавив элемент в начало (обратите внимание - именно в начало):

```
6:[4,5] → [6,4,5]
```

Функция `(:)`, как выясняется, выполняет операцию обратную тем действиям, что делают со списком функции `head` и `tail`. И действительно, для любого непустого списка `xs` следующее выражение всегда имеет значение `True`:

```
head xs : tail xs == xs
```

> Я слышу голос какого-то зануды: "А если `xs` - это список функций, в котором лежат синусы и косинусы - разве получится сравнить этот список с самим собой - функции `Float -> Float` то ведь сравнивать нельзя, в отличие от самих значений `Float`!?". Не получится, это правда. Это больше, чем равенство – это тождество.

Подождите, мы создавали новый список из уже существующего списка `[4,5]`. А как был создан сам список `[4,5]`? Да в общем-то последовательным применением той же самой операции: `4:(5:[])`. Никаких других возможностей создать список - нет.

Возможно, вы спросите, как добавить элемент в конец? А никак - только писать собственную функцию. А как слить два списка в один? Никак - только писать свою собственную функцию.
Ну ладно, ладно - можно воспользоваться стандартными. Но при этом важно понять, что стандартные функции будут как-то хитро использовать ту же самую операцию `(:)`, потому что других способов создания списка - нет:

```
addToEnd 6 [4,5] → 4:(5:(6:[])) → [4,5,6]
```

Нет обходного пути, позволяющего легко добавить элемент в конец списка, потому что можно обратиться только к голове списка и его хвосту, и чтобы "добраться" до последнего элемента, нужно "прошагать" функцией `tail` по всем его элементам.

> Да, вы правы, добавление элемента к голове списка - операция с константной сложностью, тогда как операция добавление элемента к концу списка обязательно потребует линейной сложности от длины списка.

Теперь мы знаем четыре функции, на которых строятся все операции со списками: `head`, `tail`, `null` и `(:)`. Конечно же, стандартная поставка языка включает в себя сотни уже написанных функций для работы со списками. Вот только часть из них:

```
last, init, length, (!!), (++), concat, take, drop, reverse, elem, replicate
```

Проверьте самостоятельно их тип и опробуйте их на примерах входных данных. Чуть позже, когда мы будем учиться создавать свои собственные функции, мы повторим реализации некоторых из них.

Кортежные функции

Простейшие кортежные функции включают, по сути, только две функции: `fst` и `snd`:

```
fst :: (a,b) -> a
snd :: (a,b) -> b
```

Обе функции берут кортеж, только первая возвращает первый элемент кортежа, а вторая, кхм... второй, что ли? Проверим:

```
fst (23,"hello") → 23
```

```
snd (23,"hello") → "hello"
```

Отлично, а где же функция для создания кортежа? А нет такой, собственно. Если нужно создать кортеж из двух значений `x` и `y` - надо просто записать их в скобках и разделить запятыми: `(x,y)`.

Из кортежных функций нужно еще обязательно упомянуть функции `zip` и `unzip`, мы обязательно их рассмотрим позднее.

Создание своих функций

Как несложно догадаться, в функциональном языке функции играют главную роль, и поэтому способов создания функций в языке предусмотрено аж целых пять.

Способ 1. Определение функции как выражения от параметров:

```
fact x = x * fact (x-1)
```

Определение функции в данном случае состоит из имени функции и ее параметров слева - и одного выражения справа, которое показывает, какое значение должна возвращать функция, если ей передали все параметры.

Способ 2. Несколько определений одной функции:

```
fact 0 = 1
fact x = x * fact (x-1)
```

Оказывается, функцию можно задать не одним определением, а несколькими. Как компилятор или интерпретатор языка сможет разобраться, какое определение ему использовать, если вы, например, захотели вычислить значение `fact 4`? Интуитивно понятно, что для вычисления значения `fact 4` первое определение нам "не подходит". Чуть позже, когда мы будем говорить об образцах, мы уточним, что это такое "не подходит", а в самом конце курса найдем то единственное абсолютно логичное объяснение, которое и может быть в таком строгом и логичном языке, как Haskell. А пока просто запомним, что если определений несколько - берется первое из них, которое "подходит", то есть позволяет вычислить требуемое значение.

Способ 3. Определение функции через синоним:

```
productMy = product
```

Тут все просто - если уже есть какая-то функция `product`, то мы можем дать ей новое имя `productMy`. Ну, или по-другому, - можем объявить, что под именем `productMy` скрывается значение `product`, имеющее тип функции `Num a => [a] -> a`.

Ту же самую функцию можно было написать и по-другому:

```
productMy xs = product xs
```

Однако между двумя этими способами, идентичными с точки зрения выполнения, эффективности и всего прочего, есть существенная разница. Она показывает, на каком уровне думает программист, написавший их.
Вспомните пример с функцией как ящиком-преобразователем на колесиках! Во втором случае мы описываем, что наш требуемый ящик на колесиках должен делать с тем списком чисел `xs`,

который ему передали - он должен в свою очередь передать этот список функции `product`, подождать результата, и, в свою очередь, вернуть его. Мы работаем так, как привыкли в любом императивном языке - на уровне данных, передаваемых туда-сюда между функциями и переменными.
А вот в первом случае мы работаем на уровне функций! Мы описываем, как скомпоновать наш требуемый ящик на колесиках из уже имеющихся ящиков - в то время, когда никаких данных еще нет! Чувствуете разницу?

> Функциональное программирование - во многом про то, как описывать процесс обработки данных без упоминания данных вообще; как описывать нужный процесс обработки данных в терминах существующих маленьких ящичков-преобразователей.

Способ 4. Лямбда функция (анонимная функция):

```
vectlen = \ x y -> sqrt (x*x + y*y)
```

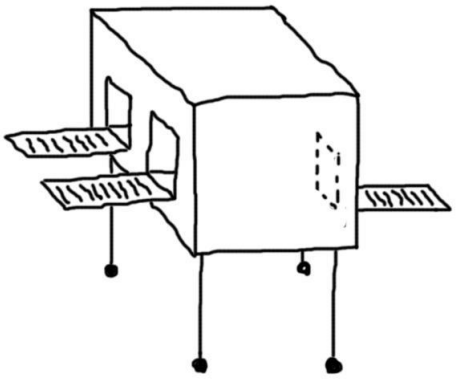
Вот, собственно, выражение справа от равенства и есть лямбда-функция. Купированный значок лямбды, за которым перечисляются параметры, и после стилизованной стрелки - единственное выражение от параметров. Обратите внимание, что мы одновременно используем и упомянутый выше третий вариант - построенной анонимной функции дается имя `vectlen`. Зачем, спросите вы - если можно написать обычную функцию:

```
vectlen x y = sqrt (x*x + y*y)
```

Дело в том, что мы дальше будем жонглировать функциями как снежками - закидывать их в функции, ловить обратно как результаты - и вовсе не всегда удобно бывает создавать отдельную функцию `vectlen` только для того, чтобы тут же первый и последний раз передать ее в качестве параметра куда-то еще. Гораздо удобнее в этом случае передать в качестве параметра именно созданную прямо там, где она понадобилась, анонимную функцию.

Способ 5. Частичное применение функции:

Частичное применение функции - одна из самых мощных концепций функционального программирования, один из тех столпов, на которых стоит восхитительно красивое здание. Самое время нам опять вернуться к нашим ящикам на колесиках.



Представим, что у нас есть ящик с двумя входными отверстиями и одним выходным. Мы можем положить в каждое отверстие по числу, они закроются, и из выходного отверстия выкатится,

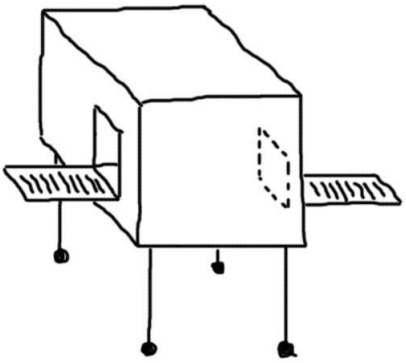
например, их сумма:

```
(+) :: Num a => a -> a -> a
```

А теперь давайте повторим эксперимент, но только пока положим в первое отверстие число: отверстие закроется, и наш ящик будет ждать второе число. А мы сделаем паузу и посмотрим, что же у нас получилось на текущий момент:

```
(+) 1 :: Num a => a -> a
```

Пока у нас нет второго числа, мы стоим перед ящиком, у которого только одно входное отверстие и одно выходное! Отдав функции двух аргументов только один из них, мы получили функцию одного аргумента!



Это видно даже просто на картинке: представьте себе функцию `a -> a -> a`, а потом отдайте ей один требуемый `a`, что у нас остается, когда первый параметр `a` исчезнет вместе со своей стрелочкой? Правильно, останется `a -> a`, что это за зверь? То-то же.
Такая ситуация, когда функции N аргументов подаются M аргументов (M<N) и в результате получается функция N-M аргументов, называется частичным применением функции, или построением остаточной процедуры.
Конечно, мы можем сразу подать и второй аргумент и получить результат, но это вовсе не обязательно, в этом то и вся идея! Полученную промежуточную функцию можно отдать куда-то еще, а можно просто дать ей имя и потом использовать там, где это будет нужно:

```
inc = (+) 1
```

Отметим еще раз: того же самого результата можно было добиться и без частичного применения:

```
inc x = (+) 1 x
```

Но мы же хотим научиться думать на уровне функций, а не на уровне данных, правда? К тому же, выражение `(+) 1 x` имеет тип `Num a => a`, и мы не сможем вставить его туда, где понадобится вставить функцию. А выражение `(+) 1` имеет тип `Num a => a -> a`, то есть является функцией и может быть использовано везде, где нужна функция.

Отмечу еще, что частичное применение операторов называется **сечением**, и имеет важную особенность. Вообще-то, если какая-нибудь обычная функция `f` принимает параметры `x y` и `z`, то ей можно передать только `x`, или передать только `x` и `y`, или передать все параметры сразу – но нельзя, например, передать только `y`.

А вот с сечениями все проще: `(^2)` есть функция, возводящая в квадрат, а `(2^)` есть функция, вычисляющая `2` в заданной степени, и отличаются эти функции тем, какой именно из параметров функции возведения в степень `(^) :: => Double -> Integer -> Double` мы передаем: первый или второй.

Давайте еще раз вернемся к функции `(+)`:

```
(+) :: Num a => a -> a -> a
```

Что мы получаем, передавая функции `(+)` только один аргумент? Функцию от оставшегося аргумента. Это означает, что тип функции `(+)` можно рассматривать не только так, как мы написали, но и по-другому:

```
(+) :: Num a => a -> (a -> a)
```

В данном случае мы явно выделяем тот факт, что `(+)` на самом деле является функцией одного аргумента! Просто вызывая `(+)` с двумя параметрами мы не успеваем заметить ту стадию вычислений, на которой появляется функция одного аргумента. Но на самом деле она есть:

```
(+) 2 3 → (2+) 3 → (2+3) → 5
```

> Более того, получается, что на любую функцию `f :: a -> b -> c -> d -> … -> y -> z` можно посмотреть как на функцию `f :: a -> (b -> (c -> (d -> … -> (y -> z)…)`. Это означает, что абсолютно все функции в Haskell имеют ровно один параметр (и один результат, который может быть чем угодно – в том числе и опять функцией). Ничего себе открытие?
> Такие функции называются "каррированными", по имени все того же великого и ужасного Хаскеля Карри (а правильнее было бы называть их - "шонфинкелированные", по имени их первооткрывателя - Моисея Эльевича Шейнфинкеля).
> В отличие от обычных, например, функций языка C++, где все функции, наоборот, некаррированные. В упомянутом C++ даже оператор сложения имеет тип примерно такой: `(+) :: (Double,Double) -> Double`, принимая кортеж и возвращая число.
> Понимаете теперь, почему в C++ вы были обязаны писать `f(x,y)`, например? Потому что функции некаррированные, и принимают кортеж.
> А в случае каррированной функции в Haskell мы обязаны писать `f x y`, и не можем писать `f (x,y)`, потому что функция принимает параметр `x`, а не кортеж `(x,y)`.

Образцы и сопоставление с образцом

Совсем скоро мы начнем писать свои функции, но прежде нам нужно разобраться с потрясающе красивым, мощным и выразительным механизмом, который называется - сопоставление с образцом.

Давайте-ка вернемся к функции, находящей факториал целого числа, и посмотрим внимательно на два ее определения, и вспомним, как они будут совместно работать:

```
fact 0 = 1
fact x = x * fact (x-1)
```

Когда кто-то вызовет эту функцию с определенным параметром, например так: `fact 5`, среда выполнения языка попробует **сопоставить** этот вызов с каждым определением, для того, чтобы

выяснить, какое определение "подходит". По сути, язык попробует сопоставить фактический параметр `5` сначала с нулем `0`, а потом и с переменной `x`. Естественно, первое сопоставление будет неуспешным, а второе успешным.
Можно считать, что и `0`, и `x` - это образцы, с которыми сопоставляется фактическое значение, пришедшее в функцию. Это пришедшее в функцию значение или "влезает" в какой-то образец, или не влезает. А что же может быть образцом?

1. Образцом может быть **константа**: с ней успешно сопоставляется только та же самая константа. Это мы как раз наблюдаем в первом определении факториала.

2. Образцом может быть **имя**: с ним успешно сопоставляется любое значение, и это имя (и полученное им значение) может быть использовано в правой части определения функции. Это мы наблюдаем во втором определении факториала. Именно поэтому, кстати, важно определения дать именно в таком порядке - иначе определение с именем будет срабатывать всегда, и рекурсия никогда не закончится.

3. Образцом может быть специальный **знак подчеркивания** "_". С ним тоже успешно сопоставляется любое значение, но при этом само значение теряется. Этот значок удобно использовать тогда, когда само сопоставленное значение нам по каким-то причинам не нужно. Например, как в гипотетической функции безопасного деления:

```
safediv _ 0 = 0
safediv x y = x/y
```

Если второй операнд равен нулю, то первый операнд нам не важен.

4. Ну и наконец, самый главный пункт, одновременно и самый важный, и самый сложный для логичного объяснения. Давайте я пока расплывчато сформулирую это так: образцом может быть **конструкция**, единственным образом раскладывающая переданное в функцию значение на составные части. Страшно? Вот пример:

```
inclist [] = []
inclist (x:xs) = x+1 : inclist xs
```

Вот как мы прочитаем эту функцию: "Функция inclist от пустого списка возвращает пустой список. А функция `inclist` от любого другого списка возвращает `x+1 : inclist xs` , где `x` - это голова этого списка, а `xs` - хвост". Как написано, так и читается, не так ли?

Давайте разбираться, почему `x:xs` - это образец. Напомню, операция `(:) :: a -> [a] -> [a]` добавляет один элемент в голову существующего списка. Так вот давайте спросим себя: пусть дан какой-то список, например, `[4,5,6]`.
Можно ли подобрать такие `x` и `xs`, чтобы `x:xs` равнялось бы как раз списку `[4,5,6]`? Конечно можно, `x=4`, `xs=[5,6]`. А всегда ли единственны такие `x` и `xs` для определенного списка? Очевидно - да, потому что любой список единственным образом "раскладывается" на одноэлементную голову и остаточный хвост.
Значит `(x:xs)` будет являться правильным образцом. С таким образцом успешно сопоставится любой непустой список; а пустой не сопоставится успешно потому, что для пустого списка вы не сможете найти такие `x` и `xs`, чтобы `x:xs` равнялось бы пустому списку `[]` (выражение `[]:[]` будет равно не `[]`, а `[[]]`, что есть список длины 1).

Сравните, кстати, как выглядела бы та же самая функция без использования образцов:

```
inclist [] = []
inclist xs = (head xs)+1 : inclist (tail xs)
```

Кажется, я говорил, что функции `head` и `tail` – базовые? Ну, почти – за исключением того, что они сами тривиально пишутся с использованием образцов:

```
head (x:xs) = x
tail (x:xs) = xs
```

Другой пример правильного образца:

```
foo (x:y:xs) = x+y : xs
```

Данный образец `(x:y:xs)` успешно сопоставится только со списком, в котором есть как минимум два элемента. И это действительно правильный образец, потому что для любого списка (длиной не меньше 2) вы можете найти только один возможный набор `x`, `y` и `xs`, чтобы `(x:y:xs)` совпадало с исходным списком.

И опять, сравним, как эта функция бы выглядела без использования образцов:

```
foo xs = (head xs)+(head (tail xs)) : tail (tail xs)
```

Польза образцов начинает проявляться, не так ли? Еще пример:

```
product [x] = x
product (x:xs) = x * product xs
```

Обратите внимание на образец из первого определения. Это корректный образец, потому что он сопоставится только со списком, состоящим из единственного элемента. Список из двух и более элементов уже "не пролезет" в узкое горлышко образца `[x]`.
Вы спросите, а что же будет, если функцию `product` вызовут от пустого списка? Ошибка времени выполнения будет, потому что среда выполнения не найдет подходящего определения и не сможет вычислить значение.

Образцам не обязательно быть списочными. Вот, например, как на самом деле написаны кортежные функции `fst` и `snd`:

```
fst (x,y) = x
snd (x,y) = y
```

Действительно, `(x,y)` является правильным образцом, потому что единственным образом раскладывает переданный в функцию кортеж на две его составляющие. Мы даже можем представить себе гипотетическую функцию, которая берет кортеж из двух списков, и возвращает кортеж из двух голов этих списков:

```
foo ((x:xs),(y:ys)) = (x,y)
```

Вся конструкция `((x:xs),(y:ys))` является одним большим образцом, потому что раскладывает пришедший кортеж из двух списков на описанные составляющие строго единственным образом: `foo ([2,3,4],[5,6,7]) → (2,5)`.

Синтаксический хлеб и синтаксический сахар

Еще немного возможностей языка, которые упрощают написание функций и делают их красивыми и самоописываемыми. Я понимаю, очень хочется, наконец, перейти к изучению заманчивых идей и самой сути функционального программирования – но тогда придется потом часто отвлекаться, чего не хотелось бы.

Условия и ограничения

```
max :: Ord a => a -> a -> a
max x y = if x > y then x else y
```

Функция, возвращающая из двух большее значение. Первая строчка определяет тип функции, но в большинстве случаев без нее можно обойтись: компилятор сможет сам вывести тип функции. При этом он постарается вывести такой тип, чтобы он подходил под как можно большее число типов. Например, тип функции `max` мог бы быть: `Integer -> Integer -> Integer`, но это означало бы, что функцию нельзя применить ни к каким другим типам.

Вот как примерно будет думать компилятор: "Так, посмотрим, что у нас внутри функции делается с двумя параметрами? Ага, к ним применяется операция сравнения `(>)`, а какие у нее требования? Она требует любой тип `a`, принадлежащий классу `Ord`. Стало быть `x` и `y` оба принадлежат этому типу. А возвращает эта функция как раз или `x`, или `y`, значит, тот же тип является и возвращаемым".

Под всей этой с виду алхимией лежит на самом деле вполне серьезная математика, которая называется - система типов Хиндли-Милнера. Вообще, под всем функциональным программированием лежит серьезная математика - лямбда-исчисление и прочие чёрчи.

Возвращаясь к нашей функции: мы видим условную конструкцию языка. В отличие от оператора `if` в императивных языках, где он имеет вид `if выражение then действие else действие`, в языке Haskell конструкция `if выражение1 then выражение2 else выражение3` - это именно выражение, значение которого равно значению `выражения2` или `выражения3` в зависимости от того, равно ли `True выражение1`. Понятно, что ветка `else` обязана быть в такой конструкции, а `выражение2` и `выражение3` должны быть одного типа.

Если условий, которые надо проверять, становится много, то конструкция из вложенных типов становится очень сложной. В таком случае удобнее пользоваться такой возможностью, как ограничения:

```
sign x | x > 0          = 1
       | x < 0          = (-1)
       | otherwise      = 0
```

Здесь мы видим три определения одной функции, причем у последних двух левая часть определения до условия для краткости опущена. На самом деле здесь работает старое доброе сопоставление с образцом, только образец `x`, который раньше срабатывал всегда, теперь срабатывает, только если прилагаемое к нему условие равно `True`. Красивое решение, которым, я уверен, очень гордятся разработчики языка (я бы гордился на их месте) - слово "`otherwise`". Это совсем не какая-то особенная часть языка, как слова `then` или `else`. Где-то глубоко в стандартных библиотеках написано, оцените юмор:

```
otherwise = True
```

Так что, это просто замена такому условию, которое срабатывает всегда.

Локальные определения

Простая функция, находящая решение квадратного уравнения:

```
solve a b c
    | d < 0 = []
    | otherwise = [(-b + sqrt d)/2/a, (-b - sqrt d)/2/a] where
        d = b*b - 4*a*c
```

Заметили это `where`? После него идет локальное определение имени `d`, используемого в теле основной функции. Можно было бы объявить глобальную функцию:

```
d a b c = b*b - 4*a*c
```

Но тогда и вызывать мы ее пришлось с параметрами, - да и пригодится ли где-то еще значение дискриминанта, кроме как внутри решения квадратного уравнения? Различных функций и констант в секции `where` может быть много, и все они видны только из главной функции, к которой относится `where`. Зато они легко могут использовать параметры главной функции, что здесь и продемонстрировано.

Есть и другая альтернатива - выражение `let определения in выражение`:

```
solve a b c =
      let
            d = b*b - 4*a*c
      in
            if d < 0 then [] else [(-b + sqrt d)/2/a, (-b - sqrt d)/2/a]
```

Здесь мы сначала даем все нужные локальные определения, а затем уже используем их в теле функции. Еще одно отличие заключается в том, что секция `where` включает локальные определения, тогда как `let определения in выражение` само по себе является выражением, имеющим значение - то есть может подставляться в функции и появляться везде, где может быть выражение.

Двумерный синтаксис

Вы уже заметили, наверное, что в языке Haskell нет ни точек с запятыми, разделяющих операторы, ни `begin ... end` или фигурных скобок, группирующих операторы. Хотя на самом деле потребность в группировке имеется.
Посмотрите еще раз на определение функции `solve` в версии с `where`. Откуда следует, что `d = b*b - 4*a*c` есть локальное определение, а не просто следующая за `solve` функция? Компилятор понимает это благодаря такой вещи, как двумерный синтаксис.
То, с какой колонки начинается первое определение за словом `where`, определяет все, что будет к этому `where` относиться. Как только вы вернетесь обратно к тому столбцу, с которого начинается сама функция `solve`, компилятор поймет, что локальные определения кончились.
Это очень удобный подход, который избавляет от необходимости брать все локальные определения в фигурные скобки `{}` и разделять их точками с запятой "`;`". Хотя это тоже возможно, конечно, если по каким-то очень странным причинам вам захочется в одной строчке дать

несколько определений функций.

Арифметические последовательности

Раз списки – главная структура данных в функциональных языках, то не удивительно, что возможностей по их созданию и обработке есть очень много. Вот, например, удобная возможность для создания арифметических последовательностей. Думаю, все абсолютно ясно из примеров:

```
[1..10] → [1,2,3,4,5,6,7,8,9,10]
[1,3..10] → [1,3,5,7,9]
[10,9..1] → [10,9,8,7,6,5,4,3,2,1]
[1..] → [1,2,3,4,5,6,7,8,9,10,…
```

В последнем случае мы случайно создали бесконечный список натуральных чисел. Если мы попробуем его вывести на экран, то так и не дождемся конца. Как с такими списками работать, мы рассмотрим чуть позже. А пока нужно сказать, что арифметические последовательности на самом деле не ограничены целыми числами. Можно использовать многие другие типы (на самом деле только те, что входят в класс `Enum`), правда, иногда с неожиданными последствиями:

```
['a'..'f'] → "abcdef"
[0.0,3.5..10.0] → [0.0,3.5,7.0,10.5]
```

Замыкания списков

Еще одна очень удобная особенность языка, связанная с генерацией и обработкой списков – это так называемые замыкания списков. Появилась она в языке из-за мощного лобби математиков, которые вообще сначала хотели, чтобы все их математические определения без переписывания оказались работающими программами (шутка, конечно, хотя…). Оцените сами:

```
{x² | x ← N, x <= 10}
[x^2 | x <- [1..10]]
```

Первое на языке математиков означает множество квадратов натуральных чисел, меньших `10`. Второе на языке Haskell означает список квадратов натуральных чисел, меньших или равных `10`. В общем случае, замыкание списков имеет вид:

```
[выражение | образец <- список, образец <- список, … , условие, условие, … ]
```

Замыкание создает список по следующей схеме: из каждого списка берется по элементу, и если все условия выполняются, то из этих элементов выражение генерирует новый элемент выходного списка.

```
[x+1 | x <- [1..5]] → [2,3,4,5,6]
[(x,x+1) | x <- [1..5]] → [(1,2),(2,3),(3,4),(4,5),(5,6)]
```

Как видим, выражение, генерирующее список, может быть довольно нетривиальным. Если списков задано несколько, то выражение генерирует новый элемент списка по всевозможным комбинациям значений из списков-источников:

```
[(x,y) | x <- [1..4], y <- [1..4], x < y] → [(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
```

Генераторы списков – очень удобная вещь, но в процессе начального обучения функциональному

программированию я бы рекомендовал ими не пользоваться, именно потому, что они позволяют обойти проблему, а не научиться решать ее в функциональном стиле.

Функциональное мышление

Теперь, вооружившись целым арсеналом способов создания функций, умением использовать образцы и знанием об удобных и приятных синтаксических возможностях языка Haskell, мы начнем, собственно, писать свои собственные функции, и в процессе изучать эти самые синтаксические возможности.
На самом деле, синтаксис языка – это всегда отражение идей, которые в этом языке скрыты. И барьер, который вам вот-вот предстоит преодолеть, связан не с синтаксисом, а с мышлением. Мышление требуется в функциональных языках – ну просто совсем другое. Будьте готовы начать мыслить по-новому!

Рекурсия как основное средство

Из чего состоят программы в императивных языках? Из присваиваний, проверки условий и переходов. Ну и еще из циклов, которые на самом деле есть все это вместе взятое. Вместо присваиваний и переходов в функциональных языках есть рекурсия. Давайте вспомним еще раз функцию нахождения длины списка:

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

Как мыслил создатель этой функции? Предположим, нам дали какой-то список. Список – это такая хитрая структура данных, у которой видна только голова и хвост, причем длина хвоста не видна. Чтобы посчитать длину, надо пройтись по каждому элементу… Нет, у нас нет такой возможности! Вот идея! Надо посчитать длину хвоста, и к ней прибавить единицу. Постойте, а как посчитать длину хвоста? Надо воспользоваться той же самой функцией!
В итоге, функция запускает себя саму, передавая все время хвост списка, и этот хвост будет уменьшаться и уменьшаться, и в итоге выродится в пустой список. Значит, надо отдельно обработать случай, когда требуется вычислить длину пустого списка.

Решение задачи над списком с помощью рекурсии всегда заключается в следующем вопросе самому себе. Предположим, нужно решить какую-то задачу для списка `xs`. Предположим, что кто-то нам дал ответ на нашу задачу, но для списка `tail xs`. Сможем ли мы тогда сразу легко дать ответ и для всего списка `xs` тоже?
Посмотрите, ведь именно такая логика скрыта в функции `length`! Как задача нахождения длины списка `(x:xs)` решается при условии, что уже известна длина списка `xs`? Да просто прибавлением единицы!

В целом, рекурсивный способ решения задачи сводится к тому, что задача разбивается на подзадачи, для которых запускаются другие (или те же самые) функции. Если подзадачи всегда оказываются меньше размером, чем исходная задача, то такой подход приведет к тому, что в итоге функция дойдет до некоторого числа простейших случаев, в каждом из которых легко сразу дать ответ.

Пусть мне нужно решить задачу `K`. Предположим, что я умею решать задачи `L`, `M` и `N`, которые меньше по размеру, чем задача `K`. Можно ли, зная решения задач `L`, `M` и `N` создать на них основе решение для задачи `K`? Если можно, то у нас сразу готов рекурсивный алгоритм решения нашей задачи.

> Математики тут вспомнят доказательство по методу индукции, а программисты вспомнят методы динамического программирования. Можно еще вспомнить о связи динамического программирования и рекурсий, эффективные и неэффективные рекурсии и хранение результатов промежуточных задач.

Вот, оцените, как такой подход (разделяй и властвуй, в рекурсивной реинкарнации) позволяет элегантно написать функцию сортировки (первое кунг-фу из введения):

```
sort [] = []
sort (x:xs) = sort [y | y <- xs, y <= x] ++ [x] ++ sort [y | y <- xs, y > x]
```

Что тут происходит? Если нужно отсортировать пустой список, то ничего делать не нужно: результатом является пустой список. Если нужно отсортировать список, состоящий из головы `x` и хвоста `xs`, то результатом является:

- отсортированный список из всех таких элементов `y` из `xs`, которые **не больше** `x`, плюс…
- список из самого элемента `x`, плюс…
- отсортированный список из всех таких элементов `y` из `xs`, которые **больше** `x`.

Кстати, а что у нас с типом функции, сможет ли компилятор сам его восстановить? Где мы дали подсказку компилятору о типе элементов нашего списка `xs`? Да вон же, там где мы сравнивали элементы друг с другом! Раз к элементам применяется операция сравнения, значит тип этих элементов должен обязательно принадлежать классу `Ord`:

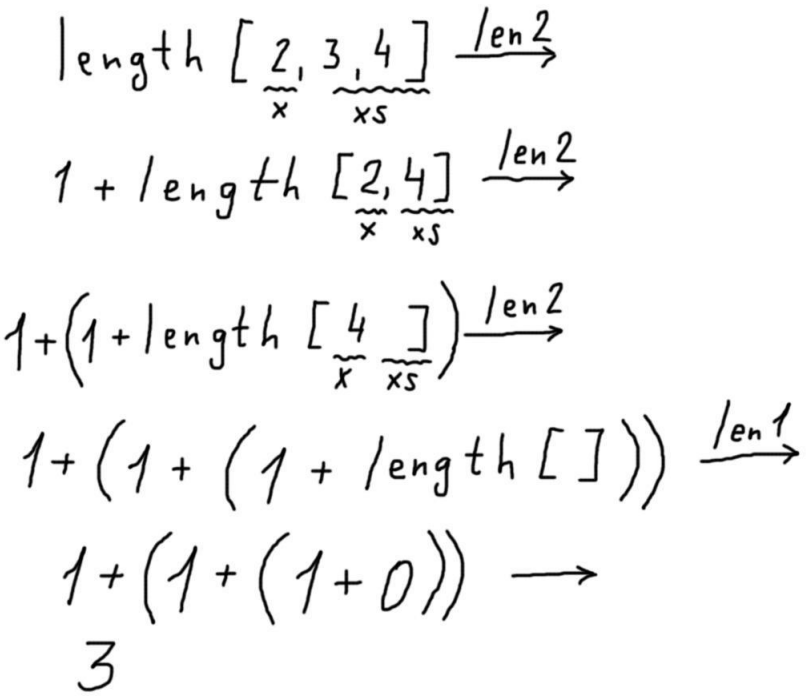
```
sort :: Ord a => [a] -> [a]
```

Ручная редукция выражений

Если вы хотите посмотреть, как определенная функция строит свой результат, нужно выключить голову и вручную заняться преобразованием выражений по заданным определениям функций. Часто это необходимо, чтобы понять, **что** функция делает вообще, или почему она что-то делает не так, как надо.
Ничего особенно умного здесь нет, поэтому и нужно отключить голову: чтобы не вкладывать в процесс вычислений наши "хотелки" и интуицию, а действовать строго по правилам. Помните школьную алгебру и преобразование алгебраических выражений? Вот и здесь то же самое, только не надо думать, какую формулу применить. Давайте посмотрим, как работает функция `length`:

```
length [2,3,4] → 1 + length [3,4] → 1 + (1 + length [4]) → 1 + (1 + (1 + length [])) →
1 + (1 + (1 + 0)) → 1 + (1 + 1) → 1 + 2 → 3
```

В таком процессе удобно над стрелочками "→" делать надписи, показывающие, какое определение какой функции мы сейчас применяем, - а под фактическими данными подписывать обозначения формальных параметров. Например, в первом случае, когда мы применяем второе определение функции length, `x` оказывается равен `2`, а `xs` – `[3,4]`.



Вот тот же самый процесс для функции `sort` (я опущу некоторые промежуточные стадии, которые очевидны):

```
sort [2,1,0,3,4] →
sort [1,0] ++ [2] ++ sort [3,4] →
(sort [0] ++ [1] ++ sort []) ++ [2] ++ (sort [] ++ [3] ++ sort [4]) →
([0] ++ [1] ++ []) ++ [2] ++ ([] ++ [3] ++ [4]) →
[0,1,2,3,4]
```

Думаем функционально, шаг раз

Давайте потренируемся в написании простейших списочных функций, которые мы упоминали выше. Длину мы находить умеем. А как насчет последнего элемента списка? Если у нас есть список `x:xs`, то как нам найти его последний элемент? Надо найти последний элемент списка `xs`, он и будет ответом! А раз список `xs` меньше, чем наш исходный список, то через конечное число шагов мы получим ответ. Все, что нам остается – это сделать "заглушку", описать поведение функции в простейшем случае, до которого неизбежно дойдет рекурсия:

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

В первом определении образец `[x]` сработает только в случае, если в функцию передадут список ровно с одним элементом (а благодаря рекурсии, этот случай рано или поздно случится). Проверим?

```
last [1,2,3] → last [2,3] → last [3] → 3
```

А как насчет добавления элемента в конец списка?

```
appendElem :: a -> [a] -> [a]
appendElem x [] = [x]
```

```
appendElem x (y:ys) = y : appendElem x ys
```

Посмотрите еще раз на эту функцию, очень важно ее понять и пропустить через себя. Что мы здесь делаем? Мы разбираем исходный список на голову и хвост, запускаем ту же самую функцию от хвоста, и к результату спереди приписываем обратно голову. Вот как это работает:

```
appendElem 7 [1,2,3,4,5,6] →
1 : appendElem 7 [2,3,4,5,6] →
1 : (2 : appendElem 7 [3,4,5,6]) →
1 : (2 : (3 : appendElem 7 [4,5,6])) →
1 : (2 : (3 : (4 : appendElem 7 [5,6]))) →
1 : (2 : (3 : (4 : (5 : appendElem 7 [6])))) →
1 : (2 : (3 : (4 : (5 : (6 : appendElem 7 []))))) →
1 : (2 : (3 : (4 : (5 : (6 : [7]))))) →
1 : (2 : (3 : (4 : (5 : [6,7])))) →
1 : (2 : (3 : (4 : [5,6,7]))) →
1 : (2 : (3 : [4,5,6,7])) →
1 : (2 : [3,4,5,6,7]) →
1 : [2,3,4,5,6,7] →
[1,2,3,4,5,6,7]
```

Видите, как функция `appendElem` "вгрызается" в список `[1,2,3,4,5,6]` словно червяк в горох, а добравшись до самого дна списка, делает свое черное дело? Собственно, функция `last` поступала точно так же, - только она отбрасывала элементы списка, потому что они ей были не нужны. А функция `appendElem` почти что "бережно" оставляет за собой все пройденные элементы.

> Можно сказать, что функция `appendElem`, вызывая саму себя, оставляет позади в стеке нечто, что можно назвать "контекст вычислений". Дойдя до конца списка и вставив туда нужный элемент, функция завершается, возвращая управление самой же себе с прошлого шага рекурсии, прибавляя контекст (последнюю встреченную "голову"), делает еще один шаг назад, и так далее (вернее, и так обратно).
> Важно, что самой функции `appendElem`, когда она доходит до конца, не приходится явно думать "а что же там у было то раньше" - это за нее сделает вся развернутая позади в стеке система вызовов. Мы будем часто этим пользоваться.

Что там у нас на очереди? Ага, функция сливания, или конкатенации, двух списков `(++)`. Прежде чем давать определение этой функции, давайте оценим сложность проблемы. У нас есть два списка, `xs` и `ys`, и надо создать третий список, в котором сначала будут элементы из списка `xs`, а потом элементы из списка `ys`. А что у нас есть в арсенале? Да только операция `(:)` добавления одного элемента в список. Ну и написанная функция `appendElem`, добавляющая элемент в конец списка.

Идея! Надо брать по одному элементы из `ys` и добавлять в конец `xs`! А если `ys` кончился, то вот он и ответ! Итак, на каждом шаге алгоритма прибавляем к `xs` текущую голову `ys` и запускаем функцию заново:

```
slowConcat :: [a] -> [a] -> [a]
slowConcat xs [] = xs
slowConcat xs (y:ys) = slowConcat (appendElem y xs) ys
```

Проверим как работает?

```
slowConcat [1,2,3] [4,5,6] →
slowConcat (appendElem 4 [1,2,3]) [5,6] →
slowConcat [1,2,3,4] [5,6] →
slowConcat (appendElem 5 [1,2,3,4]) [6] →
slowConcat [1,2,3,4,5] [6] →
slowConcat (appendElem 6 [1,2,3,4,5]) [] →
slowConcat [1,2,3,4,5,6] [] →
[1,2,3,4,5,6]
```

Вроде работает. Я специально при первой возможности выполнял функцию appendElem, чтобы не получить запись на полстраницы.

> На самом деле, какую из функций в выражении раскрывать первой – это совершенно отдельная проблема, называемая "стратегии редукции выражений".

Думаю, вы уже догадались, почему я так назвал эту функцию. Проблема в том, что функция `appendElem` пробегает по всем элементам своего списка, и, таким образом, имеет линейную сложность `O(n)`. А раз `slowConcat` вызывает ее столько раз, сколько есть элементов в списке, общая сложность функции `slowConcat` получается квадратичной $O(n^2)$, и это, разумеется, никуда не годится.

Почему так получилось? Потому, что мы мыслили так, как привыкли думать в императивном программировании. Вот примерно какой алгоритм мы реализовали:

```
zs := xs;
пока ys не пустой
      xs := xs + head ys;
      ys := tail ys;
```

> На самом деле, уже на этом примере можно понять, почему в Haskell можно так легко обойтись без присваиваний, и почему функциональные языки по вычислительной силе равны языкам императивным.
> Роль изменяемых переменных в функциональных языках переходит к параметрам функции, вызываемой рекурсивно, видите?

Давайте попробуем зайти с другой стороны. Мы по одному брали элементы из `ys`: рассматривали два списка `xs=[1,2,3] ys=[4,5,6]` как `xs=[1,2,3] ys=4:[5,6]`, и смотрели, что надо сделать с головой и хвостом списка `ys`.

А что если наоборот? Давайте рассмотрим списки `xs` и `ys` как `xs=1:[2,3] ys=[4,5,6]`. Казалось бы, зачем мы отделили голову от списка `xs`, - мы ведь ничего с этой единицей сделать не можем. Здесь уместно сделать большую театральную паузу.

А с хвостом? Что если вызвать ту же самую функцию для сливания списков `[2,3]` и `[4,5,6]`, поможет ли нам это? Конечно! Ведь нам останется только добавить обратно единичку к голове получившегося списка, а это операция практически моментальная!

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

Проверим, как теперь работает эта функция:

```
(++) [1,2,3] [4,5,6] →
1 : (++) [2,3] [4,5,6] →
1 : (2 : (++) [3] [4,5,6]) →
1 : (2 : (3 : (++) [] [4,5,6])) →
1 : (2 : (3 : [4,5,6])) →
1 : (2 : [3,4,5,6]) →
1 : [2,3,4,5,6] →
[1,2,3,4,5,6]
```

Видите, как теперь (++) играет роль червяка, вгрызающегося в горох-список? Оставляя за собой ~~в стеке вычислительный контекст~~ информацию о том, что нужно еще после вычислений добавить обратно элементы. Теперь все нормально – линейная алгоритмическая сложность нас в данном случае устраивает.

Думаем функционально, шаг два: аккумуляторы

Еще одна очень важная списочная функция: функция переворачивания списка, которая первый элемент ставит последним, второй – предпоследним, и так далее.

Следуя той логике, которую мы только что описали, эту функцию следует реализовать так:

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

Если вы так написали, то я вас поздравляю: вы успешно освоили первую ступень кунг-фу функционального программирования. Вы начали думать рекурсивно. Но по названию моей функции вы уже, наверное, догадались, в чем проблема. Да, функция `slowReverse` тоже имеет квадратичную сложность, потому что она вызывает саму себя `N` раз (где `N` – длина списка), и на каждом шаге вызывает функцию `(++)`, которая тоже имеет линейную сложность по первому аргументу.
Что же делать? Может быть, поступить аналогично функции `(++)`?

```
quasiReverse [] = []
quasiReverse (x:xs) = x : quasiReverse xs
```

Думаю, вы уже поняли, что эта функция вообще ничего не делает, и возвращает список в неизменном состоянии. Что же делать? Может быть, как-то откусывать от списка `xs` элементы по одному и складывать их в какой-то промежуточный список? Но какой список, где его хранить, и как его изменять?

И вот в этом творческом тупике к нам на помощь и придет еще один важный прием

функционального программирования. Помните, я уже говорил о том, что место локальных переменных в рекурсивном программировании занимают изменяемые локальные параметры? Так давайте введем дополнительный параметр, и будем в нем **накапливать** откусываемые головы от списка `xs`. Раз в этом параметре будет что-то накапливаться, то называться он будет – **аккумулятор**!

```
reverse :: [a] -> [a]
reverse xs = reverse2 xs []

reverse2 :: [a] -> [a] -> [a]
reverse2 [] acc = acc
reverse2 (x:xs) acc = reverse2 xs (x:acc)
```

Почему у нас две функции, `reverse` и `reverse2`? Потому что пользователю нужна функция `reverse` безо всяких лишних параметров, а функции `reverse` нужна вспомогательная функция с параметром. Вот как они работают:

```
reverse [1,2,3] →
reverse2 [1,2,3] [] →
reverse2 [2,3] (1 : []) →
reverse2 [3] (2 : (1 : [])) →
reverse2 [] (3 : (2 : (1 : []))) →
(3 : (2 : (1 : []))) →
[3,2,1]
```

Видите теперь, почему в аккумуляторе оказывается в итоге весь список в перевернутом виде? Потому что в начале вычисления функции аккумулятор пуст, а в процессе выполнения в него по одному заносятся элементы исходного списка – но заносятся всегда в начало!

Я предлагаю вам сейчас остановиться и еще раз посмотреть на тот прием, что мы применили. Вы должны научиться так думать, чтобы подобные приемы сами собой всплывали у вас в голове при решении задач со списками и не только. Сравните две вот эти строчки из функций `quasiReverse` и `reverse2`:

```
quasiReverse (x:xs) = x : quasiReverse xs
reverse2 (x:xs) acc = reverse2 xs (x:acc)
```

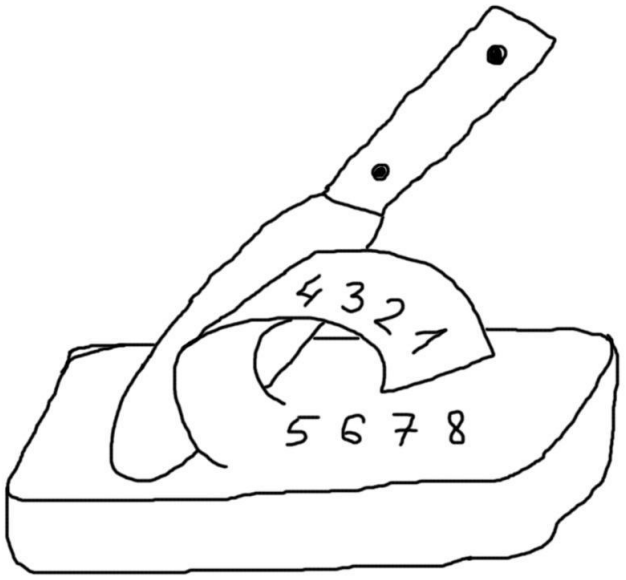
Давайте добавим во вторую функцию ничего не значащий и ничего не делающий аккумулятор:

```
quasiReverse (x:xs) acc = x : quasiReverse xs acc
reverse2 (x:xs) acc = reverse2 xs (x:acc)
```

И переименуем обе функции, чтобы действительная разница между двумя строчками была очевидна:

```
foo (x:xs) acc = x : foo xs (    acc)
foo (x:xs) acc =     foo xs (x : acc)
```

Что делает первая функция? Она пользуется потоком рекурсивного выполнения для того, чтобы пробежать по всем элементам списка, пропустить их все через себя и оставить без изменений. Вторая же функция пользуется силой потока рекурсивного выполнения для того, чтобы сохраняя по одному элементы в аккумуляторе, вывернуть наизнанку весь список!



Еще раз аккумуляторы мы вспомним чуть позже, когда будем говорить про хвостовую рекурсию.

Реализация простейших списочных и не только списочных функций

Давайте еще потренируемся в написании простейших списочных функций, которые мы упоминали.

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

Функция `init` возвращает все элементы списка, кроме последнего. Вы видите, как она это делает: пропускает без изменений все элементы списка – а как только у нее остается список из одного элемента – сразу возвращает пустой. В итоге, последний элемент списка в результат и не попадает.

Еще одна функция – вернуть из списка элемент по его номеру. Как это сделать, если функцию, к примеру, просят вернуть пятый элемент, а функция, глядя на список, может обратиться только к его голове или к хвосту, но не ко второму, третьему или пятому элементу?
Элементарно! Функция просто должна рекурсивно вызвать саму себя от хвоста списка, потребовав вернуть номер с элементом на единицу меньше, чем от нее самой хотят:

```
(!!) :: [a] -> Int -> a
(!!) [] _ = error “index too large”
(!!) (x:xs) 0 = x
(!!) (x:xs) n = (!!) xs (n-1)
```

Функция `error` прерывает выполнение функции, выводя сообщение об ошибке. Мы должны это сделать, если на каком-то шаге выяснилось, что от нас требуют вернуть элемент с каким-то номером из пустого списка. Обратили внимание, как параметр `n` с каждым шагом становится на единицу меньше? Вот как это будет работать:

```
(!!) [1,2,3] 2 →
(!!) [2,3] 1 →
(!!) [3] 0 → 3

(!!) [1,2,3] 6 →
```

```
(!!) [2,3] 5 →
(!!) [3] 4 →
(!!) [] 3 → error
```

Функция `(++)` сливала два списка в один. А что если у нас целый список списков, как слить его в один? Конечно же, для этого можно написать функцию `concat`, которая, в свою очередь, будет рекурсивно вызывать функцию `(++)`:

```
concat :: [[a]] -> [a]
concat [] = []
concat (list : lists) = list ++ concat lists
```

Из очень часто используемых списочных функций нужно выделить функции `take` и `drop`. Первая выбирает из списка первые несколько элементов, а вторая наоборот, выкидывает из списка первые несколько элементов:

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

drop :: Int -> [a] -> [a]
drop 0 xs     = xs
drop _ []     = []
drop n (_:xs) = drop (n-1) xs
```

Обратите внимание, и `take` и `drop` корректно обрабатывают все терминальные случаи параметров, которые у них могут оказаться. Ну и еще можно обратить внимание на хитрый образец `(_:xs)`, который использует `drop`. Конечно же, там могло быть написано `(x:xs)`, но эта запись подчеркивает, что само значение головы списка нам не интересно.

Ну и наконец, функция проверки принадлежности элемента списку:

```
elem :: Eq a => a -> [a] -> Bool
elem x [] = False
elem x (y:ys) = if x == y then True else elem x ys
```

Если в какой-то момент голова оказалась равна текущему элементу, то поиск прекращаем с положительным результатом. Иначе результат поиска зависит от (точнее, заключается в) результата поиска в хвосте списка. Можно было функцию записать и так:

```
elem x [] = False
elem x (y:ys) = (x == y) || elem x ys
```

А можно было обойтись без конструкции `if`:

```
elem x [] = False
elem x (y:ys) | x==y      = True
              | otherwise = elem x ys
```

Еще несколько околосписочных функций. Функции суммы и произведения списка чисел:

```
sum :: Num a => [a] -> a
sum [] = 0
sum (x:xs) = x + sum xs
```

```
product :: Num a => [a] -> a
product [] = 1
product (x:xs) = x * product xs
```

Обратите внимание, что функции суммы и произведения работают со списком любого типа `a`, относящегося к классу `Num`, то есть с любым числовым типом.

Максимальное и минимальное число из списка:

```
max :: Ord a => a -> a -> a
max x y = if x > y then x else y

min :: Ord a => a -> a -> a
min x y = if x < y then x else y

maximum :: Ord a => [a] -> a
maximum [x] = x
maximum (x:xs) = max x (maximum xs)

minimum :: Ord a => [a] -> a
minimum [x] = x
minimum (x:xs) = min x (minimum xs)
```

Функцию нахождения максимума из списка можно было написать и по-другому:

```
maximum [x] = x
maximum (x:y:xs) = if x > y then maximum (x:xs) else maximum (y:xs)
```

Логические функции `and` и `or` принимают список значений типа `Bool` и возвращают `True`, в случае если все (`and`) или одно (`or`) значение из списка равно `True`. Реализация этих функций для вас теперь тоже, надеюсь, тривиальна:

```
and :: [Bool] -> Bool
and [] = True
and (x:xs) = x && and xs

or :: [Bool] -> Bool
or [] = False
or (x:xs) = x || or xs
```

Ну и напоследок, пара кортежных функций. Функция `zip` превращает два списка значений в список кортежей из элементов, стоящих на одной и той же позиции. Например, `zip [1..5] ['a'..'z'] → [(1,'a'),(2,'b'),(3,'c'),(4,'d'),(5,'e')]`. Обратите внимание, как функция `zip` работает, если списки разной длины:

```
zip :: [a] -> [b] -> [(a,b)]
zip (a:as) (b:bs)   = (a,b) : zip as bs
zip  _      _       = []
```

А вот обратную функцию `unzip` написать сложнее и поучительнее:

```
unzip :: [(a,b)] -> ([a],[b])
```

Функция должна принять список кортежей, разделить каждый кортеж на элементы, и сложить

элементы в отдельные списки. Проблема тут в том, как вы понимаете, что никаких промежуточных переменных для складывания списков у нас нет. И, разумеется, это и есть подсказка – вместо таких переменных нужно использовать два дополнительных аккумулятора. Ну и `reverse`, чтобы избавиться от ненужных эффектов переворачивания списков (разберитесь сами, почему так просходит).

```
unzip list = unzip' list [] []

unzip' [] as bs = (reverse as, reverse bs)
unzip' ((a,b):list) as bs = unzip' list (a:as) (b:bs)
```

Обратили внимание, какой сложный образец был использован? Пришедший список кортежей был разложен на первый кортеж `(a,b)` и список остальных кортежей `list`, а первый кортеж был дополнительно разложен на его две составляющие `a` и `b`. Попробуйте представить, как выглядела бы эта функция без использования образцов и в виде одного определения? Фу, гадость:

```
unzip list as bs =
    if null list then
        (as,bs)
    else
        unzip (tail list) (fst (head list) : as) (snd (head list) : bs)
```

Лисп напоминает, только скобок еще добавить.

Думаем функционально, шаг три: хвостовая рекурсия

Хвостовая рекурсия – это один из широко применяемых методов в функциональном (и не только) программировании. Понимание и свободное использование хвостовой рекурсии является признаком того, что вы уже вполне разбираетесь в функциональном решении задач.

Не Совсем Формальное, Но Вполне Совсем Страшное Определение:

Хвостовая, или остаточная рекурсия – это вид рекурсии, при котором значение, возвращаемое функцией `F` при заданных параметрах `P`, определяется по одному из трех вариантов:

1. `F(P) = g(P)`, где `g(P)` – это какая-то функция, при вычислении которой не используется вызов функции `F`. Этот вариант определяет нерекурсивное завершение вычисления функции `F`, хотя само по себе это вычисление может быть сложным;
2. `F(P) = F(h(P))`, где `h(P)` – это какая-то функция, при вычислении которой так же не используется вызов функции `F`. Этот вариант определяет рекурсивное вычисление функции `F`.
3. `F(P) = if u(P) then r1(P) else r2(P)`, где `u(P)` - это какая-то функция, задающая условие, при вычислении которой не используется вызов функции `F`; а выражения `r1(P)` и `r2(P)` в свою очередь удовлетворяют вариантам 1,2 или 3.

Вы, конечно, пропустили формальное определение, и правильно сделали, потому что сейчас я объясню то же самое на пальцах. В чем суть хвостовой рекурсии? В том, что если функция вычисляется рекурсивно, то ее значение определяется только значением самой функции, а не значением выражения, в которое функция входит как один из операндов.

Хотите еще более простое определение? Есть их у меня. Обычно, если функция запускается рекурсивно, то из рекурсии потом приходится возвращаться обратно, чтобы "дособрать" все, что

осталось в прошлых вызовах. В итоге, возвращаемое значение получается только тогда, когда все рекурсивные вызовы, наконец, вернулись обратно.

А хвостовая рекурсия – это такая рекурсия, из которой никогда возвращаться не надо – итоговое значение формируется где-то там, в самой глубине рекурсии.

Давайте вспомним для примера функцию вычисления факториала:

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Вот например как вычисляется факториал от трех:

```
factorial 3 →
3 * (factorial 2) →
3 * (2 * (factorial 1)) →
3 * (2 * (1 * (factorial 0))) →
3 * (2 * (1 * 1)) →
3 * (2 * 1) →
3 * 2 →
6
```

Видите, как рекурсия "провалилась" в самую глубину, нашла там единичку, - но потом пришлось возвращаться из всех скобок обратно, и результат получился только тогда, когда мы вернулись из всех рекурсивных вызовов.
Почему так? Да потому, что при рекурсивном вызове функции `factorial (n - 1)` требуется запоминать, что результат надо еще домножить на `n`. Кому нужен формализм – выражение `n * factorial (n - 1)` не подходит ни под один пункт определения.

В чем, собственно, проблема-то? В том, что вызов такой функции может привести к большому расходу памяти и, в частности, к переполнению стека. В более сложных функциях, при наивной их реализации, рост используемой памяти может быть даже не линейным, как в случае с факториалом, а экспоненциальным!

Вот, например, простая реализация вычисления числа Фибоначчи:

```
fib :: Integer -> Integer
fib 1 = 1
fib 2 = 1
fib n = fib (n-1) + fib (n-2)
```

Вычисляя хотя бы шестое число Фибоначчи, получим ужасную последовательность вызовов:

```
fib 6 →
fib 5 + fib 4 →
(fib 4 + fib 3)+(fib 3 + fib 2) →
((fib 3 + fib 2)+(fib 2 + fib 1))+((fib 2 + fib 1)+1) →
(((fib 2 + fib 1)+1)+(1+1))+((1+1)+1) →
(((1+1)+1)+(1+1))+((1+1)+1) →
8
```

Сколько раз тут вычислялось первое число Фибоначчи? Очевидно, что попытка вычисления большого числа Фибоначчи очень быстро переполнит стек, можете проверить.

Вернемся к функции вычисления факториала `factorial`.

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Очевидно, что второе ее определение не соответствует определению хвостовой рекурсии. Можно ли как-нибудь переписать эту функцию таким образом, чтобы второе выражение приняло вид:

```
factorial … = factorial (…)   ?
```

Конечно можно, но для этого придется воспользоваться аккумулятором! Определим новую функцию уже двух переменных `factorial_a` следующим образом:

```
factorial_a :: Integer -> Integer -> Integer
factorial_a 0 acc = acc
factorial_a n acc = factorial (n-1) (acc*n)
```

Вызовем эту функцию с параметрами 3, 1 и получим следующую последовательность вызовов:

```
factorial_a 3 1 →
factorial_a 2 (1*3) →
factorial_a 1 (3*2) →
factorial_a 0 (6*1) →
6
```

Как видим, на каждом шаге рекурсии не приходится запоминать каких-либо значений, а только вычислять параметры функции перед вызовом ее. Никакого "расползания" выражения при вычислении не происходит, потому что результат вычисления функции всегда строится как строго только вызов той же самой функции, пусть и от других параметров.

Легко увидеть, что такое новое определение функции `factorial_a` соответствует определению хвостовой рекурсии, так как функция в обоих своих определениях вычисляется или как простое выражение от параметров функции (`factorial_a … = acc` в первом определении), или как значение той же функции с другими параметрами, вычисление которых не включает вычисление функции `factorial_a` в процессе (`factorial_a … = factorial (n-1) (acc*n)` во втором определении). Память в стеке при такой рекурсии расходуется только на хранение адресов возврата значения функции.

Каким образом тогда будет реализована сама функция `factorial`? Очевидно, что через функцию `factorial_a` следующим образом:

```
factorial :: Integer -> Integer
factorial n = factorial_a n 1
```

> Хотя при программировании на функциональном языке (по крайней мере, при обучении) о реализации вычислений, о том, как будет работать компилятор или интерпретатор, стоит думать в самую последнюю очередь, стоит все-таки упомянуть о реализации хвостовой рекурсии с точки зрения компилятора. При реализации хвостовой рекурсии, вычисления могут выполняться при помощи итераций в постоянном объеме памяти. На практике это обозначает, что "хороший" транслятор функционального языка должен "уметь" распознавать хвостовую рекурсию и реализовывать её в виде цикла, а не рекурсии вообще.

> И еще одно замечание, вдогонку. Как известно, функциональная и императивная парадигмы программирования равномощны в смысле разрешаемости задач. Любая задача, разрешимая на функциональном языке, может быть разрешена и на императивном – и наоборот. В частности при доказательстве этого утверждения придется показать соответствие между основными конструкциями императивного языка (присваиваниями, ветвлениями и циклами) и функционального языка. Так вот, циклы императивных языков и соответствуют рекурсии в функциональных языках, и на примере хвостовой рекурсии это становится видно более явно.

Как может выглядеть общая схема преобразования функции в хвостовую рекурсию?

1. Вводится новая функция с дополнительным аргументом (аккумулятором), в котором накапливаются результаты вычислений.
2. Начальное значение аккумулирующего аргумента задается в равенстве, связывающем старую и новую функции.
3. Те равенства исходной функции, которые соответствуют выходу из рекурсии, заменяются возвращением аккумулятора.
4. Равенства, соответствующие рекурсивному определению, выглядят как обращения к новой функции, в котором аккумулятор получает то значение, которое возвращается исходной функцией.

В некоторых случаях хвостовую рекурсию и выдумывать не надо – она рождается сама по себе, как единственный очевидный способ реализации требуемой функции. Вот, например, наши определения из прошлой главы, которые сразу являются хвостовой рекурсией:

```
(!!) :: [a] -> Int -> a
(!!) [] _ = error "index too large"
(!!) (x:xs) 0 = x
(!!) (x:xs) n = (!!) xs (n-1)

drop :: Int -> [a] -> [a]
drop 0 xs     = xs
drop _ []     = []
drop n (_:xs) = drop (n-1) xs

elem :: Eq a => a -> [a] -> Bool
elem x [] = False
elem x (y:ys) = if x == y then True else elem x ys
```

Еще раз о рекурсии

В заключение, в этой главе я хотел бы еще раз вернуться к тому, о чем мы говорили чуть раньше:

> Решение задачи над списком с помощью рекурсии всегда заключается в следующем вопросе самому себе. Предположим, нужно решить какую-то задачу для списка `xs`. Предположим, что кто-то нам дал ответ на нашу задачу, но для списка `tail xs`. Сможем ли мы тогда сразу легко дать ответ и для всего списка `xs` тоже?

Это, на самом деле, очень важная идея, и я хотел бы еще раз проиллюстрировать ее на поучительном примере. Возьмем опять функцию `length`:

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

Ну кто, скажите, в здравом уме будет при написании этой функции рассуждать в ключе "а что если кто-то уже вычислил длину хвоста списка"? Этот пример ни разу не поучительный по той самой причине, что нашему разуму самого начала понятно, как посчитать длину списка: что там считать-то, тыкай пальцем в каждый элемент и называй следующее натуральное число! И такая рекурсивная запись ничего разуму не дает – просто еще один странный способ ~~выдирания гланд через желудок~~ вычисления того, что всем ясно, как вычислять.

Давайте рассмотрим нетривиальный пример! Пусть есть мальчишки во дворе, обозначаемые буквами, и нужно найти все возможные способы разделения их на две команды. Нам нужна функция такого типа:

```
divide :: [a] -> [([a],[a])]
```

Она берет список, например, наших символов, и возвращает – вглядитесь в эту мешанину круглых и квадратных скобок – список кортежей вида `([a],[a])`, где на первом месте стоит список всех тех, кто попал в первую команду, а на втором месте – кто попал во вторую.

> На всякий случай. Я в курсе, что можно перебрать все бинарные представления чисел от 0 до 2^N. Вы ведь понимаете, что речь совсем не об этом, правда? Мы учимся искать рекурсивные решения. Когда вы этому научитесь, то сможете в каждом случае решать, нужно ли рекурсивное решение, или удобнее нерекурсивное.

Простой случай для функции: если есть только я, то я могу образовать собой единственным или одну команду, или другую:

```
divide [me] = [ ([me],[]) , ([],[me]) ]
```

А что дальше? Ну как, мозг, есть варианты? Вот здесь, где пасует наше интуитивное алгоритмическое мышление, и выходит на передний план рекурсивный способ во всей его красе. Что если есть "я" и "все остальные", как нам разделиться всеми возможными способами на две команды?
Так вот – предположим, что "все остальные" уже умеют делиться на команды, и уже поделились всеми возможными способами! А как же я? Меня-то забыли! Надо модифицировать все возможные способы с учетом того, что еще есть я!

```
divide (me:others) = join me (divide others) where
```

Вот оно, мы свели задачу к другой: пусть есть набор всевозможных разделений мальчишек на команды, - как этому набору добавить еще и меня? Это и будет делать локальная функция `join`:

```
    join :: a -> [([a],[a])] -> [([a],[a])]
```

Она берет меня и всевозможные деления на команды, и должна создавать новые всевозможные деления, - но уже с учетом меня.

```
    join me [] = []
    join me ((side1,side2):divisions) = [(me:side1,side2),(side1,me:side2)]
                                        ++ join me divisions
```

Итак, если есть список возможных разделений, то нужно его рекурсивно обработать. Берем каждое разделение без учета меня `(side1,side2)`, и в результат складываем два возможных разделения уже с учетом меня. Одно, где я присоединяюсь к первой команде, а второе – где я иду

во вторую команду: `[(me:side1,side2),(side1,me:side2)]`.

Вот, собственно, и все. Если выкинуть определения типов, то задача решается в 4 строчки:

```
divide [me] = [ ([me] , []) , ([] , [me]) ]
divide (me:others) = join me (divide others) where
    join me [] = []
    join me ((side1,side2):divisions) = [(me:side1,side2),(side1,me:side2)]
                                        ++ join me divisions
```

Проверим?

```
SomeModule> divide "abc"
[("abc",""), ("bc","a"), ("ac","b"), ("c","ab"),
("ab","c"), ("b","ac"), ("a","bc"), ("","abc")]
```

Чем этот пример поучителен? Тем, что наше интуитивное умение строить в голове алгоритмы здесь откровенно пасует, пытаясь сразу представить себе алгоритм построения решения. Все, что интуиция может – это охватить один шаг: если есть 10 вариантов разбиения на команды для какой-то группы людей, то в случае, если появляется еще один кто-то, у нас становится 20 вариантов разбиения, потому что в каждом варианте новый человек может пойти либо в одну команду, либо в другую.
Интуиция может охватить только один шаг алгоритма – но большего то и не надо! Все остальное сделает правильно запущенная рекурсия!

Полезные хитрости языка

Настало время рассказать о некоторых полезных хитростях языка Haskell, которые, опять-таки, во многом отражают идеи функционального программирования.

Ленивые вычисления и строгие функции

Haskell - крайне ленивый язык. Идея ленивости, на самом деле, очень проста: никакое выражение не будет вычисляться, пока его результат не понадобится. Более того, мы видели, что вычисление рекурсивных функций, если его производить руками, превращается в последовательное раскрытие одних выражений во вторые, вторых в третьи, и так далее. Часто бывает так, что уже частично раскрытого выражения достаточно для того, чтобы функция могла вернуть результат - в таком случае, выражение так и не будет вычислено до конца.

Классический пример - функция `const`:

```
const :: a -> b -> a
const x _ = x
```

Обратили внимание, что функция не использует свой второй параметр вообще? Работает она так:

```
const "hello" 1 → "hello"
const "hello" False → "hello"
const "hello" (1/0) → "hello"
```

Обратили внимание на последний вызов? В качестве второго параметра мы передаем ошибочное, нетерминальное выражение - а функция все равно возвращает правильный результат. Если бы мы работали с обычными неленивыми, строгими функциями, то передавая в функцию

нетерминальное выражение, мы бы получали в ответ тоже нетерминальное выражение.

> Немного занудства, позволите? Если обозначать ошибочное, нетерминальное (вычисление которого никогда не закончится) выражение как `bot`, то строгие функции - это те, которые всегда сами сразу виснут или выкидывают ошибку, если им такое значение передать. То есть для строгих функций, `f(...,bot,...) → bot`. Для нестрогих, соответственно, это может быть так, а может быть и не так.

Haskell - язык ленивый. Он не станет вычислять значение выражения `(1/0)` в надежде, что оно никогда не понадобится - и в данном случае его лень оказывается оправданной. А функции Haskell, соответственно - нестрогими. Конечно, если бы нетерминальное значение передавалось в качестве первого параметра - мы получили бы ошибку:

```
const (1/0) "hello" → Program error: {primDivDouble 1.0 0.0}
```

Однако и в этом случае ошибки можно избежать - достаточно просто не требовать выводить значение функции `const` на экран, и вообще, не требовать использовать его как-либо:

```
length [const (1/0) "hello"] → 1
```

В данном случае, функции `length` не требуется вычислять значение функции `const`. От функции `length` только требуют выяснить, сколько значений находится в переданном ей списке - а это значение одно: `const (1/0) "hello"`, вне зависимости от того, вернет эта функция одно значение, список значений или что-либо еще. А значит, функция `length` не будет вычислять результат этого выражения.

Бесконечные списки

Ленивые вычисления, кроме своих прочих особенностей (например, вы не можете быть уверены, что нечто будет вычислено в тот или иной момент, или в том или ином порядке), обладают мощной возможностью: они позволяют создавать бесконечные структуры данных, например, бесконечные списки.

Помните арифметические последовательности? Выражение `[1..]` означало бесконечный список натуральных чисел, но что это за зверь, и откуда он взялся - было совершенно не понятно. А вот посмотрите на функцию `repeat`:

```
repeat :: a -> [a]
repeat x = x : repeat x
```

Вот что будет, если попробовать вычислить ее значение и вывести на экран:

```
repeat 1 →
1 : repeat 1 →
1 : (1 : repeat 1) →
1 : (1 : (1 : repeat 1)) → ...
```

В общем, вы поняли - функция никогда не завершится. И вы будете смотреть на появляющиеся на экране единички, которые никогда не кончатся. Но ведь не обязательно выводить результат этой функции на экран? Можно спросить у нее, например, первый элемент, или первые несколько элементов:

```
head (repeat 1) → 1
take 5 (repeat 1) → [1,1,1,1,1]
```

Вот, например, полезная функция: `replicate`, дублирующая заданное значение заданное количество раз. Как можно ее реализовать?

```
replicate :: Int -> a -> [a]
replicate 0 _ = []
replicate n x = x : replicate (n-1) x
```

А можно и воспользоваться функцией `repeat`:

```
replicate :: Int -> a -> [a]
replicate n x = take n (repeat x)
```

А как получить тот самый список натуральных чисел?

```
naturals = naturalsFrom 1
naturalsFrom n = n : naturalsFrom (n+1)
```

Большая часть списочных функций, которые мы с вами писали, будет работать не только с обычными, но и с бесконечными списками. Но некоторые, разумеется, не будут - такие как `length` и `last` - и вполне понятно, почему.

> Еще точнее: какие-то функции, если им передать бесконечный список, вернут конечный результат (например, `head`). Другие, если им передать бесконечный список, вернут тоже бесконечный список, с которым вполне можно работать, если не вычислять его целиком (например, `drop`). А третьи, если им передать бесконечный список, вообще ничего не вернут и зависнут (например, `last`). Обдумайте разницу.

Функция show

Функция `show` используется тогда, когда нужно значение определенного типа превратить в строку, вот ее тип:

```
show :: Show a => a -> String
```

Предлагаю вам вместе со мной порадоваться этой логичности языка: функция `show` умеет показывать (преобразовывать в строку) значения только тех типов, которые относятся к классу `Show`, то есть умеют преобразовываться в строку. А что - логично:

```
show 1 → "1"
show True → "True"
show "hello" → "\"hello\""
```

Классу `Show` принадлежит большинство простейших типов, и, что еще важнее, производные от них сложные структуры. Например, если система умеет отображать число (а она умеет), то она умеет отображать и список чисел, как и список любых других отображаемых значений - для этого система напечатает открывающую квадратную скобку "`[`", потом через запятую покажет преобразованные в строку элементы списка, а потом закрывающую скобку "`]`". То же касается и кортежей:

```
show [1..3] → "[1,2,3]"
show (3, "hello", [1,2,3]) → "(3,\"hello\",[1,2,3])"
```

На самом деле, когда вы печатаете выражение в командной строке интерпретатора, он вычисляет его значение, а потом преобразовывает его в строку, чтобы вывести на экран. Именно поэтому, если типом выражения будет нечто странное (например, функция), вы увидите ошибку, связанную с тем, что функции `show` не удается правильно работать с элементами этого типа:

```
Prelude> sin
ERROR - Cannot find "show" function for:
*** Expression : sin
*** Of type    : Double -> Double
```

Совсем немного о классах

Давайте вспомним, как у нас появилось понятие класса типов. Мы разбирались, применительно к каким типам должна работать, например, операция `(>)`, и решили, что ограничивать ее каким-то одним типом нельзя, но и работать со всеми возможными типами эта операция, очевидно, не может. Тогда и выяснилось, что работать операция `(>)` может только над типами, входящими в класс `Ord`, то есть над типами, допускающими сравнение на больше-меньше. Довольно тавтологично, не так ли?

```
(>) :: Ord a => a -> a -> Bool
```

В прошлой главе мы встретили функцию `show`, которая умеет показывать (преобразовывать в строку) значения тех типов, которые относятся к классу `Show`, то есть умеют преобразовываться в строку. Давайте посмотрим на сам класс `Show`:

```
class Show a where
    show      :: a -> String
…
```

Когда определяется класс, задаются все функции, которые должен реализовывать тип, для того, чтобы иметь право принадлежать этому классу. Классы в Haskell – это **что-то вроде интерфейсов**, или абстрактных классов во многих развитых объектно-ориентированных языках: они перечисляют операции, но не задают никакой реализации. Последнее, правда, для Haskell не совсем верно, вот посмотрите на полное объявление класса `Eq`:

```
class Eq a where
    (==), (/=) :: a -> a -> Bool

    -- Minimal complete definition: (==) or (/=)
    x == y      = not (x/=y)
    x /= y      = not (x==y)
```

Класс `Eq` заявляет: все, кто хочет сертифицироваться на принадлежность мне, должен будет предоставить мне одно из двух – или свою операцию равенства, или операцию неравенства. То бишь, если ты какой-нибудь `Integer`, то расскажи классу `Eq` о том, как сравнивать два значения своего типа, и получи отметку о полном служебном соответствии.

> Записка на полях: кстати, а иккс-иггрек-зеет одинаков со строкой "Зачет внимательным читателям", запомните это – вдруг пригодится?

И что тогда? А тогда операция (==) сможет сравнивать и значения твоего типа `Integer`. Вот он, полиморфизм в стиле Haskell в действии. Видите аналогию? В терминах ООП можно сказать, что операция `(==)` работает с типами, у которых есть перекрытая реализация полиморфной функции сравнения.

А что же с классом `Show`? Если у вас есть какой-нибудь новосозданный тип данных, то функция `show` с ним работать, конечно, не будет. А вот если вы расскажете, как ваш тип преобразовывать в строку, то получите сертификат о том, что ваш тип теперь принадлежит классу `Show`, и тогда функция `show` сможет его показывать. Разобрались в тавтологиях?

Позже, когда будем создавать свои собственные типы, мы увидим конкретные примеры.

Функция read

Функция `read` в некотором смысле обратна функции `show` и используется тогда, когда нужно строку преобразовать в значение определенного типа. Вдумайтесь в ее тип:

```
read :: Read a => String -> a
```

Оценили? Функция получает строку, а возвращает значение произвольного (принадлежащего какому-то классу Read) типа `a`. Как такое может быть? Какой тип в итоге функция вернет?

Попробуем вызвать эту функцию, чтобы преобразовать строку в число:

```
Prelude> read "54"
ERROR - Unresolved overloading
*** Type       : Read a => a
*** Expression : read "54"
```

Если вы прочитали прошлую главу про классы, то можете догадаться, что не нравится интерпретатору. В случае с функцией `show` он знал, какую конкретно перегруженную функцию незаметно от пользователя нужно вызвать: если вызывали `show 5`, он вызывал какую-нибудь функцию типа `primitiveIntToString`, если вызывали `show True`, он вызывал что-то вроде `primitiveBoolToString`. А если вызывали `show sin`, то он ругался, потому что не находил нужной примитивной функции, так как тип `Double -> Double` к классу `Show` не относится.

Короче говоря, в случае `show` было ясно, какой конкретно тип нужно преобразовать в строку. В случае функции `read`, интерпретатор не знает, значение какого типа нужно прочитать из строки. А как он это может узнать? А по тому, как мы будем с этим значением работать! Вот, например:

```
read "54" + 1 → 55
```

В данном случае контекст позволяет интерпретатору выявить, что преобразовать `"54"` нужно именно в число, потому что дальше с этим значением что-то складывается. Ну и, в конце концов, можно и напрямую попросить значение отнести к конкретному типу:

```
read "54" :: Integer → 54
read "54" :: Double → 54.0
```

Функция error

После функции `read`, тип функции `error` будет уже вполне понятен:

```
error :: String -> a
```

Эта функция принимает строку, а возвращает значение вообще произвольного типа. Раз так, то вставить ее можно в любую функцию – она везде по типу подойдет:

```
head [] = error “Program error: {head []}”
head (x:xs) = x

myDiv _ 0 = error “Program error: {primQrmInteger _ 0}”
myDiv x y = div x y
```

И в первом, и во втором случае функция error подходит по типу, а значение какого конкретно типа возвращается – уже не важно, потому что при вычислении значения этой функции, программа прерывается с ошибкой.

Побочные эффекты и функция trace

Говорят, что функция производит (имеет) побочный эффект, если ее результат может зависеть не только от ее собственных параметров, но и от ее окружения. Побочный эффект имеют функции, которые в процессе своих вычислений читают или модифицируют какие-то глобальные переменные, производят ввод-вывод (потому что это тоже есть чтение портов или модификация какой-нибудь видеопамяти). Если функцию с побочным эффектом вызвать дважды с одними и теми же входными значениями, она вполне может вернуть разный результат – потому что ее окружение, от которого она зависит, могло измениться.
Так вот, все функции, с которыми мы имели дело до сих пор в Haskell, побочных эффектов не имеют, и иметь не могут. Вы в принципе не сможете из своей функции что-то вывести на экран в процессе вычислений.
Это, конечно, ужасно неудобно, потому что как тогда отлаживать свои функции? Однако есть мнение, и автор с ним согласен, что потребность в трассировке в Haskell возникает гораздо реже – и именно потому, что большинство функций принципиально не могут иметь побочных эффектов.

> Ну и еще, конечно, из-за строгой системы типов, через которую поначалу сложно продраться, но благодаря которой очень часто функция скомпилированная (с десятого раза) работает так, как нужно с первого раза. А какое соотношение у вас, в вашем сиплюсплюсе?

Однако возможности для трассировки все равно есть. Для этого нужно к вашему модулю подключить модуль `Trace` (или просто подгрузить этот модуль в командную строку командой `:l Trace` или `:l Debug.Trace` или `import Debug.Trace`, в зависимости от используемой среды), и тогда у вас появится функция `trace`:

```
trace :: String -> a -> a
```

Эта функция берет определенную строку, значение произвольного типа `a`, и возвращает то же самое значение, предварительно распечатав в консоли переданную строку:

```
Trace> trace "five" 5
five5

Trace> xx where xx = [(trace (show x) x)^2 | x <- [1..3]]
[11,24,39]
```

Что происходит в последней строчке? Генератор `x <- [1..3]` создает значение `1`, вызывается функция `trace (show 1) 1`, которая распечатывает строку `"1"` и возвращает тоже `1`, число `1` возводится в квадрат и выводится на экран. Потом генератор `x <- [1..3]` создает значение `2`, вызывается функция `trace (show 2) 2`, которая распечатывает строку `"2"` и возвращает тоже `2`, число `2` возводится в квадрат и выводится на экран, и так далее.

Однако при более глубоком рассмотрении, если вспомнить о ленивости языка, возникает очень много вопросов:

- что является результатом вычисления выражения `xx`?
- как выводится на экран результат вычисления выражения `xx`?
- что выводится на экран в процессе вычисления выражения `xx`?

Это все очень разные вопросы, и вот, например, откуда это хорошо видно:

```
Trace> xx ++ xx where xx = [(trace (show x) x)^2 | x <- [1..3]]
[11,24,39,1,4,9]

Trace> sum xx where xx = [(trace (show x) x)^2 | x <- [1..3]]
12314
```

В последней строке `"12314"`, выведенной на экран, `"123"` вывела функция `trace` при обработке списка `[1,2,3]`, а `"14"` – это значение выражения `sum [1,4,9]`. Думаете, при таком нетривиальном поведении функции `trace` в связке с ленивыми функциями, вам будет легко найти глюк в своей функции? Ну-ну.

Функции высших порядков

Функции высших порядков в Haskell и вообще в функциональном программировании – это один из самых важных механизмов, которые позволяют коду быть кратким и выразительным, и вообще, - функциональное программирование стоит изучать хотя бы только из-за этой великолепной идеи. Даже если вы никогда в жизни не будете применять функциональный язык, идея функций высших порядков навсегда изменит то, как вы думаете.

Мотивация

Всем известно, что дублирование кода – это зло. Всем известно, что очень похожие куски кода, отличающиеся только мелочами, стоит оформлять как функции, и просто вызывать эти функции. Например, пусть у вас есть две функции:

```
pow2 x = x^2
pow3 x = x^3
```

Разумеется, гораздо лучше будет даже в этом простейшем случае ввести и использовать вместо двух этих функций одну более универсальную:

```
pow x y = x^y
```

Давайте рассмотрим более сложный случай. Вот вам несколько функций, разберитесь (методом пристального взгляда, или методом ручной редукции выражений), что они делают:

```
foo [] = []
foo (x:xs) = x + 1 : foo xs
```

```
boo [] = []
boo (x:xs) = x * 2 : boo xs

moo :: [[a]] -> [a]
moo [] = []
moo (x:xs) = head x : moo xs

goo [] = []
goo (x:xs) | x < 0     = x+1 : goo xs
           | otherwise = x   : goo xs

hoo [] = []
hoo (x:xs) = sum xs : hoo xs
```

Первая функция увеличивает каждый элемент списка на единицу, вторая увеличивает каждый элемент в два раза. Третья функция составляет новый список из первых элементов каждого подсписка (например, `moo ["Hello","World"] → "HW"`), а четвертая функция увеличивает на единицу каждый отрицательный. Пятая функция создает список сумм из всех постфиксов переданного ей списка (например, `hoo [1,2,3,4] →[9,7,4]`).

Вопрос: не кажется ли вам, что эти пять функции **похожи**? Может быть, даже **слишком похожи** для того, чтобы быть пятью разными функциями, а не одной общей?

Что в этих функциях общего, давайте смотреть. Все эти функции работают со списками (правда, с разными). Все эти функции, если им передать пустой список, возвращают тоже пустой список. Все? При более внимательном рассмотрении можно заметить, что все эти функции идут по списку, отделяя голову от хвоста, вычисляя на основании этих данных новый элемент результирующего списка, и запуская самих себя рекурсивно.

В случае с функциями возведения в квадрат и в куб было очевидно, где у тех двух функций общая часть, а где изменяемая. И что мы делали? Посмотрите, мы создавали новую функцию, у которой эта изменяемая часть (показатель степени) была параметром.

Давайте попробуем привести и эти функции к как можно более общему виду, чтобы было очевидно, где у них общая часть, а где параметр:

```
foo [] = []
foo (x:xs) = (\t -> t+1) x : foo xs

boo [] = []
boo (x:xs) = (\t -> t*2) x : boo xs

moo [] = []
moo (x:xs) = (\t -> head t) x : moo xs

goo [] = []
goo (x:xs) = (\t -> if t < 0 then t+1 else t) x : goo xs

hoo [] = []
hoo (x:xs) = sum xs : hoo xs
```

Пардон, если вы так боитесь лямбда-функций, давайте обойдемся без них:

```
foo [] = []
foo (x:xs) = process x : foo xs where process t = t+1

boo [] = []
boo (x:xs) = process x : boo xs where process t = t*2

moo [] = []
moo (x:xs) = process x : moo xs where process t = head t

goo [] = []
goo (x:xs) = process x : goo xs where process t = if t < 0 then t-1 else t

hoo [] = []
hoo (x:xs) = sum xs : hoo xs
```

Думаю, теперь все очевидно, не так ли? Параметром, то есть изменяемой частью, является та функция, которую мы применяем к каждому очередному элементу. Это было скрыто синтаксисом кода, но мы смогли докопаться до сути.

Функция map

Итак, мы теперь должны написать функцию, которая будет применять к каждому элементу списка какую-то другую функцию. Эта другая функция будет передаваться в качестве параметра. Что должна наша функция (давайте назовем ее `map` – от английского слова, означающего "отображение") принимать? Очевидно – преобразующую функцию (функция `process`, или функция `f`, или как мы ее там назовем – не важно) и список элементов.
Что должна функция делать, если ей передали пустой список? Очевидно, возвращать пустой список. Что она должна делать, если ей передали непустой список? Делать все то же, что и ее конкретные прародители: применять функцию преобразования к каждому элементу и запускать саму себя рекурсивно:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x) : map f xs
```

Скобки в выражении `(f x)`, конечно, не нужны – я их поставил только чтобы выделить логически ту часть, которая у нас изменялась.

Как теперь можно написать нашу функцию `foo` с использованием `map`? А надо просто создать и передать внутрь map такую функцию `process`, которая как раз сделает то, что нужно с каждым элементом:

```
foo xs = map process xs where process t = t+1
```

Уже лучше! Давайте пересилим свой страх перед лямбда-функциями, и увидим, что создавать функцию `process` и давать ей имя только для того, чтобы передать ее внутрь функции `map` – есть излишество. Можно просто обойтись лямбда-функцией и заодно понять, наконец, зачем эти лямбда-функции нужны: именно для того, чтобы передавать их в другие функции:

```
foo xs = map (\t -> t+1) xs
```

Отлично, еще более короткое определение! Если так пойдет, то оно и вообще сократится до пары слов. Вы будете смеяться, но мы этого и добиваемся. Давайте еще раз посмотрим, что нам нужно?

Нам нужна функция, которая принимает число и увеличивает его на единицу. У нас уже есть функция `(+)`, но она принимает два числа и складывает их, а не одно число. Чувствуете, куда я клоню? Что если функции `(+)` передать только один параметр? Принцип частичного применения, он же способ номер 5 создания функций, говорит о том, что в результате будет функция от одного аргумента!

```
(+1) :: Num a => a -> a
```

А значит, зачем нам лямбда-функция вообще? У нас уже есть выражение `(+1)`, значением которого и является нужная нам функция!

```
foo xs = map (+1) xs
```

Ну и остался последний шаг. У нас есть функция map, которая принимает функцию, список и возвращает список:

```
map :: (a -> b) -> [a] -> [b]
```

А что если мы функции `map` передадим не два параметра, а только один, то есть функцию `(+1)`?

```
map (+1) :: Num a => [a] -> [a]
```

Передав функции `map` только один параметр, мы получили функцию, которая принимает только один параметр: список чисел и возвращает тоже список чисел. Постойте, так это ведь и есть нужная нам функция! Значит все, что нам остается – только дать ей какой-то синоним:

```
foo = map (+1)
```

Сравните с тем, что было:

```
foo [] = []
foo (x:xs) = x + 1 : foo xs
```

Не правда ли, получилось слегка короче? Заметьте, кстати, что функции `foo` теперь не требуется обрабатывать случай пустого списка – это за нее сделает функция `map`.

Кстати, сравним опять две эквивалентные конструкции:

```
foo xs = map (+1) xs
```

```
foo = map (+1)
```

Мы как бы взяли и стерли из обеих частей этого определения формальный параметр `xs`. Всегда ли так можно делать? Ну, если говорить формально, то если вы пришли на каком-то этапе к определению, которое можно представить в виде

```
foo a b c d e f = boo g h d e f,
```

то можно последние три параметра стереть, по правилам частичного применения функций:

```
foo a b c = boo g h
```

А что же насчет остальных функций? Запишутся ли и они с помощью функции map? Конечно запишутся:

```
foo = map (+1)

boo = map (*2)

moo = map head

goo = map (\t -> if t < 0 then t-1 else t)

hoo = map sum
```

В случае функции `goo` не нашлось подходящей функции, которую можно было бы или прямо подставить (как `head` для функции `moo`), или взять частичное применение (как `(+)` и `(*)` для `foo` и `boo`). Вот тут лямбда-функции как раз и незаменимы.

А вот с функцией `hoo` возникла какая-то проблема. Запросим ка ее тип у интерпретатора:

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

Вот ничего себе! Она требует список списков чисел, и возвращает список чисел. Проверим ее на примерах входных данных:

```
map sum [[1,2,3],[4,5,6]] → [6,15]
```

Получается, что `map sum` применяет функцию `sum` к каждому элементу списка. А значит, таковым элементом списка обязан быть список чисел. Но получается, что `map sum` делает совсем не то, что делала наша функция:

```
hoo [] = []
hoo (x:xs) = sum xs : hoo xs
```

Почему вдруг так? А давайте-ка опять вернемся к той ситуации, когда мы все функции записывали еще без `map`, но уже с помощью лямбда-функций:

```
foo [] = []
foo (x:xs) = (\t -> t+1) : foo xs

...

hoo [] = []
hoo (x:xs) = sum xs : hoo xs
```

Функция `hoo` уже, вроде как, записана достаточно просто, и поэтому мы ее не стали переписывать через лямбду. А зря: если бы записали – поняли бы, в чем проблема:

```
hoo [] = []
hoo (x:xs) = (\t -> sum t) xs : hoo xs
```

Думаю, вы уже догадались: то, что делает с каждым элементом списка функция `hoo`, нельзя представить в виде функции `(\t -> что-то такое только от t)`, применяемой к голове `x`.

В чем же заключается идея функции `map`? В чем ее смысл, суть? В чем схема ее работы? Какие функции можно переписать с использованием функции `map`, а какие нельзя?

Давайте еще раз вернемся к определению функции `map` и подумаем:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x) : map f xs
```

Что делает функция `map`? Пустой список оставляет пустым, это мы видим. А непустой? Отделяет по одному элементы от непустого списка, и преобразует их с помощью переданной функции `f`, помещая по одному в том же порядке в результирующий список… Постойте, а **как** преобразует?

Какие данные имеет функция `f`? На основании чего она может решить, как преобразовывать очередной переданный ей элемент? Вот она, проблема: у функции `f` есть только то, что ей передали. А передают ей всякий раз – только один очередной элемент, и **никакой** дополнительной информации.

Следовательно, с помощью функции `map` можно написать только такое преобразование списка, при котором каждый элемент списка изменяется независимо от других, на основании только своего собственного значения. Ни значения соседних элементов, ни позиция самого элемента в списке – не могут использоваться при преобразовании этого элемента.

Очень важно видеть в стоящей перед вами задаче ее схему и понимать, укладывается эта задача в схему функции `map`, или не укладывается.

Увеличить каждый элемент в два раза? Легко: каждый элемент меняется только на основании своего значения, а значит `map` можно использовать: `map (*2)`.

Увеличить каждый символ в строке (капитализировать строку)? Легко: `map toUpper`.

Маленькие буквы строки увеличить, а большие уменьшить? Легко: каждый символ меняется только на основании своего значения (пусть и с условием – важно, что условие это опять-таки только на его собственное значение), а значит `map` можно использовать: `map (\c -> if isUpper c then toLower c else toUpper c)`.

Преобразовать каждое число `N` из списка в целый список из `N` единиц? Легко! Каждое число преобразуется в список из единиц независимо, поэтому это все та же схема: `map (\x->replicate x 1)`.

Узнать, все ли числа в списке больше нуля? Это тоже схема функции `map`, - хотя только этой функцией уже не обойтись: `all (map (>0) xs)`, убедитесь сами:

```
map (>0) [1,2,-3,-4,5] → [True, True, False, False, True]
all [True, True, False, False, True] → False
```

Увеличить положительные элементы из списка на единицу, а остальные вообще выкинуть? Легко! Каждый элемент преобразуется независимо от других, поэтому это опять `map`, а именно: `map (\x -> if x > 0 then x+1 else … else … else … блин)...`
Что же должно быть в секции `else`? Что-то ведь обязательно должно быть, потому что функция обязана возвращать значение. Может быть, там должен быть пустой список `[]`? Не пойдет, потому

что выражение `if` всегда должно возвращать значение одного и того же типа, а в данном случае это не список, а число…

В чем проблема? Да в том, что это уже **не схема** работы функции `map`! Посмотрите опять на определение функции `map` – из самого определения видно, что функция `map` возвращает новый список с ровно тем же количеством элементов.

Прибавить к каждому элементу списка следующий элемент? Это опять не функция `map`, потому что преобразование элемента требует информации о следующем элементе, чего `map` предоставить не может.

Занулить те числа в списке, которые совпадают со своим порядковым номером? Это опять не map, потому что у элемента нет информации о том, под каким номером он в списке находится… А что если к каждому элементу прикрепить такую информацию?

```
[1,4,3,5,5] → [1,4,3,5,5]
zip [1,4,3,5,5] [1..] → [(1,1),(4,2),(3,3),(5,4),(5,5)]
```

Посмотрите, что мы сделали: мы список чисел превратили в список кортежей, где на первом месте стоит число из исходного списка, а на втором месте – порядковый номер. И вот теперь функция `map` вполне подходит: вся необходимая информация для обработки кортежа в этом кортеже имеется (оцените, кстати, как в лямбда-функции удачно использованный образец позволяет обойтись без кортежных функций `fst` и `snd`):

```
map (\(x,n) -> if x==n then 0 else x) [(1,1),(4,2),(3,3),(5,4),(5,5)] → [0,4,0,5,0]
```

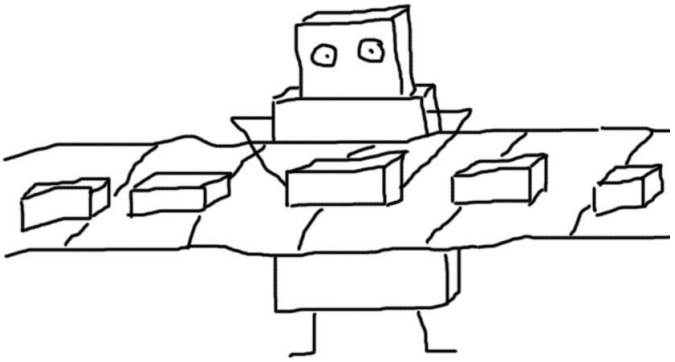
Итого, функция, которая нам нужна, записывается целиком так:

```
zeroSome xs = map (\(x,n) -> if x==n then 0 else x) (zip xs [1..])
```

Ну, или так:

```
zeroSome xs = map f (zip xs [1..]) where f (x,n) = if x==n then 0 else x
```

В целом, функцию `map` можно представить себе как робота, вдоль которого двигается конвейер с однотипными предметами. Когда очередной предмет доезжает до робота, тот что-то с этим предметом делает по одной и той же загруженной в него программе.
При этом область видимости у этого робота очень маленькая, так что он видит всегда только один единственный предмет на конвейере, а памяти у самого робота нет.



Робот обязан каждый предмет на конвейере обработать, и ни увеличивать, ни уменьшать количество предметов на конвейере у робота возможности нет. В чем полезность такого робота, ведь без программы (трансформирующей функции `a -> b`) он не умеет ничего? Полезность его именно в том, что в него можно вставить любую, произвольно сложную программу, и больше ни о чем заботиться не надо: он обработает с ее помощью весь конвейер.

> Кстати, функция `map` будет отлично работать и с бесконечными списками! Проверьте, и убедитесь сами!

Функция filter

Рассмотрим еще несколько функций. Опять-таки, сначала разберитесь, что конкретно делает каждая из этих функций:

```
foo [] = []
foo (x:xs) | x > 0     = x : foo xs
           | otherwise = foo xs

moo [] = []
moo (x:xs) | x `mod` 2 == 0 = x : moo xs
           | otherwise      = moo xs

boo [] = []
boo (x:xs) | x `elem` ['0'..'9'] = boo xs
           | otherwise           = x : boo xs

goo [] = []
goo (s:ss) | length s == 0 = goo ss
           | otherwise     = s : goo ss
```

Что делают все эти функции? Первая функция берет числовой список и возвращает список только положительных элементов из него. Вторая функция оставляет в числовом списке только четные числа. Третья функция выкидывает из строки (списка символов) цифры. Четвертая функция берет список строк, и выбрасывает из него пустые строки.

Все функции работают по одной и той же схеме: они проходят по всем элементам списка, для каждого элемента проверяют определенное условие, и в зависимости от результатов, либо оставляют в списке элемент, либо безвозвратно выкидывают его.

Думаю, вы уже поняли, что и эти функции следуют одному и тому же шаблону, который легко выписывается в виде еще одной очень часто используемой функции:

```
filter f [] = []
filter f (x:xs) | f x       = x : filter f xs
                | otherwise = filter f xs
```

И здесь опять параметризуется (то есть становится параметром более общей функции) то, что отличает эти четыре функции: условие, по которому принимается решение - выбросить элемент или поместить в результирующий список. Давайте теперь перепишем все функции с помощью функции `filter`:

```
foo = filter (>0)

moo = filter (\x -> x `mod` 2 == 0)

boo = filter (\x -> not (x `elem` ['0'..'9']))

goo = filter (\s -> not (null s))
```

А как насчет функции, которая бы выкидывала отрицательные числа из списка, а положительные увеличивала бы на единицу? Думаю, вы уже догадались - это **не** `filter`, потому что `filter` не может изменить элементы списка, а только выкинуть или не выкинуть.
А как насчет функции, которая бы оставляла в списке только те элементы, которые больше чем предыдущий элемент списка? Опять-таки, это **не** `filter`, потому что условие, применяемое функцией `filter` к каждому элементу списка, является функцией, которая зависит только от значения этого элемента и ни от чего другого.

Если вспомнить функцию `map`, то функцию `filter` тоже можно представить в виде робота, который идет вдоль конвейера с предметами, применяет к каждому предмету свое условие, и если условие не выполняется - выкидывает предмет с контейнера. Как и робот для функции `map`, новый робот тоже видит в каждый момент времени только один элемент, и ничего не может запоминать, переходя от одного элемента к другому.

> И функция `filter` также отлично фильтрует бесконечные списки! Правда, вычисление выражения `filter (==2) [1..]` не завершится никогда: функции `filter` неоткуда взять информацию о том, что в списке натуральных чисел число `2` встречается только один раз!

Композиция функций

Давайте еще раз посмотрим на последнюю функцию, которая выбрасывала из списка строк пустые:

```
nonEmptyStrings = filter (\s -> not (null s))
```

Что если нам наоборот нужно было оставлять только пустые, а выбрасывать все остальные? Такая функция записалась бы еще проще:

```
emptyStrings = filter null
```

Почему с непустыми строками приходится использовать лямбду? Потому что нам нужна не функция `null`, а функция противоположная ей: функция, которая возвращает `True` тогда, когда

функция `null` возвращает `False`, и наоборот. Функция `not` превращает `False` в `True`, а `True` в `False`, но функция `not` не может превратить функцию `null` в противоположную ей! Это приходится делать с помощью лямбды: `(\s -> not (null s))`. Мы конструируем функцию, которой если передать строку, то она передаст ее в функцию `null`, потом результат функции `null` передаст в функцию `not`, и уже результат той вернет.

Такое поведение встречается очень часто: когда результат работы одной функции полностью передается на вход другой функции. В математике это называется **композицией функций**.

Точнее даже не так: композицией двух функций `f` и `g` называется функция, заключающаяся в последовательном применении к чему-то функции `f`, а затем функции `g`:

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

Вот она, композиция в Haskell, - функция с крайне непонятным, на первый взгляд, типом. Если присмотреться, то видно, что функция `(.)` принимает одну функцию типа `(b -> c)`, еще одну функцию типа `(a -> b)`, и возвращает функцию типа `(a -> c)`.
Впрочем, если присмотреться по-другому, то видно, что функция `(.)` принимает одну функцию типа `(b -> c)`, еще одну функцию типа `(a -> b)`, элемент типа `a` и возвращает элемент типа `c`. Вот два эквивалентных определения функции `(.)`, отражающих оба этих взгляда на композицию:

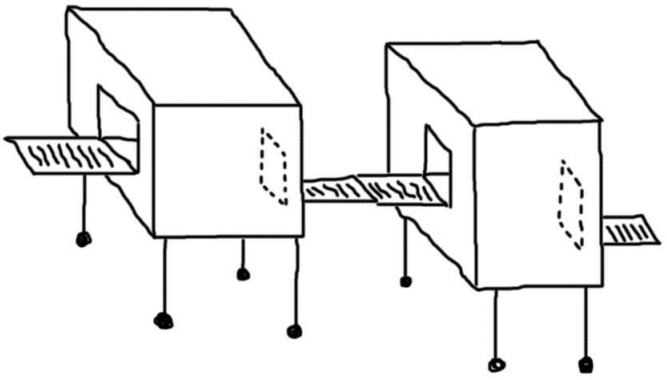
```
(.) f g = \x -> f (g x)
```

```
(.) f g x = f (g x)
```

С помощью композиции можно легко составлять сложные функции из двух простых. Например, если `h = (+1).(^2)`, то `h 5 → 26`. Ну и если нужно вычислить функцию `null`, результат ее подать на вход функции `not`, то функция, делающая сразу и то и другое с помощью композиции записывается так: `not.null`.

Обращаю ваше внимание, что функция `(.)` работает именно с функциями, а не с данными. Она берет две функции и "состыковывает" их вместе, получая третью, сложную функцию.

Если опять представить себе две функции как передвижные ящики на колесиках, у которых есть по одному входному и одному выходному отверстию, то композиция этих функций можно получить, поставив эти два ящика рядом так, чтобы выходное отверстие одного вплотную прилегало к входному отверстию другого.



Обратите внимание, что данных на этой картинке еще нет, потому что композиция - это про функции, а не про данные. Данные могут вообще никогда не появиться, а результат работы композиции - вот он уже, готов и помаргивает огоньками.

Если вдруг когда кто-нибудь бросит что-то во входное отверстие первой функции, она сразу создаст свой выход, и он автоматически попадет на вход во вторую функцию, и преобразуется ей в что-то третье. Произойдет это без всякого вмешательства с нашей стороны, и поэтому можно будет такие два состыкованные ящика рассматривать как единый ящик, единую функцию.

Итак,

```
emptyStrings = filter null

nonEmptyStrings = filter (not.null)
```

С помощью композиции можно укорачивать запись преобразований, состоящих из нескольких этапов. Например, что если нужно каждый элемент списка увеличить на единицу, потом возвести в квадрат, а потом опять увеличить на единицу? Вот варианты:

```
foo1 xs = map (+1) (map (^2) (map (+1) xs))

foo2 = map (+1) . map (^2) . map (+1)

foo3 = map ((+1).(^2).(+1))
```

Или, например, пусть нужно посчитать количество четных чисел в списке, не делящихся на 4:

```
foo1 xs = length (filter (\x -> x `mod` 2 == 0 && \x -> x `mod` 4 /= 0) xs)

foo2 = length . filter (\x -> x `mod` 4 /= 0) . filter (\x -> x `mod` 2 == 0)

foo3 = length . filter ((/=0).(`mod` 4)) . filter ((==0).(`mod` 2))
```

> Видите, как в каких-то вариантах функций нет ни одного упоминания о данных вообще? Ни одного формального параметра - только манипуляции функциями: создание нужных функций путем частичного применения и композиции, и передача этих функций в функции высших порядков, и опять композиция...

Функция foldr

И третий пример, третий набор функций. Потратьте время, разберитесь, что делает каждая из этих функций:

```
foo [] = 0
foo (x:xs) = x + foo xs

boo [] = 1
boo (x:xs) = x * boo xs

goo [] = True
goo (x:xs) = x && goo xs

moo [] = []
```

```
moo (x:xs) = x ++ moo xs

hoo [] acc = acc
hoo (x:xs) acc | x < acc   = hoo xs x
               | otherwise = hoo xs acc

doo [] acc = acc
doo (x:xs) acc | x > 0   = doo xs (acc+1)
               | otherwise = doo xs acc

zoo [] acc = acc
zoo (x:xs) acc = zoo xs (x:acc)
```

Я специально выбрал аж семь функций для того, чтобы дать вам побольше материала для анализа. Вы, конечно, уже понимаете, что требуется: найти в этих всех функциях общий шаблон поведения, общую абстракцию, общую идею – найти и формализовать в виде еще одной функции высших порядков.

Что делает каждая из этих функций? Первая находит сумму списка (то есть, другими словами, это функция `sum`), вторая – произведение (`product`), третья результат логического "и" всего списка (`and`), четвертая – конкатенация всех подсписков (`concat`). Пятая функция находит минимальное число из списка (`minimum`), шестая находит количество положительных элементов, а седьмая просто переворачивает список (`reverse`).

А что делают **все** эти функции? Опять-таки, они работают со списками, и обрабатывают элементы списка по одному, пока список не кончится. Первые четыре функции, вроде как, больше похожи друг на друга. Рассмотрим их пока отдельно.

Они "прибавляют" очередной элемент к результату вызова той же функции от хвоста списка, и отличаются смыслом этого "прибавляют" - это, соответственно: бинарное сложение, бинарное умножение, бинарное логическое "и" и бинарная конкатенация списков.
Очевидно, что именно эта отличающаяся операция, назовем ее `f`, и будет являться параметром новой функции высшего порядка. Вторым отличием первых четырех функций является то, что они возвращают при пустом списке. Это будет второй параметр функции, назовем его – нулевой элемент `z`.

Если выписать все элементы списка и посмотреть, как к ним применяется эта операция, то окажется, что результат формируется из вычисления вот такого выражения:

x_1 f (x_2 f (x_3 f … (x_N f z)

На каждом шаге работы функции бинарная функция `f` применяется к очередному элементу и результату вызова той же самой функции от хвоста списка. Математики такую операцию называют сверткой (причем **правой** сверткой). Давайте, наконец, напишем функцию:

```
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Ура! Как теперь использовать эту функцию для реализации наших семи близнецов?

```
foo xs = foldr (+) 0 xs
```

```
boo xs = foldr (*) 1 xs

goo xs = foldr (&&) True xs

moo xs = foldr (++) [] xs

hoo …
```

А что же делать с остальными функциями? Какую функцию надо передать в `foldr`, чтобы получилась функция `hoo`? Давайте немного перепишем функцию `hoo`!

```
hoo [] acc = acc
hoo (x:xs) acc = hoo xs (if x < acc then x else acc)
```

Вам ничего не напоминает это выражение в скобках? Переименуйте `acc` в `y`… Да это же тело бинарной функции `min`! Может быть, тогда так?

```
hoo xs = foldr min (head xs) xs
```

Заметьте, что мы передали в функцию `foldr` в качестве нулевого элемента `z`? Ну а что еще – не плюс бесконечность же передавать. Работает! А что с функцией `doo`? Перепишем:

```
doo [] acc = acc
doo (x:xs) acc = doo xs (if x > 0 then acc+1 else acc)
```

И теперь очевидно, что является функцией, передаваемой в `foldr`:

```
doo xs = foldr (\x z -> if x > 0 then z+1 else z) 0 xs
```

Осталась только последняя функция, сейчас мы ее тоже быстренько расколем:

```
zoo [] acc = acc
zoo (x:xs) acc = zoo xs (x:acc)
```

Тут даже ничего переписывать не надо, сразу видно, что за операция пойдет в `foldr`…

```
zoo xs = foldr (:) [] xs
```

Проверим… `zoo [1,2,3] → [1,2,3]`. Вот тебе раз, а где же `reverse`?

Функция foldl

Здесь очень важно остановиться и подумать, что и где мы делаем не так, где наша логика оторвалась от жизни. Вернемся к нашей функции `zoo` еще раз:

```
zoo [] acc = acc
zoo (x:xs) acc = zoo xs (x:acc)
```

Вот как она работает:

```
zoo [1,2,3] [] → zoo [2,3] [1] → zoo [3] [2,1] → zoo [] [3,2,1] → [3,2,1]
```

А теперь вспомним, что функция foldr как бы расставляет свою операцию между всеми

элементами списка по схеме:

x_1 `f` (x_2 `f` (x_3 `f` … (x_N `f z`)

Ну-ка, подставим на место `f` операцию `(:)`, а на место нулевого элемента – пустой список:

x_1 `:` (x_2 `:` (x_3 `:` … (x_N `: []`)

Теперь очевидно, почему ничего со списком не изменилось, не так ли? И одновременно это означает, что функция `foldr` не может реализовать нашу функцию `zoo`, по крайней мере, так легко, как мы на это надеялись.

А как тогда записать результат работы функции `zoo`, в виде списка элементов, между которыми расставлены какие-то операции?

([] `:` x_1) `:` x_2) `:` x_3) `:` … `:` x_N)…)

Постойте, операция `(:)` ведь требует, чтобы сначала ей дали элемент, а потом только список! Разумеется, вы правы, и нам придется переписать это выражение:

([] `f` x_1) `f` x_2) `f` x_3) `f` … `f` x_N)…),

где

```
f xs x = x:xs
```

Остановитесь и подумайте еще раз. Да, теперь у нас не правая свертка, а **левая**. И функция используется не обычная `(:)`, а как бы перевернутая – это то же `(:)`, но принимающее свои параметры в обратном порядке.

> Кстати, это бывает нужно довольно часто, так что когда-то кому-то надоело плодить такие двойники функций, и он придумал функцию `flip`:
>
> ```
> flip :: (a -> b -> c) -> b -> a -> c
> flip f = \x y -> f y x
> ```
>
> Так что, наша функция `f` – это всего лишь `flip (:)`

И при ближайшем рассмотрении-то, функция `zoo` ведет себя совсем не так, как `foldr`, это видно даже по тому, как используется нейтральный элемент, то бишь, начальное значение. Функция `foldr` проходит рекурсивно до конца списка, и соединяет с нейтральным элементом **последнее** значение.
А функция `zoo` с пустым списком соединяет **первое** значение списка. Ну еще бы они себя вели одинаково – это ведь разные свертки, левая и правая!

А что же функции `hoo` и `doo`? Они ведь были копией функции `zoo`? Давайте посмотрим, какую свертку реализуют они, левую или правую?

```
hoo [] acc = acc
hoo (x:xs) acc | x < acc   = hoo xs x
               | otherwise = hoo xs acc
```

```
doo [] acc = acc
doo (x:xs) acc | x > 0   = doo xs (acc+1)
               | otherwise = doo xs acc
```

Они идут по списку, начиная с первого элемента до последнего, и начинают с того, что первый элемент как-то соединяют с аккумулятором. Да, это действительно **левая** свертка. Почему же у нас получился правильный результат, когда мы использовали `foldr`? Да потому что эти функции таковы, что у них получается одинаковый результат что слева, что справа! И действительно, какая разница, искать минимальный элемент, или подсчитывать количество положительных чисел, справа – или слева? Результат будет один!

Можно даже эти функции переписать так, что становится очевидным, почему `foldr` для них сработал нормально:

```
hoo [x] = x
hoo (x:xs) = min(x, hoo xs)

doo [] = 0
doo (x:xs) = (if x > 0 then 1 else 0) + doo xs
```

Вот теперь это точно **правая** свертка, а функции, очевидно, дадут тот же самый результат.

А что же с нашей **левой** сверткой? Мы ведь так и не написали для нее функции! Давайте сначала приведем три последние функции к общему виду, чтобы стало очевидно, где у них общность:

```
hoo [] acc = acc
hoo (x:xs) acc = hoo xs (if x < acc then x else acc)


doo [] acc = acc
doo (x:xs) acc = doo xs (if x > 0 then acc+1 else acc)

zoo [] acc = acc
zoo (x:xs) acc = zoo xs (x:acc)
```

Что общего у всех этих функций? Они идут по своим спискам, держа в уме текущее значение аккумулятора, и при обработке каждого элемента вызывают себя рекурсивно, заменяя текущее значение аккумулятора. Как заменяя? Что такого общего есть у всех трех этих функциях в тех трех выражениях в правой части, которые как раз и отличаются? А все эти три выражения есть какая-то **функция от текущего элемента и аккумулятора**! И именно эту функцию мы и вынесем в параметры как функцию `f`:

```
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Вот теперь мы можем довершить переписывание наших функций через свертки:

```
hoo xs = foldl min (head xs) xs

doo xs = foldl (\ acc x -> if x > 0 then acc+1 else acc) 0 xs

zoo xs = foldl (flip (:)) [] xs
```

Свертки: разбор полетов

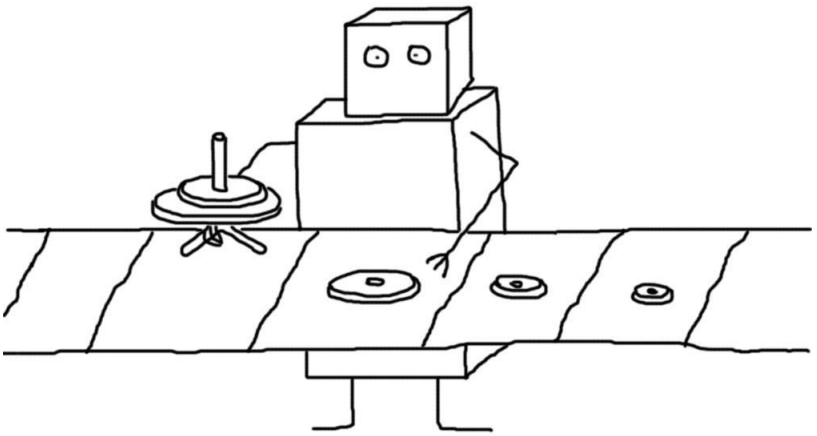
В чем идея функций `foldl` и `foldr`? Сами функции мы написали, а теперь давайте запросим у интерпретатора их тип:

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

Обе функции принимают какую-то бинарную операцию, принимают нулевой элемент, он же есть начальное значение аккумулятора типа `a`, и принимают список значений какого-то типа `b`. Теперь посмотрим на тип операции: эта операция принимает аккумулятор `a`, очередной элемент `b`, и возвратить она должна новое значение аккумулятора `a`. Отличие сигнатуры типов только в том, в каком порядке принимает операция свои аргументы: `(a -> b -> a)` для `foldl` и `(b -> a -> a)` для `foldr`.

В чем идея этих функций? Робот `map` шел вдоль-по списку-конвейеру, преобразовывал каждый элемент и складывал обратно. Робот `filter` шел по списку-конвейеру, проверял каждый элемент и какие-то выкидывал. А какая аналогия здесь?

Такая аналогия на самом деле есть. Робот `foldl` тоже идет по списку, как и робот map, но в отличие от последнего, робот `foldl` имеет **аккумулятор**, то есть память. Он идет вдоль конвейера, держа в руках нечто полусобранное – текущее состояние аккумулятора. Робот берет каждый элемент, и применяет какую-то операцию к этому элементу и своему полусобранному предмету, получая в результате новый полусобранный предмет, с которым в руках идет дальше. В итоге, к концу конвейера в его руках будет нечто, потенциально собранное из всех предметов на конвейере.



> Признаюсь, `foldl` – моя любимая функция. Чудится мне в ней аллегория на все возможные вычисления вообще. Ведь аккумулятор у `foldl` совсем ничем не ограничен – он может быть вообще произвольным. В аккумуляторе у `foldl` может находиться целиком все состояние памяти компьютера, и тогда каждый шаг функции `foldl` – это выполнение какой-то команды, взятой с конвейера, которая как-то изменяет состояние аккумулятора, то есть ей памяти, и передает это состояние на следующий шаг вычислений. Впечатляет, не так ли?

Для иллюстрации того, что может делать функция `foldl` (а делать она может все, или почти все), посчитаем с помощью нее сразу количество больших и маленьких букв в заданной строке. Для

этого аккумулятор, передаваемый с шага на шаг, должен содержать текущее найденное количество больших и маленьких букв. Кортеж для этого пойдет лучше всего:

```
cnt xs =
    foldl (\ (sm,bg) c -> if isLower c then (sm+1,bg) else (sm,bg+1)) (0,0) xs
```

Причем формальный параметр xs можно и опустить, а при большом желании и без лямбды обойтись:

```
cnt = foldl docnt (0,0) where
    docnt (sm,bg) c | isLower c = (sm+1,bg)
                    | otherwise = (sm,bg+1)
```

> В заключение о свертках: попробуйте сами разобраться, могут ли функции `foldr` и `foldl` работать с бесконечными списками? Очень полезное мероприятие. Думаю, не ошибусь, если предскажу, что в процессе поиска правильного ответа на этот вопрос, вы несколько раз поменяете свое мнение :).

Выявление общей функциональности

Давайте оглянемся на найденные функции `map`, `filter`, `foldl` и `foldr` и подумаем, что мы сделали. Мы рассматривали обычные функции, и выявляли в них общее – ту самую схему, идею, по которой строилась обработка данных. В некоторых случаях приходилось преобразовать определение функции для того, чтобы общность идеи стала видна. Мы знали, в каком направлении преобразовывать код именно потому, что прежде увидели это направление разумом.

Идея каждой функции была достаточно общей, чтобы охватить достаточно большое количество конкретных функций, встречающихся в жизни. Достаточно общей – но не более того. Например, функция map у нас преобразовывала каждый элемент списка с помощью общей функции, которой было доступно только значение элемента, - но не его порядковый номер, или значения соседних элементов. Вполне можно было бы написать и более общую функцию, которая бы позволяла обрабатывать элемент, учитывая его порядковый номер:

```
map :: (Integer -> a -> b) -> [a] -> [b]
map f xs = mapN f 1 xs

mapN f _ [] = []
mapN f n (x:xs) = f n x : mapN f (n+1) xs
```

Здесь каждый элемент преобразовывается с помощью функции `f`, которая принимает не только очередной элемент `x`, но и его порядковый номер `n`.

В таком случае, некоторые функции можно было бы записать проще именно с такой версией функции `map`. Помните, была у нас задача - занулить те числа в списке, которые совпадают со своим порядковым номером? Элементарно, Ватсон!

```
zeroSome xs = mapN (\n x -> if x==n then 0 else x) xs
```

Но подождите, ведь того же самого мы добились и с помощью обычной функции `map`, приклеив дополнительно к каждому элементу его номер?

```
zeroSome xs = map f (zip [1..] xs) where f (n,x) = if x==n then 0 else x
```

Кстати, зная теперь, что такое композиция, мы перепишем последнюю функцию еще короче:

```
zeroSome = map f . zip [1..] where f (n,x) = if x==n then 0 else x
```

Что получается? Да, модифицированная версия функции `map` может все то же самое, что и классическая версия функции `map`. Но часто ли требуется такое преобразование? Лучше так: достаточно ли часто требуется такая функциональность, чтобы устать от повторного использования `zip`?

Стандартные функции высших порядков

В стандартный набор библиотек входит много достаточно общих функций высших порядков. Я их приведу сейчас вместе с определениями, но самое главное в этих функциях, конечно – это не их код: он тривиален. Самое главное – идея достаточно общего поведения, которое кто-то заметил и оформил в функцию высших порядков:

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

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

Обычные функции `take` и `drop` брали или, соответственно, выбрасывали из списка заданное количество элементов, а функция `filter` берет или выбрасывает элементы из списка по какому-то условию, применяемому ко всем элементам поочередно. Функции `takeWhile` и `dropWhile` проверяют условие и берут или выкидывают элементы только до первого случая, когда условие **не срабатывает**.

```
takeWhile (not.wtspc) "They are coming. They are here" → "They are coming" where
    wtspc = `elem` ".,!?"
```

Очень часто возникает необходимость применять некоторую функцию несколько раз: к исходному значению, потом к результату первого применения, потом к результату второго применения, и так далее. Это типовое поведение охватывается функцией `iterate`:

```
iterate :: (a -> a) -> a -> [a]
iterate f x        = x : iterate f (f x)
```

Функция `iterate`, как мы видим, создает бесконечный список из исходного значения x, первого результата `f x`, второго результата `f (f x)`, третьего результата `f (f (f x))` и так далее. Согласитесь, бесконечный список в данном случае получать удобнее, чем передавать этой функции какую-то дополнительную информацию о том, сколько именно раз нужно функцию применить. Как эту функцию использовать?

```
take 10 (iterate (*2) 3) → [3,6,12,24,48,96,192,384,768,1536]
take 10 (iterate (drop 1) "hello") → ["hello","ello","llo","lo","o","","","","",""]
```

```
take 5 (iterate (\(x,y) -> if x > y then (x-y,y) else (x,y-x)) (18,24)) →
      [(18,24),(18,6),(12,6),(6,6),(6,0)]
```

Вот еще несколько примеров:

```
any, all  :: (a -> Bool) -> [a] -> Bool
any p      = or  . map p
all p      = and . map p
```

Надеюсь, для вас уже не составляет труда расшифровать любое из этих определений как:

```
any p xs      = or (map p xs)
all p xs      = and (map p xs)
```

Эти две функции проверяют, выполняется ли определенное условие хотя бы на одном (или, соответственно, на всех) элементе списка:

```
all (>0) [1,2,3,-5] → False
all (>0) [1,2,3] → True
```

Обратите внимание, у нас уже одни функции высших порядков (`any`, `all`) выражаются через другие (`map`), и так и должно быть! Всегда, когда вам нужно реализовать какую-то функциональность, постарайтесь увидеть, как эта функциональность реализуется через уже известные вам функции высших порядков, и только потом уже пишите реализацию свою.

```
($) :: (a -> b) -> a -> b
($) f x = f x
```

Функция `($)`, как мы видим, это просто применение функции. Отличается она от обычного применения только приоритетом. Выражение `sin x + 1` будет означать `(sin x) + 1`, а выражение `sin $ x + 1` будет вычисляться как `sin (x + 1)`. Таким образом, пробел можно условно рассматривать как бинарный оператор применения функции к аргументу, имеющий наивысший приоритет; тогда как функция `($)` имеет, наоборот, наименьший.

```
zipWith                  :: (a->b->c) -> [a]->[b]->[c]
zipWith z (a:as) (b:bs)   = z a b : zipWith z as bs
zipWith _ _      _        = []
```

Очень полезная функция `zipWith` позволяет "слить" вместе два списка, применяя к элементам, стоящим на соответствующих местах, заданную функцию. Например, даже обычная функция `zip` на самом деле может быть реализована с помощью `zipWith (\x y -> (x,y))`, а кроме этого:

```
zipWith (+) [1,1,2,3,5,8] [1,2,3,5,8,13] → [2,3,5,8,13,21]

fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

Упс, мы случайно написали функцию, находящую бесконечный список чисел Фибоначчи. Да, с Haskell такое бывает…

```
curry          :: ((a,b) -> c) -> (a -> b -> c)
curry f x y     = f (x,y)

uncurry        :: (a -> b -> c) -> ((a,b) -> c)
uncurry f p     = f (fst p) (snd p)
```

Помните, мы давным-давно говорили о том, что все функции в Haskell принимают ровно один параметр? Помните про каррированные и некаррированные функции? Вот эти две функции позволяют преобразовывать каррированную функцию (принимающую параметры по одному) в некаррированную (принимающую все параметры разом, в виде кортежа) – и наоборот.

```
span, break         :: (a -> Bool) -> [a] -> ([a],[a])
span p []            = ([],[])
span p (x:xs)
       | p x       = (x:ys, zs)
       | otherwise = ([], x:xs)
                       where (ys,zs) = span p xs
break p              = span (not . p)
```

Функции `span` и `break` выполняют сразу работу и `takeWhile` и `dropWhile` вместе взятых. Они принимают условие `p` и список `xs`, разделяют `xs` на две части: в первую часть попадает значение `takeWhile p xs`, а во вторую – `dropWhile p xs`.

> Во второй части этого материала, в задачнике, в главе про списочные функции высших порядков приведены примеры других функций, реально имеющихся в стандартной поставке, и потенциально возможных для реализации. Очень рекомендую со всеми ними ознакомиться, потому что не будет особым преувеличением сказать, что сила функционального программирования заключается именно в широком применении функций высших порядков, совместно с быстрой и удобной подготовкой рабочих функций для них с помощью частичного применения и лямбда-функций.

Еще немного про строгие функции

Возможно, этот раздел покажется вам немножко сложноватым. Давайте договоримся так: как только вы теряете нить объяснений - сразу бросайте это безнадежное дело, и переходите к следующему разделу. А сюда вернетесь тогда, когда ваши функции вдруг оказываются ужасно неэффективными (по времени или памяти) тогда, когда с этому, вроде бы, никаких предпосылок нет. Договорились?

Помните, мы говорили, что все функции в Haskell - нестрогие и ленивые? Вот у функции `($)` есть специальный аналог `($!)`:

```
($) :: (a -> b) -> a -> b

($!) :: (a -> b) -> a -> b
```

Функция `($)`, как мы помним - это просто применение функции. Так вот, функция `($!)` - это тоже просто применение функции, но - строгое. То есть, если функция `($) f x` просто скажет что-то типа "эй, функция `f`, вот тебе `x`, дай-ка мне твой результат?", то функция `($!) f x` сначала вычислит `x`, а потом уже передаст его функции `f`.

> Тут можно было бы нарисовать смешной диалог. Если спросить у обычной функции `(+)`: "Чему будет равно `(+) 5 6` ?", - ленивая функция ответит: "`5+6`". А вот если был бы строгий аналог функции `(+)`, он бы ответил честно: `11`.

Как бы эту разницу увидеть? А вот, например, так:

```
SomeModule> length (take 10 (repeat $ (1/0)))
10

SomeModule> length (take 10 (repeat $! (1/0)))
Program error: {primDivDouble 1.0 0.0}
```

Что происходит в первом случае? Функция `repeat` создает целый список из нетерминальных выражений `1/0`, мы потом пытаемся выбрать `10` элементов, и считаем, сколько выбралось. Как и положено ленивым функциям, результат вычисляется нормально, потому что `repeat` и не думает вычислять `1/0` - оно ей надо?
Зато во втором случае, оператор строгого применения `($!)` вычисляет `1/0` перед тем, как передать его функции `repeat`, и, разумеется, вычисление обрывается с ошибкой.

Однако, при этом:

```
head [2, (1/) $! 0] → 2.0
length $! [map sin [1..]] → 1
```

Оставлю вас самих разбираться, почему в этом случае вроде бы нетерминальные конструкции так и не вычисляются (в смысле - не приводят к ошибке в вычислении основного выражения).

Как правило, мы не будем сталкиваться с необходимостью делать строгие функции вместо нестрогих. Но вы можете столкнуться с тем, что при вычислении очень глубокой рекурсии (даже и хвостовой), накапливаемый параметр разбухает так, что не умещается в стеке и программа падает - или не падает, но просто очень неэффективно работает. Попробуйте, например, вычислить выражение `foldl (+) 0 [1..100000]`? Казалось бы, чего там - просто посчитать сумму первых ста тысяч натуральных чисел - а ваш простой интерпретатор может и не справиться. Почему, ведь функция `foldl` даже реализована с использованием хвостовой рекурсии!

```
foldl             :: (a -> b -> a) -> a -> [b] -> a
foldl f z []       = z
foldl f z (x:xs)   = foldl f (f z x) xs
```

Проблема в том, что наш ленивый язык накапливает в аккумуляторе не частичные суммы (`1,3,6,10,...`) а конструкцию вида `(((0+1)+2)+3)+...` , в надежде, что эту сумму не придется вычислять. Но мы-то знаем, что придется! Для этого есть строгая версия функции `foldl`, которая отличается от нее только апострофом в названии (апостроф сам по себе к строгости отношения не имеет - он считается просто обычной буквой):

```
foldl'            :: (a -> b -> a) -> a -> [b] -> a
foldl' f a []      = a
foldl' f a (x:xs) = (foldl' f $! f a x) xs
```

Для сравнения, перепишем немного определение функции `foldl`, чтобы очевидна стала разница (вы же видите, что это то же самое определение, что и абзацем назад?):

```
foldl             :: (a -> b -> a) -> a -> [b] -> a
foldl f z []       = z
foldl f z (x:xs)   = (foldl f $ (f z x)) xs
```

На каждом шаге функция `foldl'` действительно вычисляет новое значение аккумулятора, а не накапливает там ленивую очередь из вызовов! Проверьте, выражение `foldl' (+) 0 [1..100000]` вычисляется легко (чтобы воспользоваться функций `foldl'`, придется, возможно, подключить

модуль `Data.List`).

Создание своих типов данных

Любой язык программирования позволяет создавать свои собственные структуры данных для решения конкретной прикладной задачи, которая, конечно же, оперирует чем-то большим, нежели отдельные числа, строки или даже кортежи.
Создание своего типа данных позволяет инкапсулировать в одном месте сущность, описывающую объекты, с которыми вам нужно работать, и описать функции для работы именно с этими объектами.

Для создания новых типов в Haskell используется ключевое слово `data`.

Простые перечислимые типы данных

Простые перечислимые типы данных легко создать с помощью ключевого слова `data` просто путем перечисления всех значений, которые этому типу принадлежат:

```
data Bool  = False | True
```

Мы взяли и объявили, что тип `Bool` состоит из двух значений: `True` и `False`. И действительно, уже после такого объявления можно запросить тип этих значений и убедиться:

```
True :: Bool
False :: Bool
```

Отлично, новый тип есть. Значения этого типа мы создавать можем – просто написав `True` или `False`, а других значений у этого типа не бывает. Как же писать теперь функции, которые с этим типом работают?
Обычно, чтобы написать функцию, нужно (просто по определению математическому) предоставить правило, по которому каждому элементу из области определения сопоставляется одно и только одно значение из области значений. Фу, выговорил.
Короче говоря: хотите, чтобы функция работала с значениями определенного типа – опишите, как она должна работать со всеми значениями этого типа.

Вот, например, придумали мы функцию `not :: Bool -> Bool`. Ну так и опишем, как она должна себя вести, если ей передают то или иное значение типа `Bool`:

```
not False = True
not True = False
```

Контейнеры

Хотя перечислимые типы данных тоже и полезны, но в большинстве случаев создаваемые типы данных являются сложными, потому что содержат внутри себя значения других типов, являясь в таком случае в некоем роде контейнерами. Вот, посмотрите на новый тип данных – точку в двумерном целочисленном пространстве:

```
data Point = Pt Integer Integer
```

В таком объявлении типа, простом на вид, уже присутствует множество новых для нас понятий. Что такое `Point`? Это наш новый тип данных. Что такое `Pt`? `Pt` – это нечто, называемое

конструктором данных. Давайте запросим у интерпретатора тип этого `Pt`, и тогда сразу поймем, в чем дело:

```
Pt :: Integer -> Integer -> Point
```

Вот оно что! `Pt` – это, оказывается, функция, которая берет два значения типа `Integer` и возвращает значение типа `Point`?! Так - да не совсем так. Не зря `Pt` пишется с большой буквы, тогда как все функции раньше мы писали строго с маленькой. `Pt` – это именно конструктор данных (в данном случае – бинарный), то есть нечто, **конструирующее** новые данные из уже существующих.

В чем разница? Значение сложного типа, а таковым является тип `Point`, всегда можно разобрать обратно на составляющие. То есть передали мы "функции" `Pt` два целых числа – она вернула там значение типа `Point`. Наигрались мы значением типа `Point` – мы всегда можем разобрать его обратно на те же самые два целых числа.
Ну-ка, попробуйте обратно разобрать на составляющие выражение `(+) 2 3`? Что у вас получится, `2` и `3`? Или `1` и `4`? Функция – это в общем случае разрушающее преобразование, а конструктор данных – никогда не разрушающее.

А в случае с перечислимыми типами данных появляются где-нибудь конструкторы данных? Конечно появляются – просто по аналогии можно считать, что если `Pt` – это бинарный конструктор данных типа `Point`, то `True` или `False` – это нуль-арные конструкторы данных типа `Bool`. Есть контакт?

Итак, объявить-то мы тип `Point` объявили, а как нам теперь создать значения этого типа?

```
Pt 5 4 :: Point
Pt (-1) 3 :: Point
```

А как нам работать с значениями типа `Point`? Как обратно из значения типа `Point` получить его первую или вторую координаты? Надо написать функцию типа `Point -> Integer`:

```
px :: Point -> Integer
px (Pt x y) = x
```

Смотрите, смотрите, что происходит! Мы наш конструктор данных использовали не по назначению! Не для создания нового значения типа `Point`,… а для разбора существующего значения типа `Point` на составляющие!

Подождите, а когда мы писали функции вида

```
head (x:xs) = x,
```

разве не то же самое мы делали? Операция `(:)` умела создавать новый список из хвоста и головы, и ее можно было использовать в образцах… Вот и получается, что наш конструктор данных `Pt` тоже можно использовать в образцах!

Помните, когда мы говорили про образцы, обсуждали, что может быть образцом, а что не может быть? У нас был четвертый пункт, который гласил:

4. Ну и наконец, самый главный пункт: одновременно и самый важный, и самый сложный для

логичного объяснения. Давайте я пока расплывчато сформулирую это так: образцом может быть **конструкция**, единственным образом раскладывающая переданное в функцию значение на составные части.

Вот теперь мы, наконец, можем сформулировать точнее: образцом может быть конструктор данных. Ура, непротиворечивость логики восстановлена.

Вернемся еще раз к функции `px`:

```
px :: Point -> Integer
px (Pt x y) = x
```

Прочитаем мы это определение так: "`px` – это функция, принимающая точку, и возвращающая целое число. Если в эту функцию передадут точку, состоящую из координат `x` и `y`, обернутых в конструктор данных `Pt`, то возвращай `x`".

Для второй координаты тоже придется написать функцию `py`:

```
py :: Point -> Integer
py (Pt x y) = y
```

А если бы значений внутри нашего типа было больше? Пришлось бы для каждого поля писать отдельную функцию? Пришлось бы. Или можно было воспользоваться механизмом именованных полей, тогда тип `Point` объявлялся бы так:

```
data Point = Pt {px :: Integer, py :: Integer}
```

Тогда система языка автоматически сгенерирует функции для доступа к полям:

```
px :: Point -> Integer
py :: Point -> Integer
```

Давайте, наконец, напишем функцию, находящую квадрат расстояния между двумя точками:

```
sqdist :: Point -> Point -> Integer
sqdist (Pt x1 y1) (Pt x2 y2) = ((x1-x2)^2 + (y1-y2)^2)
```

О чем говорит эта функция? Если ее вызвали и передали ей две точки, одну с координатами `x1` и `y1`, а другую с координатами `x2` и `y2`, то возвращай значение такого-то выражения.

О сравнении, отображении и прочих стандартных операциях

Мы создали тип данных, Point, однако первая же попытка распечатать в интерпретаторе значение этого типа приведет к ошибке:

```
SomeModule> Pt 1 2
ERROR - Cannot find "show" function for:
*** Expression : Pt 1 2
*** Of type    : Point
```

Помните про функцию `show` и класс `Show`? Претензии интерпретатора понятны: он не знает, как своей функцией `show` отобразить наш созданный тип данных. Похожая проблема возникнет, если мы попробуем сравнить два значения нашего типа `Point`. Позже я покажу, как эту проблему

решать в общем случае, а пока воспользуемся магией:

```
data Point = Pt {px :: Integer, py :: Integer} deriving (Eq, Show)
```

Вот этот маленький довесок нам очень облегчит жизнь. Он позволит сравнивать значения типа `Point` между собой и видеть значения этого типа в интерпретаторе. Смысл этой магии в том, что мы просим интерпретатор как-нибудь самостоятельно придумать, как сравнивать и отображать значения этого типа:

```
SomeModule> Pt 1 2
Pt{px=1,py=2}
```

Параметрические типы данных

Рассмотренный тип данных, `Point`, представляет собой точку с целочисленными координатами. А если мы захотим нецелочисленные координаты – создавать новый тип и переписывать все функции? Для того, чтобы этого избежать, можно сразу создать параметризованные типы данных:

```
data Point a = Pt a a deriving (Eq, Show)
```

Как видим, вместо конкретного типа `Integer`, мы использовали какое-то `a`, которое в данном случае будет называться – переменная типа. По сути, под `a` может скрываться любой тип – и `Float`, и `Integer`, и даже `Char`. Что такое здесь `Pt`? Это бинарный конструктор данных: он берет кусочки данных и создает из них сложные данные. Что такое `Point` теперь? Это унарный конструктор типа: он берет какой-то тип `a` и создает на его основе сложный тип `Point a`:

```
Pt :: a -> a -> Point a
Pt 1 2 :: Point Integer
Pt 1.0 2.0 :: Point Float
Pt 'a' 'b' :: Point Char
```

Кстати, вместо `Pt` можно использовать то же самое слово `Point`. Это разрешено, потому что имена конструкторов типов и конструкторов данных находятся в разных пространствах имен. Короче говоря, интерпретатор всегда поймет, о чем идет речь – о конструкторе типа или о конструкторе данных, поэтому они могут иметь одинаковые имена.

Кстати, а что теперь с нашей функцией квадрата расстояния:

```
sqdist :: Num a => Point a -> Point a -> a
sqdist (Pt x1 y1) (Pt x2 y2) = ((x1-x2)^2 + (y1-y2)^2)
```

Вот какой у нее теперь оказался тип: она будет работать только с точками, у которых координаты имеют численный тип. Что логично, раз функция эти координаты вычитает и возводит в квадрат.

Сложные типы данных

Теперь сведя в кучу все рассмотренные возможности, мы создадим новый тип данных – фигура, которая может быть точкой, а может быть и кругом:

```
data Shape =
    Point {x :: Float, y :: Float} | Circle {x :: Float, y :: Float, r :: Float}
        deriving (Eq, Show)
```

О чем говорит эта запись? О том, что значение нового типа данных `Shape` может быть или точкой, и тогда внутри нее будут храниться только координаты, или кругом, и тогда внутри будут храниться координаты и радиус:

```
Point 1 1 :: Shape
Circle 1 1 2 :: Shape
```

Раз мы задали имена полям, то автоматически создано несколько функций для доступа к ним. Причем для всех полей – хотя, конечно, вызов функции `r` от точки приведет к ошибке.

```
x :: Shape -> Float
y :: Shape -> Float
r :: Shape -> Float
```

В большинстве случаев эти функции нам не понадобятся, так как работать с объектами мы будем с помощью образцов. Давайте, например, напишем функцию, которая будет находить площадь любого объекта типа `Shape`. Вспомним, как мы говорили раньше: чтобы описать функцию над определенным типом, нужно описать, как эта функция работает со всеми возможными значениями этого типа.
Сейчас у нас возможных значений (возможных точек и кругов) бесконечное число, но все они – или точки, или круги. Поэтому нам необходимо и достаточно описать, как требуемая функция будет работать в случае каждого из двух возможных способов организации значений `Shape`. То бишь, проще говоря – как вычислять площадь точек, а как площадь кругов:

```
area :: Shape -> Float
area (Point _ _) = 0
area (Circle _ _ r) = 3.14 * r^2
```

Прочитаем это определение функции мы так: требуется вычислить площадь данной фигуры. Если это точка с любыми координатами, то это строго ноль. Если это круг с любыми координатами и радиусом `r`, то площадь его равна значению вот этого выражения.

Тип данных Maybe

Это мой любимый, веселый, самокритичный тип данных, вот как он определяется:

```
data Maybe a = Nothing | Just a
             deriving (Eq, Ord, Read, Show)
```

Где взять значения этого типа? А вот их примеры:

```
Nothing    :: Maybe a
Just True  :: Maybe Bool
Just 1     :: Num a => Maybe a
Just "ice" :: Maybe [Char]
```

Зачем может быть нужен такой тип? А давайте представим себе функцию, которая ищет значение в ассоциативном массиве по ключу. Пусть есть список кортежей `[(a,b)]`, причем тип `a` допускает сравнение на равенство. И пусть у нас есть какое-то значение типа `a`. Мы хотим написать функцию, которая будет искать в списке кортежей пару `(a,b)`, у которой значение `a` совпадает с искомым, и возвращает значение `b` из этой пары. Какой должен быть тип у этой функции? Может быть, нужно возвращать просто значение типа `b`?

```
lookup :: Eq a => a -> [(a,b)] -> b
```

Но ведь поиск в массиве может оказаться и неудачным, что возвращать тогда? Может быть, нужно возвращать список значений типа `b`? Тогда, если функция ничего не нашла, она сможет вернуть пустой список:

```
lookup :: Eq a => a -> [(a,b)] -> [b]
```

Но в этом случае тип этой функции может ввести кого-нибудь в заблуждение: можно будет подумать, что функция вполне может вернуть несколько значений, а не только одно или ноль.

Как раз в таких ситуациях и пригодится тип `Maybe`. Он как бы дополняет нашу функцию контекстом – дополнительной информацией о том, что функция может и ничего не найти:

```
lookup:: Eq a => a -> [(a,b)] -> Maybe b
lookup key []          = Nothing
lookup key ((k,v):dict)
      | key == k      = Just v
      | otherwise     = lookup key dict
```

Вот оно: из типа этой функции сразу становится ясно: функция берет значение ключа и словарь, и **может быть** возвратит значение из словаря, сопоставленное с ключом. Давайте проверим:

```
lookup 2 [(1,"one"),(2,"two"),(3,"three")] → Just "two"
lookup 4 [(1,"one"),(2,"two"),(3,"three")] → Nothing
```

А как работать с значениями `Maybe String`, ведь именно такие значения вернула нам функция? Можно, например, взять и склеить эту строку с другой?

```
SomeModule> Just "two" ++ " is a lot"
ERROR - Type error in application
*** Expression     : Just "two" ++ " is a lot"
*** Term           : Just "two"
*** Type           : Maybe [Char]
*** Does not match : [Char]
```

В чем тут проблема? Да в том, что функция (`++ " is a lot"`) ожидает, что ей передадут значение типа `String`, а ей передают `Maybe String`. А это, как говорится – две большие разницы. Значение `Maybe String` нельзя просто так сливать с другой строкой, ведь `Maybe String` может и пустым, `Nothing`.
Другими словами, тип возвращаемого функцией `lookup` значения "заражен" контекстом возможной неудачи `lookup`, контекстом возможной пустоты. Мы, конечно, можем преобразовать значение этого типа в обычную строку – но мы тогда должны взять на себя ответственность за то, как в обычную строку должно преобразовываться значение `Nothing`:

```
maybeToString :: Maybe String -> String
maybeToString Nothing = ""
maybeToString (Just s) = s
```

Вот теперь можно попробовать и сложить:

```
maybeToString (Just "two") ++ " is a lot" → "two is a lot"
maybeToString (lookup 3 [(1,"one"),(2,"two")]) ++ " is a lot" → " is a lot"
```

Рекурсивные типы данных: списки

Самые впечатляющие возможности той системы создания своих типов, что есть в Haskell, проявляются при создании рекурсивных и потенциально бесконечных структур данных. Начнем со списка:

```
data List a = Empty | Cons a (List a) deriving Show
```

О чем говорит эта запись? Список значений типа `a` - это или пустой список, или пара из значения типа `a` и списка типа `a`, созданная конструктором данных `Cons`. Давайте еще раз: любой список, согласно этому определению - это или пустой список, или пара (`значение`, `список`). Вам это ничего не напоминает?

Что такое `Empty`? Что такое `Cons`? Это все конструкторы данных: с их помощью можно создать список:

```
Empty :: List a
Cons :: a -> List a -> List a
```

`Empty` возвращает пустой список, а `Cons` принимает значение типа `a`, потом список типа `a`, и возвращает новый список типа `a`. Например, давайте создадим небольшие списки, начиная с пустого. Таким образом, удлиняясь и удлиняясь, список может стать потенциально бесконечным:

```
Empty :: List a
Cons 'a' Empty :: List Char
Cons 'a' (Cons 'b' Empty) :: List Char
Cons 'a' (Cons 'b' (Cons 'c' Empty)) :: List Char
```

Давайте напишем функцию, которая находит длину списка. Как и любая функция, которая работает со сложным типом данных, эта функция должна уметь обрабатывать все возможные значения типа `List a`:

```
len :: List a -> Int
len Empty = 0
len (Cons x xs) = 1 + len xs
```

В этой функции показано, как функция должна работать, если `List a` является значением `Empty`, и если `List a` является `Cons` от значения и другого списка. Вам ничего эта функция не напоминает? Давайте-ка вспомним стандартную функцию `length`, которая работает на стандартных списках:

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

Стопроцентное повторение ведь! Возьмем опять наше определение `List a` и немного его преобразуем: заменим `Empty` на `[]`, `Cons` на `(:)`, ну и `List a` на `[a]`, и получится:

```
data List a = Empty | Cons a (List a)

data [] a = [] | (:) a []

data [a] = [] | a : [a]
```

Последнее, впрочем, уже чистый синтаксический сахар: такое (запись вида `[a]`) позволено только обычным стандартным спискам. Но идея ясна - обычные списки объявляются именно так, как мы записали. Обычный список значений типа `a` - это или пустой список `[]`, или запись вида `x : xs`, где `x` - это значение этого типа `a`, а `xs` - это опять список значений типа `a`.
Думаю, теперь должно быть абсолютно понятно, почему нельзя напрямую обратиться к последнему элементу списка, а только к хвосту: потому что список - это, по сути, пара (голова, хвост), и обращаться можно только к одному из этих элементов.

Рекурсивные типы данных: деревья

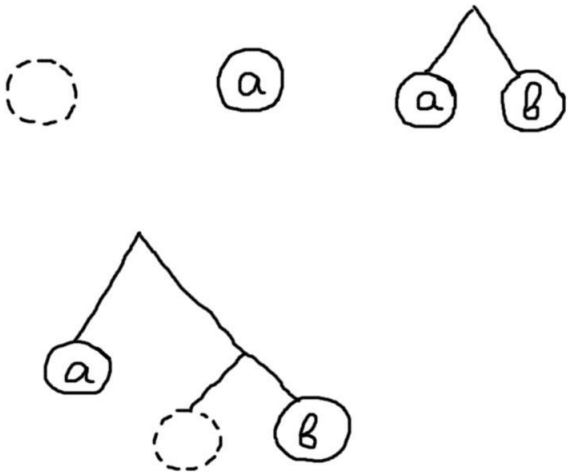
Аналогичным образом создаются структуры данных для деревьев:

```
data Tree a = Empty | Leaf a | Branches (Tree a) (Tree a) deriving Show
```

Что такое дерево типа `a`, согласно этому определению? Во-первых, деревом может быть пустое дерево. Во-вторых, деревом может быть лист со значением типа `a`. И третий вариант - деревом может быть разветвление на две ветви, каждая из которых может быть опять произвольным деревом.

Вот примеры деревьев:

```
Empty :: Tree a
Leaf 'a' :: Tree Char
Branches (Leaf 'a') (Leaf 'b') :: Tree Char
Branches (Leaf 'a') (Branches Empty (Leaf 'b')) :: Tree Char
```



Различных типов деревьев может быть очень много, какого типа дерево создали мы такой структурой данных? Во-первых, наше дерево бинарное: любое ветвление - двоичное. Во-вторых, наше дерево может быть пустым. В-третьих, наше дерево хранит значения только в листах. Видите, как эти три утверждения следуют из определения нашего типа данных? Если вдруг понадобится использовать дерево с произвольным количеством потомков у узла, или если понадобится хранить значения в узлах - придется делать другой тип данных.

Как работать с таким деревом? Точно так же, как мы работали с определяемыми самостоятельно списками, или любыми другими собственными типами данных. Давайте, например, напишем

функцию, которая собирает в список в произвольном порядке все листья с заданного дерева. Функция должна обработать все три случая, все три формы существования дерева:

```
fringe :: Tree a -> [a]
fringe Empty          = []
fringe (Leaf x)       = [x]
fringe (Branches lt rt) = fringe lt ++ fringe rt
```

Если дерево пустое, функция возвращает пустой список. Если дерево состоит из одного листа, функция возвращает список с единственным значением. И наконец, если дерево - это две ветви, то возвращается список из листьев с левого дерева, к которым добавлены листья с правого дерева. По сути, это LDF обход дерева - левый обход в глубину.

В качестве второго примера, напишем функцию, которая применяет к каждому элементу дерева какую-то функцию, сохраняя структуру дерева нетронутой:

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f Empty = Empty
mapTree f (Leaf x) = Leaf (f x)
mapTree f (Branches lt rt) = Branches (mapTree f lt) (mapTree f rt)
```

Функция преобразует лист, если он ей встретился, в такой же лист - но с измененным значением. И проталкивает изменения рекурсивно вниз по веткам, если ей встретились ветки.

Кстати, обратите внимание на тип функции `mapTree`, и сравните его с типом функции `map`:

```
mapTree :: (a -> b) -> Tree a -> Tree b
map     :: (a -> b) -> [a]    -> [b]
```

Не правда ли, у этих функций слишком много общего? Но это уже - совсем другая история.

Ввод-вывод

Любая программа будет производить действия ввода-вывода (в самом широком смысле – от вывода чего-нибудь на экран до чтения и записи в порты устройств), если мы вообще хотим получить от нее какой-нибудь полезный выхлоп, кроме загрузки центрального процессора и обогревания комнаты.
Однако даже простая печать на экран в Haskell сопряжена с определенного рода идейными проблемами. Мы ведь говорили о том, что все функции в Haskell не могут иметь побочные эффекты, а что такое ввод-вывод, как не побочные эффекты?

Итак, с одной стороны, функции ничего не могут печатать на экран и читать с клавиатуры, - а с другой стороны, они должны это делать, иначе как программой пользоваться? Для разрешения этой проблемы был использован довольно серьезный механизм – монады. Это такая абстрактная концепция, которая очень много чего может моделировать, и вот выяснилось, что ввод-вывод тоже отлично в монады укладывается.

Нет, чудес не будет, конечно – функциям придется разрешить производить побочные эффекты. Однако система ввода-вывода в Haskell позволяет явно разделить те части программы, где происходит ввод-вывод, где всякие разные плохие побочные эффекты – а где все тот же чистый строгий декларативный код без побочных эффектов.

С первого взгляда (а честно говоря, и со второго, и с третьего тоже), монады – это ужас, летящий на крыльях ночи. Монадами пугают взрослые программисты своих маленьких программистиков. Однако хорошая новость для вас заключается в том, что, как гласит популярная аналогия, понимание монад требуется для работы с вводом-выводом не больше, чем понимание теории групп для использования простой арифметики. И это действительно так. Мы начнем с того, что напишем простейшую программу, производящую ввод-вывод, и вы убедитесь, что это очень просто. А потом попробуем заглянуть немножко за занавес, чтобы увидеть всю (ну, как минимум, часть) кухни, которая там творится.

В монадах (как и в любой другой абстрактной концепции, впрочем) разобраться можно двумя путями. Можно просто принять абстрактные определения, пропустить их через себя, не пытаясь вдумываться в их жизненный смысл – и потом уже пытаться смотреть частные случаи, проявления этих абстракций. А можно идти с другой стороны – сначала разобраться в простом частном применении, потом в другом применении, потом в третьем – а потом увидеть во всех них общую абстрактную часть, и понять ее смысл через примеры ее частных проявлений.

Я надеюсь, что мои простые объяснения монады ввода-вывода потом помогут вам понять, что такое монады в целом, - я за этот путь.

Простейший ввод-вывод

Итак, вот простейшая программа, которая читает входной текстовый файл, приводит все буквы к верхнему регистру, а потом пишет выходной файл:

```
main :: IO ()
main = do
    contents <- readFile "input.txt"
    writeFile "output.txt" (process contents)

process :: String -> String
process s = map toUpper s
```

Вот, собственно, и все. Видите тут две части программы? Вторая часть для вас должна быть вполне понятна – тут обычная функция `process`, преобразующая строку в строку. Это действительно обычная функция, никаких побочных эффектов у нее нет и быть не может.

А вот функция `main` – это как раз та самая часть программы, что отвечает за ввод-вывод, это специфическая функция. В ней мы связываем и именем `contents` все содержимое файла `input.txt`, потом преобразовываем это содержимое с помощью функции `process`, и записываем результат в файл `output.txt`.

Между двумя этими функциями проходит незримая граница, отделяющая чистый функциональный мир строгих функций без всяких побочных эффектов - от грязного императивного чистилища.

Ну вот, а вы боялись. Усложняйте функцию `process`, и делайте преобразование файла любой сложности. Проблемы у вас возникнут только тогда, когда потребуется смешивать в лапшу чистые вычисления и взаимодействие с пользователем: если потребуется что-то запросить у пользователя (или прочитать из файла), потом что-то посчитать, потом опять что-то запросить, и так далее. В этом случае – ничего не поделаешь, придется разбираться с кухней.

Объяснение кухни

Итак, попробуем разобраться, как же все-таки производится ввод-вывод, и что означает этот новый синтаксис, который мы увидели в функции `main`, со словом `do`, которое подозрительно пахнет банальной императивщиной.

Итак, начнем с того, что разберемся в новом понятии. Представим себе функцию, которая должна прочитать с клавиатуры символ и вернуть его. Какой тип у этой функции вы ожидаете?

```
getChar :: Char
```

Но разве такое возможно? Ведь это не функция, это, простите, константа! Сравните:

```
getChar :: Char
'a' :: Char
```

Но мы интуитивно понимаем, что `getChar` и `'a'` имеют явно разный тип. И это действительно так, потому что `getChar` – это не просто функция, возвращающая `Char`, это **действие** ввода-вывода. Вот он, неизвестный доселе науке зверь – **действие**. Есть просто `Char`, а есть действие, которое **может быть выполнено**, и тогда в результате из действия можно **извлечь** `Char`. Вот как этот факт отражается в типе функции `getChar`:

```
getChar :: IO Char
```

Видите? Бывает `Tree Char` – это не символ, это дерево символов. Бывает `[Char]` – это список символов. Бывает `Maybe Char` – это тоже не символ, это "может быть, символ, а может и ничего". Ну и вот `IO Char` – это тоже не символ, это действие ввода-вывода, которое можно выполнить и потом извлечь из него символ. Если действие выполняется, происходит общение программы с ее окружением: чтение портов клавиатуры и прочих устройств, правка видеопамяти, и так далее. Но даже по окончании выполнения действия у нас не получается `Char`. Нельзя просто написать что-то типа `(execute getChar) : "ello world!"` – символ надо еще извлечь.

В книжке с синим слоном приводится следующая аналогия, может она вам поможет: действие – это ящик на ножках. Мы можем отправить его в реальный мир, он туда сбегает и принесет в ящике `Char`.

Вот еще пример действия:

```
putChar :: Char -> IO ()
```

На самом деле это не действие, это функция, которая принимает самый обычный `Char` и возвращает действие. Например:

```
putChar 'H' :: IO ()
```

Вот это уже действие. Если его выполнить, то произойдет печать на экране буквы `'H'`, а в результате действия ничего не будет возвращено. Если уж продолжать аналогию Мирана Липовачи, то `putChar 'H'` – это ящик на ножках, который можно отправить в реальный мир, и он там распечатает на экране букву `'H'` и вернется назад, не принеся ничего.

Вот еще примеры функций, возвращающих действия:

```
getLine :: IO String
```

```
putStr :: String -> IO ()
readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
```

Как видим, действия умеют возвращать значения различных типов: `IO String` – это действие, из которого после выполнения можно извлечь `String`, `IO Integer` - это действие, из которого после выполнения можно извлечь `Integer`, а `IO ()` – это действие, которое можно только выполнить, но ничего извлечь после этого из него нельзя.

Важно понимать, что **действие** – это одно, а **выполнение действия** – это совсем другое. Например, несколько действий можно сложить в список:

```
[putChar 'a', putChar 'b', putChar 'c'] :: [IO ()]
```

Но тип этого выражения – список элементарных (простейших) действий. А списки нельзя выполнять, - выполнять можно только действия. От того, что вы создали и положили в список три (чуть было не написал – последовательных; ну вот, написал) действия, они не будут выполнены.

Итак, бывают элементарные действия, и их можно выполнять. А что же делать, если требуется последовательно выполнить несколько действий? Нужно несколько действий превратить в одно действие!

Для этого есть страшная и ужасная операция, приготовьтесь:

```
(>>) :: IO a -> IO b -> IO b
```

Что это за зверь? Это функция, как следует из ее типа, которая принимает одно действие `IO a`, потом принимает еще одно действие `IO b`, и возвращает действие `IO b`. Пример:

```
putChar 'h' >> putStr "ello world!" :: IO String
```

Тип функции теперь, я думаю, понятен. А о том, что эта функция делает, я думаю, вы догадались. Нет, она не выполняет два действия последовательно. Она создает новое, сложное действие так, что **если** попытаться выполнить его, то сначала выполнится первое действие, а потом второе действие. Чувствуете разницу? Ленивость на марше!

Да, если вы в интерпретаторе напишете `putChar 'h' >> putStr "ello world!"`, то будет вычислен результат этого выражения – это будет составное действие, и раз вы его ввели в интерпретаторе, то интерпретатор и скомандует ему выполниться. Вот тут в действие вступит ленивая функция `(>>)`, которая вспомнит, что ей передали сначала одно простое действие, а потом другое, и выполнит их последовательно, и вы увидите на экране то, что и должны увидеть.

Отлично, комбинировать последовательные действия мы научились. Теперь давайте рассмотрим другой пример:

```
getChar >> getChar :: IO Char
```

Это опять сложное действие, при выполнении которого у пользователя будет запрошен с клавиатуры символ (первое действие), потом будет запрошен второй символ (второе действие), и все это будет одним большим действием, из которого потом можно будет извлечь `Char`. Но какой, первый или второй?
Тип операции `(>>)` дает нам подсказку: да, типы соединяемых ей действий могут быть абсолютно

разными (`IO a`, `IO b`), но типом всего составного действия является `IO b`. Это означает, что результат первого действия – теряется.

Но это ведь печально! Представьте, что нам нужно какую-то информацию передать из действия в действие: например, считать с клавиатуры символ, привести его к верхнему регистру и потом вывести его на экран. Просто так написать не получится:

```
SomeModule> toUpper getChar
ERROR - Type error in application
*** Expression     : toUpper getChar
*** Term           : getChar
*** Type           : IO Char
*** Does not match : Char
```

С подобной ошибкой вы будете сталкиваться часто, пока не привыкнете. Суть претензий интерпретатора заключается в том, что функции `toUpper` требуется `Char`, а мы пытаемся подсунуть ей `IO Char`, а разница между ними принципиальна.

Обдумайте эту проблему, пожалуйста, еще раз: она очень важна. Если у вас есть действие, например, `IO String`, то вы не сможете просто так взять и превратить его в обычное значение `String`, это невозможно. Если у вас есть функция:

```
foo :: String -> String,
```

вы не можете внутри нее производить ввод-вывод:

```
foo :: String -> String
foo s = s ++ getLine
```

Тип функции foo исключает возможность какого-либо ввода-вывода внутри нее. Это очень важно, потому что именно это и является границей между чистым функциональным декларативным миром – и миром императивным. Если у вас есть функция `bar :: String -> IO String`, то вы можете от нее ждать всего что угодно – вплоть до форматирования жесткого диска, с возвратом строки `"hahaha"`. Но если у вас есть функция `foo :: String -> String`, то мы можете быть уверены: эта функция никакого ввода-вывода не производит, побочных эффектов не имеет, будучи вызвана с одними и теми же параметрами, она всегда возвращает один и тот же результат, - короче говоря, получать все ништяки чистого функционального программирования.

Да, вы можете сделать то, что хотите – и сейчас покажу, как. Но тип функции `foo` будет вынужден отразить тот факт, что она производит ввод-вывод:

```
foo :: String -> IO String
foo s = s `как-то-сложить` getLine
```

Итак, суть проблемы: как-то воспользоваться значением, извлеченным из одного действия, - во втором действии. Для этого есть еще более страшная функция:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

Кошмар! Неужели в этом вообще можно разобраться? Эта операция принимает сначала одно действие `IO a`, потом принимает функцию, принимающую `a` и возвращающую действие `IO b`, и возвращает действие `IO b`. Чтобы понять, что можно передавать в эту функцию, и что она будет

делать, надо посмотреть, какие у нас есть подходящие по типу действия и функции. Например:

```
getLine :: IO String
putStr :: String -> IO ()
```

Подождите, так это как раз и есть два значения, которые подходят по типу, и которые можно передать операции `(>>=)`!

```
getLine >>= putStr :: IO ()
```

Это выражение (оно правильно типизировано, мы проверяли) представляет из себя действие, состоящее из двух простых действий. При его выполнении операция `(>>=)` сначала выполнит первое действие, извлечет его результат – и передаст его в функцию, которую `(>>=)` передали вторым параметром! Функция вернет действие, и это действие тоже будет выполнено. Если мы потребуем выполнить это действие в интерпретаторе, то сами увидим:

```
Hugs> getLine >>= putStr
hello
hello
Hugs>
```

Первую строку вводил я (она дублировалась на экране), потом я нажал на Enter, первое действие `getLine :: IO String` выполнилось, из него была извлечена строка "hello", и была передана функции `putStr :: String -> IO ()`, которая вернула действие `putStr "hello" :: IO String`, и это действие было выполнено. В результате мы увидели на экране дублированную строку.

Давайте немного перепишем выражение `getLine >>= putStr :: IO ()` :

```
getLine >>= (\s -> putStr s) :: IO ()
```

Ничего не изменилось, я просто более явно подчеркнул, что второй параметр у операции `(>>=)` – это функция. И я явно выделил то место, на которое подставляется извлеченная из первого действия строка – на место формальной переменной `s`.

И вот он, самый главный момент – переменная `s` имеет тип обычной строки! Представляете? Обычная строка, `s :: String`! Не `IO String`, а именно `String`! Ну как же вы не понимаете – это же победа! Раз это обычная строка, то к ней можно применять все наши обычные функции, которые и слышать не слышали про ввод-вывод, и про то, что кому-то там разрешено иметь побочные эффекты…

Именно в тот момент, когда операция `(>>=)` уже выполнила действие `getLine` и извлекла из него строчку, - но до того момента, как эта строчка попадет в функцию `putStr` и опять попадет в жернова императивно программирования… Именно в этот краткий миг мы можем и должны успеть сделать с этой строчкой все, что нам нужно, с помощью наших обычных функций. Например, так:

```
getLine >>= (\s -> putStr (map toUpper s)) :: IO ()
```

Попробуем запустить:

```
Hugs> getLine >>= (\s -> putStr (map toUpper s))
abcde
ABCDE
```

```
Hugs>
```

Ура, получилось! И как красиво и элегантно, а? А вот, к примеру, как выглядит составное действие, которое заключается в чтении у пользователя двух строк и вывод строки, слепленной из них:

```
Hugs> getLine >>= (\s1 -> getLine >>= (\s2 -> putStr (s1++s2)))
abcde
fgh
abcdefgh
Hugs>
```

А мы точно с типами ничего не напутали? Давайте проверим:

```
getLine >>= (\s1 -> getLine >>= (\s2 -> putStr (s1++s2)))
getLine >>= (\s1 -> foo                                  )
```

Вот в этой записи, согласно функции `(>>=)`, `foo` должно иметь тип `IO b` (в том числе и `IO ()` тоже сойдет). А что у нас на месте `foo`? Все нормально, типы совпадают.

```
getLine >>= (\s2 -> putStr (s1++s2)) :: IO ()
```

Вы, конечно, поняли, что про "красиво и элегантно" - это ирония. Конечно же, некрасиво и неэлегантно. Поэтому давайте займемся перестановкой мест слагаемых. Следите за руками:

```
getLine >>= (\s -> putStr (map toUpper s))

s <- getLine >>= (\putStr (map toUpper s))

s <- getLine >>= putStr (map toUpper s)

do
    s <- getLine
    putStr (map toUpper s)
```

Видели? Развернули направление стрелки, переставили местами, убрали лишние значки, добавили для красоты `do` – и получилось уже вполне красиво, безо всякой иронии. Да, нотация `do` – это просто синтаксический сахар для этой самой операции (>>=). Вот как запишется чтение двух строк и вывод конкатенации:

```
do
    s1 <- getLine
    s2 <- getLine
    putStr (s1 + s2)
```

Смотрится так, как будто мы тут переменным `s1` и `s2` последовательно присваиваем значения, получаемые из функции `getLine`. Но вы то помните ту страшную нотацию с `(>>=)`, и понимаете, что каждая строчка вида `s1 <- getLine` – это голова лямбда-функции, а тело этой функции – все, что следует за этой строчкой.

Закончим разбор кухни мы тем, что посмотрим, как все-таки писать функции с лапшой из действий и обработки результатов обычными функциями. Давайте напишем функцию, которая читает символ, если он равен `T`, возвращает `True`, и `False` в обратном случае. Тип нашей функции будет, что логично, `IO Bool`:

```
getBool :: IO Bool
getBool = do
             c <- getChar
             if c == 'T' then True else False
```

И все бы хорошо, но в нотации `do` каждая строчка должна быть действием, и тип действия в последней строчке определяет тип всего выражения, которым является `do`. А что у нас?

```
if c == 'T' then True else False :: Bool
```

А у нас в последней строке – `Bool`, а не `IO Bool`. Нам поможет функция return:

```
return :: a -> IO a
```

Функция `return` – это очень хитрая функция. Она берет значение и возвращает действие, которое может быть выполнено, и из которого может быть извлечено значение. В терминах книжки со слоном, функция `return` берет значение, складывает его в ящик на ножках и возвращает нам этот ящик с невинным видом. Мы можем выполнить полученное от `return` действие: тогда наш ящик нехотя сходит в реальный мир, и сделает вид, что он там совершил великие дела – но на самом деле он просто сразу вернется обратно и позволит нам достать из него то же самое значение, какое функция `return` в него положила.

Зачем все это нужно? Чтобы удовлетворить строгую и неподкупную систему типов языка Haskell. Раз в нотации `do` требуется действие, а не значение – значит вынь да положь минимально возможное (простейшее) действие, которое это значение возвращает.

Короче говоря, правильно будет функцию `getBool` написать так:

```
getBool :: IO Bool
getBool = do
             c <- getChar
             if c == 'T'
                 then return True
                 else return False
```

А вот как реализуется функция `getLine`, при условии, что уже есть функция `getChar`:

```
getLine     :: IO String
getLine     =  do c <- getChar
                  if c == '\n'
                       then return ""
                       else do s <- getLine
                               return (c:s)
```

Что делает эта функция? Выполняет действие `getChar` и извлекает из него символ. Проверяет, равен ли этот символ символу окончания строки, если да, то строка уже возвращена. Если же нет, то запускает рекурсивное действие той же самой функции `getLine`, извлекает из этого действия строку, и свой символ приписывает к началу этой строки, возвращая ее всю целиком.
Звучит легко, но написать такую функцию и не ошибиться с типами – все равно, что пройти по минному полю. Давайте разберемся.

Действие `getChar` позволяет извлечь из него `c :: Char`. Действие `getLine`, предположительно,

позволяет извлечь из него `s :: String`, выражение `(c:s)` дает нам опять `String`, и `return (c:s)` дает нам `IO String`.

Значит типом выражения

```
do s <- getLine
   return (c:s)
```

является `IO String`. Это выражение находится в ветке `else`, значит в ветке `then` должно находиться выражение того же типа. Проверим: `return "" :: IO String` – все правильно, значит типом всего выражения `if` является тоже `IO String`.
Фуу, устал, - но осталось совсем чуть-чуть. Раз типом выражения `if` является тоже `IO String`, а это выражение идет последним в списке `do`, значит и типом всего выражения `do` является `IO String`, что и требовалось доказать.

Пример программы, производящей нетривиальное преобразование текстового файла

Давайте напоследок посмотрим, как можно легко писать простенькие утилиты, анализирующие и преобразующие текстовые файлы.

Для этого нам понадобится несколько вспомогательных функций:

```
lines     :: String -> [String]
words     :: String -> [String]
unlines   :: [String] -> String
unwords   :: [String] -> String
```

Функция `lines` разбивает строку на список строк в тех местах, где встречается символ перевода строки `'\n'`, - очень полезная функция, если все содержимое файла читается разом одной функцией `readFile`. Функция `words` разбивает строку на список строк в тех местах где встречается пробел, табуляция или что-то подобное. Функции `unlines` и `unwords` производят обратные преобразования.

Итак, напишем, например, функцию, которая читает текстовый файл, и выводит на экран список строк, причем в каждой строке находится число слов в этой строке:

```
main :: IO ()
main = do
    contents <- readFile "input.txt"
    putStr (process contents)

process :: String -> String
process =
    unlines .
    map show .
    map length .
    map words .
    lines
```

Пусть у нас есть файл input.txt с таким содержанием:

```
It was the lark, the herald of the morn,
No nightingale. Look, love, what envious streaks
Do lace the severing clouds in yonder east.
```

```
Night's candles are burnt out, and jocund day
Stands tiptoe on the misty mountain tops.
I must be gone and live, or stay and die.
```

Запустим функцию `main`:

```
SomeModule> main
9
7
8
8
7
10
```

Все, как и ожидалось. Осталось только разобраться, как мы добились этого столь короткой программой. Обратите внимание, что в главной функции `process` вообще нигде не упоминаются данные. Да, `process` имеет дело со строкой, но сама строка в определении функции нигде не упоминается! Вся функция состоит из композиций, функций высших порядков и частичного применения.

Для облегчения понимания мы посмотрим, какой путь проходит строка `s`, в которой находится содержимое всего файла, через всю эту сложную композицию функций.

Сначала строка со всем содержимым передается функции `lines`, и из нее получается список строк `[String]`:

```
["It was the lark, the herald of the morn,",
...
"I must be gone and live, or stay and die."]
```

Потом к этому списку строк применяется функция `map words`, - то есть, по сути, к каждой строке этого списка применяется функция words, и получается список списков строк `[[String]]`:

```
[["It","was","the","lark,","the","herald","of","the","morn,"],
...
["I","must","be","gone","and","live,","or","stay","and","die."]]
```

Дальше к этому списку применяется функция `map length`, то есть к каждому подсписку применяется функция `length`, и получается список целых чисел `[Int]`:

```
[9,
...
10]
```

Далее к этому списку применяется функция `map show`, а она применяет `show` к каждому элементу списка, и получается опять список строк `[String]`:

```
["9",
...
"10"]
```

Ну и наконец, функция `unlines` разворачивает список строк в одну строку, вставляя символы перехода между строчками, в результате чего получается одна строка `String`:

```
"9\n7\n8\n8\n7\n10"
```

За счет чего функция получилась такой короткой? Критики функционального программирования наверняка скажут – да за счет использования большого количества стандартных функций. И это правда – но только отчасти. Стандартных функций на самом деле много, но они не являются специфическими, созданными только для одной задачи. Наоборот, функции `map`, `show`, `length` являются настолько общими, насколько это вообще возможно!

Короткой функция `process` получилась благодаря тому, что функциональный язык Haskell предоставляет широчайшие возможности для комбинирования, повторного использования кода. Функции высших порядков, частичное применение, композиция – это все ключевые возможности, которые позволяют из уже написанных кусочков достаточно общего кода собрать в нужном месте специфическую функциональность, нужную для решения конкретной задачи.

Пример решения задачи: Поиск в пространстве состояний

Напоследок, давайте решим простую поисковую задачу. Каждый «сиплюсплюсник» должен в своей жизни написать свой класс для строки, а каждый начинающий «хаскелист» в своей жизни должен перевезти через реку козу, капусту и волка.

Саму задачу знают все: надо найти, в какой последовательности, кого, куда должен перевозить мужик через реку, чтобы в итоге все оказались на другой стороне, и при этом волк не съел бы козу, а коза не съела бы капусту.

Это типичная задача на поиск в пространстве состояний. Есть текущее состояние: кто на каком берегу находится. Есть правила перехода из одного состояния в другое (перевезти кого-то одного на другой берег или переехать на другой берег самому). И есть «плохие», то есть запрещенные состояния: волк с козой без контроля мужика, или коза с капустой без этого же контроля.

Самый первый вопрос, который требуется решить – в какой структуре данных хранить текущее состояние. Это не такой простой вопрос, как кажется. От выбора структуры будет зависеть не только краткость и понятность кода. В некоторых задачах просто от того, что именно вы посчитаете за состояние, будет зависеть, удастся ли решить задачу вообще!
В нашем случае все проще: текущее состояние – это описание того, кто на каком берегу находится. Обратите внимание, никаких промежуточных состояний у нас не предусмотрено – хотя в реальной жизни процесс переезда через реку занимал бы некоторое время, и в каких-то других условиях это оказалось бы важно.

Довольно слов, начнем писать код.

Через массивы и последовательность промежуточных состояний

```
module Boat where

import List
import Array

-- объект
data Item = Wolf | Goat | Cabbage | Farmer
        deriving (Show, Eq, Ord, Ix)

-- положение
```

```
data Location = L | R
        deriving (Show, Eq, Ord)

-- обратное положение
from L = R
from R = L
```

Мы объявили несколько типов данных. Перевозимые объекты – это `Wolf` или `Goat` или `Cabbage` или `Farmer`, причем мы попросили компилятор считать эти объекты отображаемыми в строку (`Show`), сравниваемыми (`Eq`) и упорядоченными (`Ord`).

Положений у каждого объекта может быть только два: `L` и `R`, соответственно, на левом или на правом берегу. Причем эти положения – противоположные: мы сразу написали функцию, которая работает аналогично функции `not` для логических значений.
Вы спросите, почему не использовать тогда сами значения `True` и `False`, вместо `L` и `R`? Потому что мы не программируем – мы описываем задачу. Положение `Location` – есть положение, а использование вместо него обычного `Bool` – это грязный хак, недостойный декларативных программистов.

```
-- позиция: где кто
type Position = Array Item Location
```

Вот она, самая главная структура данных. Мы еще не сталкивались с такой структурой данных, как массив – ну вот, заодно и разберемся. Мы сказали, что `Position` в нашем случае – это массив по индексу `Item`, в котором хранятся `Location`. То есть, позиция показывает для каждого индекса `Item` некоторое значение `Location`, что нам и нужно.

Работают с массивами с помощью следующих функций (их добавлять не надо, они нам доступны, потому что мы подключили модуль `import Array`):

```
array :: Ix a => (a,a) -> [(a,b)] -> Array a b

listArray :: Ix a => (a,a) -> [b] -> Array a b

(!) :: Ix i => Array i e -> i -> e

indices :: Ix i => Array i e -> [i]

elems :: Ix i => Array i e -> [e]

assocs :: Ix i => Array i e -> [(i, e)]

(//) :: Ix i => Array i e -> [(i, e)] -> Array i e
```

Первые две функции используются для создания массивов. В `array` передают пару из левой и правой границ индекса и список кортежей «индекс-значение». Функция `listArray` подлиннее названием, но поэкономнее: в нее передают опять пару из левой и правой границ индекса и список значений. Вот так, например, мы опишем стартовую и целевую позицию:

```
-- начальная и целевая позиция
startPosition = listArray (Wolf, Farmer) [L, L, L, L]
goalPosition = listArray (Wolf, Farmer) [R, R, R, R]
```

Давайте проверим:

```
startPosition → array (Wolf,Farmer) [(Wolf,L),(Goat,L),(Cabbage,L),(Farmer,L)]
```

Функция `(!)` используется для доступа к значению в массиве по ключу. Например,

```
startPosition ! Farmer → L

goalPosition ! Wolf → R
```

Напишем функцию, которая определяет, является ли определенная позиция допустимой:

```
-- неправильная позиция: без контроля человека остаются
-- волк с козлом или козел с капустой
wrongPosition :: Position -> Bool
wrongPosition p =
    all (/= p!Farmer) [p!Wolf, p!Goat]  ||
    all (/= p!Farmer) [p!Cabbage, p!Goat]
```

Если вдруг сюда заглянул тот, кто боится функций высших порядков, то для него мы эту функцию перепишем так:

```
wrongPosition p =
    (p!Wolf /= p!Farmer && p!Goat /= p!Farmer) ||
    (p!Cabbage /= p!Farmer && p!Goat /= p!Farmer)
```

Из остальных функций для работы с массивами нам понадобится функция `assocs`, преобразующая массив обратно в список кортежей «индекс-значение» и функция `(//)`, которая заменяет в массиве значения по заданному списку «индекс-значение»:

```
startPosition // [(Goat,R),(Farmer,R)] →
      array (Wolf,Farmer) [(Wolf,L),(Goat,R),(Cabbage,L),(Farmer,R)]
```

Обратили внимание, как в стартовой позиции заменилось положение козы и мужика? Все, теперь мы можем писать решение нашей задачи. Во-первых, самое главное – какие новые позиции можно получить из старой позиции путем перевоза кого-нибудь на другой берег?

Пусть есть какая-нибудь позиция `p`. Кого фермер может с собой взять? Логично предположить, что тех, кто находится с ним на одном и том же берегу. Как таких найти? Надо взять список кортежей `(кто,где)` и взять кортежи только про тех, кто находится на одном берегу с мужиком. Например, для стартовой позиции это:

```
filter ((== startPosition!Farmer) . snd) $ assocs startPosition →
    [(Wolf,L),(Goat,L),(Cabbage,L),(Farmer,L)]
```

То есть из стартовой позиции на другой берег мужик может взять кого угодно. Кстати, в том числе и себя самого – но нас это устраивает: пусть мужик всегда берет кого-нибудь одного, а если он берет на другой берег самого себя – это означать будет, что он переезжает один. Это проще, чем рассматривать отдельно возможность переезда на другой берег без груза.

Итак, если для позиции мы знаем, кого мужик может из нее перевезти на другой берег, то для каждого из таких вариантов у нас должна появиться новая позиция, в которой мужик и его груз меняют свое положение на противоположное, а все остальные остаются на своих местах:

```
-- шаг переправы с берега на берег с кем-нибудь: какие варианты можно получить
step :: Position -> [Position]
step p =
    [p // [(who, from wher)] // [(Farmer, from wher)] | (who,wher) <- whom] where
    whom = filter ((== p!Farmer) . snd) $ assocs p
```

В этом генераторе списка мы выбираем с помощью `(who,wher) <- whom` вариант перевозки, и каждый из этих вариантов при помощи `p // [(who, from wher)] // [(Farmer, from wher)]` дает нам новую возможную позицию. Проверим?

```
step startPosition →

    [array (Wolf,Farmer) [(Wolf,R),(Goat,L),(Cabbage,L),(Farmer,R)],
     array (Wolf,Farmer) [(Wolf,L),(Goat,R),(Cabbage,L),(Farmer,R)],
     array (Wolf,Farmer) [(Wolf,L),(Goat,L),(Cabbage,R),(Farmer,R)],
     array (Wolf,Farmer) [(Wolf,L),(Goat,L),(Cabbage,L),(Farmer,R)]
    ]
```

Из стартовой позиции получаем 4 варианта, в каждом из которых кто-нибудь да переезжает с одного берега на другой. Но ведь какие-то из этих позиций наверняка неправильные? Давайте оставим только правильные!

```
filter (not.wrongPosition) $ step startPosition →

    [array (Wolf,Farmer) [(Wolf,L),(Goat,R),(Cabbage,L),(Farmer,R)]
    ]
```

Негусто, да? Оказывается, в начальной позиции у мужика выбора нет – надо везти козу на другой берег. Как нам теперь продолжить поиск? Надо к каждой из полученных позиций опять применить функцию `step` - конечно же, функцией `map`. В результате у нас получится список списков позиций, и мы его сольем в один список функцией `concat`. Ну или воспользуемся функцией `concatMap`, которая сразу делает и то и другое. И, конечно, про фильтрацию не забудем:

```
filter (not.wrongPosition) $ concatMap step [startPosition] →

    [array (Wolf,Farmer) [(Wolf,L),(Goat,R),(Cabbage,L),(Farmer,R)]
    ]
```

А теперь еще шаг!

```
filter (not.wrongPosition) $ concatMap step
    $ filter (not.wrongPosition) $ concatMap step
        [startPosition] →

    [array (Wolf,Farmer) [(Wolf,L),(Goat,L),(Cabbage,L),(Farmer,L)],
     array (Wolf,Farmer) [(Wolf,L),(Goat,R),(Cabbage,L),(Farmer,L)]
    ]
```

На втором шаге, оказывается, у мужика есть два варианта: везти обратно козу – или перебраться обратно одному. В первом случае мы возвращаемся к исходной ситуации (это проблема на самом деле, мы к ней вернемся). Конечно же, у вас уже руки чешутся применить функцию `iterate`:

```
searchI = iterate (filter (not.wrongPosition) . concatMap step) [startPosition]
```

Каждый элемент списка `searchI` (а это будет список списков позиций) будет хранить список позиций, получающихся на очередном шаге алгоритма поиска. Кстати, думаю, вы уже давно удивляетесь, как это у меня получаются такие более или менее читаемые результаты? На самом деле у меня получается то же самое, что у вас, - но я вручную форматирую результат. Давайте-ка такой список позиций преобразуем во что-нибудь более читаемое. Позиция умеет преобразовываться в строку сама (она принадлежит `Show`), строки мы умеем печатать с помощью `putStrLn`, но у нас список позиций, а значит, распечатать список позиций мы сможем с помощью `map (putStrLn.show.assocs)`, что даст нам список `IO ()`, которые все надо подряд выполнить с помощью функции `sequence`:

```
Boat> sequence $ map (putStrLn.show.assocs) $ searchI !! 0
[(Wolf,L),(Goat,L),(Cabbage,L),(Farmer,L)]

Boat> sequence $ map (putStrLn.show.assocs) $ searchI !! 1
[(Wolf,L),(Goat,R),(Cabbage,L),(Farmer,R)]

Boat> sequence $ map (putStrLn.show.assocs) $ searchI !! 2
[(Wolf,L),(Goat,L),(Cabbage,L),(Farmer,L)]
[(Wolf,L),(Goat,R),(Cabbage,L),(Farmer,L)]

Boat> sequence $ map (putStrLn.show.assocs) $ searchI !! 3
[(Wolf,L),(Goat,R),(Cabbage,L),(Farmer,R)]
[(Wolf,R),(Goat,R),(Cabbage,L),(Farmer,R)]
[(Wolf,L),(Goat,R),(Cabbage,R),(Farmer,R)]
[(Wolf,L),(Goat,R),(Cabbage,L),(Farmer,R)]
```

Итак, сначала у нас все на левом берегу, на втором шаге появляется возможность получить мужика с козой на правом берегу. На третьем шаге из мужика с козой на правом берегу мы можем получить или опять исходную позицию, или козу на правом берегу, а всех остальных на левом, и так далее.

Где же и когда появится, наконец, интересующая нас позиция целевая – когда все на правом берегу?

```
Boat> elem goalPosition $ searchI !! 0
False
Boat> elem goalPosition $ searchI !! 1
False
Boat> elem goalPosition $ searchI !! 2
False
Boat> elem goalPosition $ searchI !! 3
False
Boat> elem goalPosition $ searchI !! 4
False
Boat> elem goalPosition $ searchI !! 5
False
Boat> elem goalPosition $ searchI !! 6
False
Boat> elem goalPosition $ searchI !! 7
True
```

Ура-ура-ура! За семь шагов можно перевезти всех! Кстати, сколько у нас там позиций возможных получилось на седьмом шаге?

```
length $ searchI !! 7 → 86
```

Ничего себе! А уникальных из них сколько?

```
length $ nub $ searchI !! 7 → 5
```

А сколько всего уникальных позиций было просмотрено на шагах с нулевого по седьмой?

```
length $ nub $ concat $ take 8 searchI → 10
```

> Замечание на полях. По-моему, ничто так не демонстрирует мощность функционального программирования, как вот эта скорость получения ответов на возникающие вопросы. Вы только попытайтесь себе представить, сколько кода пришлось бы написать на каком-нибудь сиплюсплюсе (уже после того, как мы написали бы на нем аналог функции `searchI`) для получения ответа на последние три вопроса!
> А за счет чего получается так кратко? За счет широких возможностей склеивать требуемое решение из достаточно общих функций `concat`, `take`, `nub`, `length` и так далее.

На самом деле, легко понять, что возможных позиций всего 16 (два в четвертой степени), и из них 6 «плохие», так что все возможные позиции мы рассмотрели. Но сколько раз мы их рассмотрели?

```
length $ concat $ take 8 searchI → 158
```

Нда… Понятно дело, если бы задачка была потруднее, мы утонули бы в повторяющихся позициях и ничего бы так и не нашли за обозримое время.

Кроме этого, есть проблема и поважнее. Ну узнали мы, что решение этой задачи существует. А как его вывести? Где та последовательность состояний (или последовательность переходов), которая приводит нас к успеху, к целевому состоянию? А нигде – мы теряем эту последовательность!

Придется отступить назад и еще раз подумать. Что такое – решение нашей задачи? Видимо, это последовательность позиций от начальной – к целевой. Так и запишем:

```
-- решение: последовательность позиций (самая последняя - в начале списка)
type Solution = [Position]
```

Про начальную или целевую в типе данных ничего не сказано – это просто последовательность позиций. Мы будем искать такую последовательность позиций, у которой в конце списка – исходная позиция, а в начале списка – последняя достигнутая позиция (для удобства будем хранить позиции в обратном порядке, потому что добавлять в начало списка проще, чем в конец).

Как будем искать? Да все тем же перебором, конечно. Если у нас есть какое-то решение $[P_N,P_{N-1},\dots P_0]$, то мы должны из него построить кучу новых потенциальных решений $[P_{N+1},P_N,P_{N-1},\dots P_0]$ для каждой позиции p_{N+1}, в которую можно попасть из позиции p_N. Вот как это делается:

```
-- построение нового списка возможных решений из старого
stepSolution :: [Solution] -> [Solution]
```

```
stepSolution sols =
    [ (newpos:sol) |
        sol <- sols, newpos <- step (head sol),
        not (wrongPosition newpos)]
```

Что здесь происходит?

- Мы берем набор решений `sols` и вынимаем по одному каждое решение `sol` с помощью генератора `sol <- sols`.
- После этого вытаскиваем из `sol` последнюю достигнутую позицию: `head sol`.
- Для этой позиции находим все возможные позиции, которые из нее следуют путем применения функции шага: `step (head sol)`.
- Выбираем по одной все возможные позиции с помощью `newpos <- step (head sol)`.
- Оставляем с помощью условия `not (wrongPosition newpos)` только корректные возможные новые позиции.
- Генерируем новое решение, добавляя с помощью `(newpos:sol)` новую позицию `newpos` в голову старого решения `sol`.

Таким образом, эта функция генерирует новый набор позиций из старого набора позиций. Остается только зациклить эту генерацию, начиная с единственного решения, состоящего из единственной стартовой позиции:

```
-- итеративный процесс построения возможных решений,
-- для поиска среди них того, которое является ответом
search :: [[Solution]]
search = iterate stepSolution [[startPosition]]
```

Мы получили список списков решений, надо их все слить в один список решений, а потом найти там первое решение, которое является ответом. Это такое решение, у которого в голове списка позиций находится целевая позиция:

```
-- нахождение первого решения, которое является ответом
solution :: [Position]
solution =  head $ filter ((==goalPosition).head) $ concat $ search
```

Ну и еще раз, функция вывода решения на экран в читабельном виде:

```
-- вывод решения на экран
showSolution = sequence $ map (putStrLn.show.assocs) solution
```

Смотрим (я выделил сам шрифтом тот объект, который перемещает на каждом шаге мужик):

```
[(Wolf,R),(Goat,R),(Cabbage,R),(Farmer,R)]
[(Wolf,R),(Goat,L),(Cabbage,R),(Farmer,L)]
[(Wolf,R),(Goat,L),(Cabbage,R),(Farmer,R)]
[(Wolf,R),(Goat,L),(Cabbage,L),(Farmer,L)]
[(Wolf,R),(Goat,R),(Cabbage,L),(Farmer,R)]
[(Wolf,L),(Goat,R),(Cabbage,L),(Farmer,L)]
[(Wolf,L),(Goat,R),(Cabbage,L),(Farmer,R)]
[(Wolf,L),(Goat,L),(Cabbage,L),(Farmer,L)]
```

Попытаться преобразовать эту последовательность во что-то еще более читаемое (ближе к человеческому языку объясняющее, кого, куда, когда везут) я оставлю вам в качестве дополнительного задания.

Как, впрочем, и вторую проблему: генерация дубликатов никуда не делась. Среди полученных на

нашем седьмом шаге решений будет и такое, где мужик катается туда-сюда между берегами один, никого никуда не перевозя. Это происходит потому, что мы не проверяем, встречалось ли у нас уже когда-то определенное состояние.

Решение для тех, кто не хочет разбираться сам

Для тех, кто не хочет разбираться, а хочет просто посмотреть, как выглядит решение этой задачи на Haskell, приведем его здесь целиком:

```
module Boat where

import List
import Array

-- лодка и волк-коза-капуста

-- объект
data Item = Wolf | Goat | Cabbage | Farmer
        deriving (Show, Eq, Ord, Ix)

-- положение
data Location = L | R
        deriving (Show, Eq, Ord)

-- обратное положение
from L = R
from R = L

-- позиция: где кто
type Position = Array Item Location

-- начальная и целевая позиция
startPosition = listArray (Wolf, Farmer) [L, L, L, L]
goalPosition = listArray (Wolf, Farmer) [R, R, R, R]

-- неправильная позиция: без контроля человека остаются
-- волк с козлом или козел с капустой
wrongPosition :: Position -> Bool
wrongPosition p =
    all (/= p!Farmer) [p!Wolf, p!Goat]  ||  all (/= p!Farmer) [p!Cabbage, p!Goat]

-- шаг переправы с берега на берег с кем-нибудь: какие варианты можно получить
step :: Position -> [Position]
step p =
    [p // [(who, from wher)] // [(Farmer, from wher)]  | (who,wher) <- whom] where
        whom = filter ((== p!Farmer) . snd) $ assocs p

-- решение: последовательность позиций (самая последняя - в начале списка)
type Solution = [Position]

-- построение нового списка возможных решений из старого
stepSolution :: [Solution] -> [Solution]
stepSolution sols =
 [(newpos:sol) | sol <- sols, newpos <- step (head sol), not $ wrongPosition newpos]

-- итеративный процесс построения возможных решений,
-- для поиска среди них того, которое является ответом
```

```
search :: [[Solution]]
search = iterate stepSolution [[startPosition]]

-- нахождение первого решения, которое является ответом
solution :: [Position]
solution =  head $ filter ((==goalPosition).head) $ concat $ search

-- вывод решения на экран
showSolution = sequence $ map (putStrLn.show.assocs) solution
```

Через списки, лог истории и уникальную очередь

Рассмотрим альтернативное решение той же самой задачи. Начнем с совершенно другого представления, с другой модели данных. Может быть, мы зря связались с массивами? Ведь массивы хороши тогда, когда нужно обращаться к элементам в произвольном порядке – а нужен ли он нам здесь, произвольный порядок? Попробуем обойтись списком, и кроме того, вынесем мужика в отдельную сущность – среди объектов перевозимых его больше не будет:

```
-- объект
data Item = Wolf | Goat | Cabbage
        deriving (Show, Eq, Ord)

-- положение: где человек-лодка + кто на левом берегу + кто на правом + лог истории
data Position = L [Item] [Item] String | R [Item] [Item] String
        deriving Show
```

Перевозимые объекты теперь – все, кроме мужика. Позиция – это `(L список список история)` или `(R список список история)`, где `L` и `R` кодирует то, где сейчас находится мужик, два списка кодируют, соответственно, текущих обитателей левого и правого берега, а в `истории` мы будем хранить текстовую информацию о том, как мы до этой позиции дошли. Кстати, теоретически этой информации совсем не обязательно быть строкой – там может быть что угодно, в том числе и опять позиция – предыдущая к этой (вспомните про рекурсивные типы данных).

И важный момент – операцию сравнения для позиций нам придется написать самостоятельно, потому что компилятор за нас при сравнении двух позиций учитывал бы и их историю тоже, а нам это совсем не нужно: две позиции будем считать одинаковыми, безотносительно к их истории.

```
-- два положения одинаковые, если все одинаковое, кроме лога
instance Eq Position where
        (==) (L l1 r1 _) (L l2 r2 _)  =  l1 == l2 && r1 == r2
        (==) (R l1 r1 _) (R l2 r2 _)  =  l1 == l2 && r1 == r2
        _         == _          =  False
```

Начальная и целевая позиция, проверка корректности позиции реализуется аналогично, с поправкой на изменившиеся структуры данных:

```
-- начальная и целевая позиция
startPosition = L [Wolf, Goat, Cabbage] [] ""
goalPosition = R [] [Wolf, Goat, Cabbage] ""

-- неправильная компания: козел с капустой или козел с волком в наличии
wrongCompany c =
    null ([Goat, Cabbage] \\ c) ||
    null ([Goat, Wolf] \\ c)
```

```
-- неправильная позиция: без контроля человека остается плохая компания
wrongPosition (L left right _) = wrongCompany right
wrongPosition (R left right _) = wrongCompany left
```

Одна из самых главных функций, как и в прошлом варианте – какие варианты позиции можно получить из заданной позиции. Пусть позиция равна `(L left right hist)`, то есть, мужик находится на левом берегу, в `left` хранится список тех, кто с ним, а в `right` – список тех, кто на противоположном берегу, ну а в `hist` содержится описание того, как они дошли до жизни такой.

```
-- шаг переправы с берега на берег с кем-нибудь: какие варианты можно получить
step (L left right hist) =
    [R left right (hist++"Move Right ")]     ++
    [R (left \\ [who]) (sort $ who:right) (hist++"Take "++show who++" Right ")
        | who <- left]
```

Работает наша функция так:

- Выбирается кто-то, кто поедет с мужиком на другой берег: `who <- left`.
- Для каждого варианта строится новая позиция, причем:
    - мужик в новой позиции – на правом берегу `(R _ _ _)`,
    - с левого берега пассажир убывает: `(left \\ [who])`,
    - на правый берег он прибывает и сортируется там по росту: `(sort $ who:right)`,
    - в лог новой позиции дописывается, что мужик перевез who на правый берег.
- Кроме того, добавляется еще один вариант, когда мужик без попутчиков переезжает на правый берег

Если же мужик находился на правом берегу, функция выглядит с точностью до наоборот:

```
step (R left right hist) =
    [L left right (hist++"Move Left ")]      ++
    [L (sort $ who:left) (right \\ [who]) (hist++"Take "++show who++" Left ")
        | who <- right]
```

Проверим?

```
step startPosition →

    [R [Wolf,Goat,Cabbage] [] "Move Right ",
     R [Goat,Cabbage] [Wolf] "Take Wolf Right ",
     R [Wolf,Cabbage] [Goat] "Take Goat Right ",
     R [Wolf,Goat] [Cabbage] "Take Cabbage Right "
    ]
```

Отлично, теперь надо написать итеративное применение этой функции. Но мы учтем замечания и не допустим дублирование позиций. Создадим рекурсивную функцию `queue`. Первым параметром она будет принимать список позиций, которые уже встречались, и потому повторяться не должны. Вторым параметром у нее будет очередь из позиций, которые надо проверить. Ну и возвращать она будет список позиций, до которых она добралась в процессе своей работы:

```
queue :: [Position] -> [Position] -> [Position]

-- очередь поиска
queue old [] = []
queue old (p:ps) = p : queue (p:old) (nub $ ps ++ candidates) where
    candidates = filter (not.wrongPosition) (step p) \\ old
```

Как эта функция работает? Она берет очередную позицию `p`, применяет к ней функцию `step` и получает, тем самым, список позиций, в которые можно из `p` попасть. Далее оставляются только корректные позиции и только те, что раньше не встречались – они все становятся кандидатами `candidates`.
Далее функция `queue` возвращает только что просмотренную позицию `p` (ее вернуть нужно как можно раньше, - ведь она может оказаться целевой!), и запускает сама себя, при этом:

- позиция `p` добавляется в список уже встреченных, и поэтому больше не повторится;
- кандидаты из `candidates` добавляются в общую очередь так, чтобы не было повторений.

Осталось только запустить эту очередь и найти в ей целевую позицию:

```
-- запускаем поиск по очереди и находим в нем решение
solution = head $ dropWhile (/= goalPosition) $ queue [] [startPosition]
```

Проверим?

```
Boat> solution
R [] [Wolf,Goat,Cabbage] "Take Goat Right Move Left Take Wolf Right Take Goat Le
ft Take Cabbage Right Move Left Take Goat Right "
```

Ура! Решение найдено и описано вполне человеческим языком!

Обратите внимание, функцию queue можно было написать немножко по-другому, с помощью хвостовой рекурсии:

```
-- очередь поиска
queue old [] = old
queue old (p:ps) = queue (p:old) (nub $ ps ++ candidates) where
    candidates = filter (not.wrongPosition) (step p) \\ old
```

Ответ при этом не изменится – но вот сама функция станет опасной. В этом варианте, она возвратит всю очередь только в том случае, если эта очередь когда-нибудь закончится. Но вы же понимаете, что в какой-нибудь другой задаче поиск может никогда и не остановиться (или остановиться теоретически через «нцать» лет, что нас тоже не устраивает) – хотя решение вполне может находиться чуть ближе, чем совсем уж в бесконечности.
Поэтому в данном случае для функции `queue` очень важно возвращать рассмотренные элементы очереди как можно раньше – в этом случае ленивый язык может и не потребовать вычислять всю остальную очередь.

Решение для тех, кто не хочет разбираться сам

Приведем здесь наше новое решение целиком, для тех, кому просто интересно, как оно может выглядеть:

```
module Boat where

import List

-- объект
data Item = Wolf | Goat | Cabbage
        deriving (Show, Eq, Ord)
```

```
-- положение: где человек-лодка + кто на левом берегу + кто на правом + лог истории
data Position = L [Item] [Item] String | R [Item] [Item] String
        deriving Show

-- два положения одинаковые, если все одинаковое, кроме лога
instance Eq Position where
        (==) (L l1 r1 _) (L l2 r2 _)  =  l1 == l2 && r1 == r2
        (==) (R l1 r1 _) (R l2 r2 _)  =  l1 == l2 && r1 == r2
        _         == _          =  False

-- начальная и целевая позиция
startPosition = L [Wolf, Goat, Cabbage] [] ""
goalPosition = R [] [Wolf, Goat, Cabbage] ""

-- неправильная компания
wrongCompany c =
    null ([Goat, Cabbage] \\ c) ||
    null ([Goat, Wolf] \\ c)

-- неправильная позиция: без контроля человека остается плохая компания
wrongPosition (L left right _) = wrongCompany right
wrongPosition (R left right _) = wrongCompany left

-- шаг переправы с берега на берег с кем-нибудь: какие варианты можно получить
step (L left right hist) =
    [R left right (hist++"Move Right ")]     ++
    [R (left \\ [who]) (sort $ who:right) (hist++"Take "++show who++" Right ")
        | who <- left]

step (R left right hist) =
    [L left right (hist++"Move Left ")]      ++
    [L (sort $ who:left) (right \\ [who]) (hist++"Take "++show who++" Left ")
        | who <- right]

-- очередь поиска
queue old [] = old
queue old (p:ps) = queue (p:old) (nub $ ps ++ candidates) where
    candidates = filter (not.wrongPosition) (step p) \\ old

-- запускаем поиск по очереди и находим в нем решение
solution = head $ dropWhile (/= goalPosition) $ queue [] [startPosition]
```

Кто дочитал до этого места и знает, чему равен xyz, может обращаться ко мне за призом. Всего хорошего!

Задачник

Приведенный ниже задачник построен в форме пяти лабораторных работ, каждая из которых объединяет набор заданий на написание функций, объединенных общей тематикой. Подразумевается, что задачи будут выполняться друг за другом параллельно изучению теоретической части - или сразу после ее изучения.

Пояснения и обозначения

Как правило, в большинстве заданий приведены названия уже существующих стандартных функций. Требуется запросить в интерпретаторе тип функций, создать примеры входных данных для них и убедиться, что функции работают именно так, как вы представляете. В заданиях второго типа требуется написать замену стандартной функции, или же написать функцию, аналогичной которой нет среди стандартных. Такие функции лучше писать с постфиксом `My`, чтобы имена их не пересекались со стандартными, например, `headMy`. После написания такой функции, заменяющей стандартную, можно в дальнейшем пользоваться стандартной.

В большинстве случаев, приводимые имена функций совпадают со стандартными именами функций, делающих то же самое в стандартных модулях. Для функций с постфиксом `My` не имеется стандартных аналогов.

`Вот таким шрифтом обозначаются куски кода, имена функций и сигнатуры типа.`

> Вот таким образом оформлены дополнительные задания, выполнять которые необходимо только тем, кто хочет научиться чуть большему, чем остальные. Остальные могут их пропускать.

Лабораторная работа 1

Простейшие функции

Запросите в интерпретаторе и объясните тип функций, проверьте их поведение на примерах входных данных:

`(+), (-), (*), (/), div, mod, (^)`

Запросите в интерпретаторе и объясните тип функций, проверьте их поведение на примерах входных данных, и напишите свою реализацию каждой функции под другим именем (например, под именами `sumMy`, `productMy`, и т.д.):

`sum, product`
`max, min`
`maximum, minimum`
`even, odd`
`gcd, lcm` (greatest common divisor, least common multiple)
`(^)`

Напишите функцию `factMy :: Integer -> Integer`, которая вычисляет факториал заданного числа, с помощью простой рекурсии.

Напишите функцию `fibMy :: Integer -> Integer`, которая вычисляет заданное по номеру число

Фибоначчи с помощью простой рекурсии. Найдите, какое максимально большое число позволяет вам найти ваша система. Объясните, почему рекурсия так долго работает.

> Попробуйте придумать более эффективную функцию вычисления числа Фибоначчи в виде рекурсивной функции вида `fibMy' n prev prevprev = ...`, которая принимает в качестве параметров предыдущее и пред-предыдущее число Фибоначчи.

Простейшие логические функции

Запросите в интерпретаторе и объясните тип функций, проверьте их поведение на примерах входных данных:

```
(>), (<), (==), (/=), (>=), (<=), (&&), (||), not
```

Запросите в интерпретаторе и объясните тип функций, проверьте их поведение на примерах входных данных, и напишите свою реализацию каждой функции под другим именем:

```
and, or
```

Простейшие списочные функции

Запросите в интерпретаторе и объясните тип функций, проверьте их поведение на примерах входных данных:

```
(:), null
```

Запросите в интерпретаторе и объясните тип функций, проверьте их поведение на примерах входных данных, и напишите свою реализацию каждой функции под другим именем:

```
head, tail
last, init
length, (!!), (++), concat
take, drop
reverse, elem, replicate
```

Напишите функцию `lookupMy :: Eq a => a -> b -> [(a, b)] -> b`, которая берет значение `xa` типа `a`, значение `xb` типа `b`, список кортежей `[(a, b)]`, находит кортеж, в котором первый элемент равен `xa`, и выводит второй элемент этого кортежа. Если такого кортежа не нашлось, функция должна вернуть `xb`.

Напишите функцию `substrMy :: [a] -> Int -> Int -> [a]`, которая возвращает все элементы списка начиная с какого-то и заканчивая каким-то с помощью функций `take` и `drop`.

Напишите функцию `strReplaceMy :: Eq a => [a] -> [a] -> [a] -> [a]`, которая принимает три списка и заменяет в третьем списке все вхождения первого списка на второй список с помощью функций `length`, `(==)`, `take` и `drop`.

Напишите функцию `elemIndices :: Eq a => a -> [a] -> [Int]`, которая находит, под какими индексами в списке встречается заданный элемент.

Напишите функцию `strPosMy :: Eq a => [a] -> [a] -> [Int]`, которая находит все вхождения первого списка во второй и возвращает список номеров элементов, с которых эти вхождения

начинаются с помощью функций `length`, `(==)`, `take` и `drop`.

Напишите функцию `strRotateMy :: [a] -> Int -> [a]`, которая производит заданное количество поворотов заданного списка. Операция кругового поворота над списком заключается в том, что последний элемент списка переносится в его начало. Например, `strRotate [1,2,3,4,5,6] 2 = [5,6,1,2,3,4]`.

> Напишите функцию `unevenHandWritingMy :: String -> String`, которая берет строку и возвращает ее же, но каждая третья буква должна стать прописной, если была строчной и наоборот.

Лабораторная работа 2

Символьные функции

Запросите в интерпретаторе и объясните тип функций, проверьте их поведение на примерах входных данных:

```
isUpper, isLower, isAlpha, isDigit, toUpper, toLower
ord, chr
```

Запросите в интерпретаторе и объясните тип функций, проверьте их поведение на примерах входных данных, и напишите свою реализацию каждой функции под другим именем:

```
digitToInt
intToDigit
```

Напишите функции `hexToDecMy :: String -> Integer`, `decToHexMy :: Integer -> String`, преобразующие десятичное число в 16-ричное и наоборот. Обратите внимание, что стандартные функции `digitToInt` и `intToDigit` работают и с десятеричными цифрами тоже.

> Напишите функции `romanToArabMy :: String -> Integer`, `arabToRomanMy :: Integer -> String`, преобразующие десятичное число в обычной записи в запись в римских цифрах и наоборот.

Простейшие кортежные функции

Запросите в интерпретаторе и объясните тип функций, проверьте их поведение на примерах входных данных:

```
fst, snd
```

Запросите в интерпретаторе и объясните тип функций, проверьте их поведение на примерах входных данных, и напишите свою реализацию каждой функции под другим именем:

```
zip, unzip
```

Теоретико-множественные операции

Напишите функцию `nub :: Eq a => [a] -> [a]`, удаляющую дубликаты из списка.

Напишите функцию `delete :: Eq a => a -> [a] -> [a]`, удаляющую первое вхождение заданного элемента из списка.

Напишите функцию `union :: Eq a => [a] -> [a] -> [a]`, объединяющую два списка, как если бы это были множества.

Напишите функцию `(\\) :: Eq a => [a] -> [a] -> [a]`, вычитающую из первого списка второй, как если бы это были множества.

Напишите функцию `intersect :: Eq a => [a] -> [a] -> [a]`, находящую пересечение двух списков, как если бы это были множества.

> Булеаном множества называется множество всех подмножеств этого множества. Напишите функцию `powersetMy :: [a] -> [[a]]`, находящую булеан заданного списка, как если бы это было множество.
>
> Напишите функцию `complementsMy :: [a] -> [([a],[a])]`, находящую множество разбиений множества на подмножество и его дополнение до исходного множества.

Сортировка

Напишите функции `sort :: Ord a => [a] -> [a]`, `insert :: Ord a => a -> [a] -> [a]`, соответственно, сортирующие список по возрастанию и вставляющие элемент в отсортированный список.

Напишите функцию `countCharsMy :: String -> [(Char,Int)]`, которая подсчитывает количество вхождений каждого символа в строку и выводит список кортежей, отсортированный по убыванию второго элемента кортежа. Например, `countChars "hello" = [('l',2), ('h',1), ('e',1), ('o',1)]`.

Вспомогательные функции

Запросите в интерпретаторе и объясните тип функций, проверьте их поведение на примерах входных данных:

```
show, read, error, undefined

lines, unlines
words, unwords
```

Отладка

Подключите к какому-нибудь из своих модулей модуль Trace. Запросите в интерпретаторе, объясните тип и проверьте поведение на примерах входных данных для функции `trace`. Используйте `trace` в какой-нибудь из ваших рекурсивных функций и понаблюдайте за результатами.

Лабораторная работа 3

Списочные функции высших порядков

Запросите в интерпретаторе и объясните тип функций, проверьте их поведение на примерах входных данных, и напишите свою реализацию каждой функции под другим именем:

```
map, filter, any, all
```

Запросите в интерпретаторе и объясните тип функций, проверьте их поведение на примерах входных данных:

```
zipWith, (.), ($), flip, id, const
curry, uncurry
splitAt, takeWhile, dropWhile
span, break, until
concatMap

foldl, foldl1
scanl, scanl1
foldr, foldr1
scanr, scanr1
```

Напишите функцию `String -> String`, удаляющую из строки все гласные буквы с помощью функций `filter` и `elem`.

Напишите функцию `String -> String`, удаляющую из строки все согласные буквы с помощью функций `filter` и `elem`.

Напишите функцию `String -> String`, удаляющую из строки все цифры с помощью функций `filter` и `elem`.

Напишите функцию `[String] -> String`, берущую список строк и возвращающую строку, состоящую из первых букв каждой строки с помощью функций `map` и `head`.

Напишите функцию `[[a]] -> [a]`, берущую список списков и возвращающую список из последних элементов подсписков с помощью функций `map` и `last`. Проверьте на примере строк.

Напишите функцию `[[a]] -> Int -> [a]`, берущую список списков и возвращающую список из N-х элементов подсписков с помощью функций `map` и `(!!)`. Проверьте на примере строк.

Напишите функцию `[[a]] -> [[a]]`, берущую список списков и обращающая исходный список и все его подсписки с помощью функций `map` и `reverse`.

> Напишите функцию `mapIfMy :: (a -> Bool) -> (a -> b) -> (a -> b) -> [a] -> [b]`, берущую условие `cond`, функции `f1` и `f2` и список `xs`, и обрабатывающую, подобно map список `xs`, применяя к очередному элементу функцию `f1`, если условие `cond` на этом элементе равно `True`, и применяя функцию `f2` в обратном случае.

Напишите функцию `composeAllMy :: [a -> a] -> (a -> a)`, берущую список функций и возвращающую функцию, являющуюся их последовательной композицией.

Напишите функцию `applyIterateMy :: [a -> a] -> a -> a`, берущую список функций и применяющую эти функции последовательно к исходному значению, также переданному в функцию. Объясните разницу между этой функцией и предыдущей.

Напишите функцию `partition :: (a -> Bool) -> [a] -> ([a], [a])`, берущую предикат и список, и возвращающую кортеж из списков элементов, на которых предикат вернул `True` и `False` соответственно.

Напишите функцию `[a -> Bool] -> a -> Int`, берущую список предикатов и элемент, и находящую, сколько предикатов на этом элементе возвращают `True`.

Напишите функцию `findIndices :: (a -> Bool) -> [a] -> [Int]`, находящую индексы тех элементов в списке, на которых условие возвращает True.

Напишите функцию `sortBy :: (a -> a -> Ordering) -> [a] -> [a]`, сортирующую элементы в списке по переданной сортировочной функции.

Напишите функцию `on :: (b -> b -> c) -> (a -> b) -> a -> a -> c`, которая берет бинарную функцию `fb` и унарную функцию `fa`, и возвращает бинарную функцию. Например, применение функции `on` к умножению и какой-то функции `f` может выглядеть так: `(*) `on` f` должно возвращать функцию `\x y -> f x * f y`.
Примените функцию `on` совместно с `sortBy`, например: `sortBy (compare `on` fst) some_list_of_tuples`, приведите другие примеры.

Напишите функцию `filterMapAndMy :: [a -> Bool] -> [a] -> [a]`, которая выбирает только те элементы из списка, на которых все функции из списка функций возвращают `True`. Например, `filterMapAndMy [even, >5, <10] [1..20] = [6,8]`.

Напишите функцию `filterMapOrMy :: [a -> Bool] -> [a] -> [a]`, которая выбирает только те элементы из списка, на которых хотя бы одна функция из списка функций возвращает `True`. Например, `filterMapOrMy [even, >5] [1..10] = [2,4,6,7,8,9,10]`.

Напишите функцию `sumEqMy :: [[Int]] -> [Int]`, которая по списку списков чисел строит список чисел, получающийся при сложении всех списков выписанных один под другим "столбиком" с выравниванием по первому элементу каждого списка.
например, `sumEqMy [[1],[2,2],[3,3,3,3]] = [6,5,5,3,3]`.

Напишите функции `mapAccumLMy (или mapAccumRMy) :: (acc -> x -> (acc, y)) -> acc -> [x] -> (acc, [y])`, которая ведет себя как комбинация `map` и `foldl (или foldr)`: если функция map применяет к каждому элементу функцию `(x -> y)`, то эта должна применять функцию `(acc -> x -> (acc, y))`, передавая от элемента к элементу слева направо (справа налево) некоторое состояние `acc`. Напишите пример применения. Сравните со стандартными функциями `scanl` и `scanr`.

Напишите функцию `segregateFMy :: [a -> Bool] -> a -> ([a -> Bool] , [a -> Bool])`, которая берет список функций и определенное значение и возвращает кортеж из двух списков функций. В первом элементе кортежа должен оказаться список функций, которые возвращают `True`, а во втором элементе кортежа – список тех функций, которые возвращают `False`. Например, `segregateFMy [odd,even,(>5),(<4),(>1)] 6 = ([even,(>5),(>1)] , [odd,(<4)])`.

Арифметические последовательности

Запросите в интерпретаторе и объясните тип выражений и результат их вычисления:

```
[1..10]
[2,4..11]
[1..]
[1,5..]
['a'..'z']
```

Генераторы списков

Вычислите выражения и объясните результат их вычисления:

```
squares [1..10] where squares xs = [x^2 | x <- xs]
[x^2 | x <- [1..10], odd x]
[(x,y) | x <- [1..10], y <- ['a'..'d']]
```

Напишите с помощью генераторов списков функцию `[Integer] -> [(Integer, Integer)]`, которая из списка `[Integer]` формирует список всевозможных пар чисел из этого списка, таких что первое число меньше второго. Напишите аналогичную функцию без использования генераторов списков.

Лабораторная работа 4

Бесконечные списки

Запросите в интерпретаторе и объясните тип функций, проверьте их поведение на примерах входных данных, и напишите свою реализацию каждой функции под другим именем:

```
iterate, repeat, cycle
```

Научитесь работать с бесконечными арифметическими последовательностями вида `[N..]` и `[N,K..]`, а так же строить на их основе бесконечные списки.

Напишите с помощью функции `iterate` функцию `pseudoRandomMy :: Integer -> [Integer]`, строящую бесконечный список псевдослучайных чисел в соответствии с линейным конгруэнтным методом.

Напишите функцию, строящую бесконечный список символов, получающихся последовательной записью натуральных чисел: `['1','2','3','4','5','6','7','8','9','1','0','1','1','1','2',…]`, и проверьте ее, выведя первые сто символов, а также тысячный символ. Используйте бесконечный список `[1..]` и функции `map` и `show`.

Напишите функцию, строящую бесконечный список символов, получающихся конкатенацией текстовых записей степеней десятки `['1','0','1','0','0','1','0','0','0',…]`, и проверьте ее, выведя первые сто символов, а также тысячный символ. Используйте функции `iterate` и `show`.

> Напишите функцию, строящую бесконечный список чисел, у которых сумма чисел равна их произведению и вывести первые 10 его элементов, а так же найти максимальный номер, который ваша реализация и ваша система позволяет найти за разумное время.

Напишите функцию, строящую бесконечный список степеней двойки, у которых сумма чисел нечетная и вывести 10 первых элементов, а так же найти максимальный номер, который ваша реализация и ваша система позволяет найти за разумное время.

Напишите функцию, строящую бесконечный список подсписков чисел: в первом подсписке будут степени единицы, во втором степени двойки, в третьем - тройки и так далее: `[[1,1,1,…],[2,4,8,…],[3,9,27,…],…]`.

Ввод-вывод

Запросите в интерпретаторе и объясните тип функций, проверьте их поведение на примерах входных данных:

```
putChar, putStr, putStrLn, print
getChar, getLine
readFile, writeFile
interact
```

Напишите функцию, которая читает входной текстовый файл и проверяет, правильно ли в тексте расставлены скобки (для каждой открывающей есть своя правильная закрывающая).

Напишите функцию, которая читает входной текстовый файл и выводит в выходной файл пары (слово:число), где слово - есть каждое уникальное слово файла, а число - количество вхождений этого слова. Пары должны быть отсортированы по убыванию чисел. Постарайтесь использовать как можно больше уже написанных функций.

Напишите функцию, которая читает входной текстовый файл и выводит в выходной файл пары (буква:число), где буква - есть каждая возможная буква (возможно не встречающаяся даже в файле), а число - количество вхождений этой буквы. Пары должны быть отсортированы по убыванию чисел.

Напишите функцию, которая читает входной текстовый файл и выводит в выходной файл отсортированные по возрастанию все уникальные слова этого текста без учета регистра

Напишите функцию, которая читает входной текстовый файл и заданное слово, и выводит в выходной файл только те строки входного файла, где содержится это слово.

Нетривиальные функции

Напишите функцию `intersperse :: a -> [a] -> [a]`, вставляющую заданный элемент между всеми элементами заданного списка. Например, `intersperse ',' "abcde" == "a,b,c,d,e"`.

Напишите функцию `explodeMy :: a -> [a] -> [[a]]`, разбивающую заданный список элементов на подсписки в тех местах, где встречается заданный элемент. Например, `explode ',' "a,b,c,d,e" == ["a","b","c","d","e"]`.

Напишите функцию `explodeBy :: (a -> Bool) -> [a] -> [([a],[a])]`, разбивающую заданный список элементов по заданному условию на кортежи из двух подсписков подряд идущих элементов. В каждом кортеже в первом подсписке содержатся подряд идущие элементы, на которых условие возвращает `True`, а во втором – `False`. Например, `explodeBy (`elem` ".!?")`

`"Something happened... Finally!" == [("Something happened","..."),(" Finally","!")]`.

Напишите функцию `transpose :: [[a]] -> [[a]]`, которая берет список списков и транспонирует столбцы и строки. Например, `transpose [[1,2,3],[4,5,6]] == [[1,4],[2,5],[3,6]]`.

Напишите функцию `permutations :: [a] -> [[a]]`, находящую все возможные перестановки заданной последовательности. Например, `permutations "abc" == ["abc","bac","cba","bca","cab","acb"]`.

Напишите функцию `group :: Eq a => [a] -> [[a]]`, группирующую подряд идущие одинаковые элементы в отдельный подсписок. Например, `group "Mississippi" = ["M","i","ss","i","ss","i","pp","i"]`.

Напишите функцию `groupBy :: (a -> a -> Bool) -> [a] -> [[a]]`, делающую то же самое, что и `group`, но по задаваемой операции сравнения.

Напишите функцию `inits :: [a] -> [[a]]`, находящую все префиксы заданного списка. Например, `inits "abc" == ["","a","ab","abc"]`.

Напишите функцию `tails :: [a] -> [[a]]`, находящую все суффиксы заданного списка. Например, `tails "abc" == ["abc","bc","c",""]`.

Напишите функцию `infixesMy :: [a] -> [[a]]`, находящую все непрерывные подсписки заданного списка. Например, `infixesMy "abc" == ["","a","b","c","ab","bc","abc"]`.

Напишите функцию `subsequences :: [a] -> [[a]]`, находящую все возможные подпоследовательности заданной последовательности. Например, `subsequences "abc" == ["","a","b","ab","c","ac","bc","abc"]`.

Напишите функцию `isPrefixOf :: Eq a => [a] -> [a] -> Bool`, проверяющую, является ли одна строка префиксом другой.

Напишите функцию `isSuffixOf :: Eq a => [a] -> [a] -> Bool`, проверяющую, является ли одна строка суффиксом другой.

Напишите функцию `isInfixOf :: Eq a => [a] -> [a] -> Bool`, проверяющую, является ли одна строка инфиксом (подстрокой) другой.

Напишите функцию `isSubsequenceOfMy :: Eq a => [a] -> [a] -> Bool`, проверяющую, является ли одна строка подпоследовательностью другой.

Напишите функцию `cumSumPostfixMy :: Num a => [a] -> [a]`, которая вычисляет кумулятивные суммы всех постфиксов списка, составляя из списка `[x1,x2,…xN]` список `[x1+…+xN,x2+…+xN,…,x(N-1)+xN,xN]`. Например, `cumSumPostfixMy [1,2,2,4] == [9,8,6,4]`.

Напишите функцию `cumSumPrefixMy :: Num a => [a] -> [a]`, которая вычисляет кумулятивные суммы всех префиксов списка. Например, `cumSumPrefixMy [1,2,2,4] == [1,3,5,9]`.

Напишите функцию `diffMy :: [Int] -> [Int]`, которая вычисляет разницы между парами подряд идущих чисел, составляя из списка `[x1,x2,…xN]` список `[x2-x1,x3-x2,…,xN-x(N-1)]`.

Говорят, это называется численным дифференцированием табулированной функции. Например, `diffMy [1,2,2,4] == [1,0,2]`. Попробуйте использовать функцию `zipWith` и `(-)`.

> Дана строка вида `"2*3 5 3*2 0 4*2"`, в которой присутствуют как отдельные числа, так и конструкции типа `"N*K"`. Напишите функцию `String -> String`, строящую новую строку по заданной с помощью следующих преобразований: отдельные числа переходят в строку-результат как есть, а конструкции типа `"N*K"` преобразуются в строку `"N N N ... N"`, где `N` повторяется `K` раз. Например, `foo "2*3 5 3*2 0 4*2" == "2 2 2 5 3 3 0 4 4"`.
>
> Напишите обратную функцию к предыдущей, используя `group`.

Напишите функцию `[Int] -> [Int]`, находящую для каждого числа списка, сколько чисел имеется строго меньше него и выводящую в результирующий список на соответствующей позиции это количество. Например, `foo [1,5,3,4,3] = [0,4,1,3,1]`.

Лабораторная работа 5

Простые числа и факторизация

Напишите функцию `primesMy :: [Integer]`, строящую бесконечный список простых чисел. Найдите максимально большое простое число, которое ваша система позволяет найти за разумное время.

Напишите функцию `factorizeMy :: Integer -> [Integer]`, которая разлагает заданное число на простые множители, возвращая список степеней при простых множителях. Например, раз `350 ==` $2^1 * 3^0 * 5^2 * 7^1$, то `factorizeMy 350 == [1,0,2,1]`.

Напишите функцию `defactorizeMy :: [Integer] -> Integer`, которая собирает обратно число, разложенное на простые множители. Например, `defactorizeMy [1,0,2,1] == 350`.

> Напишите реализацию функций gcd и lcm с помощью `factorizeMy` и `defactorizeMy`.

Деревья

Определите тип данных дерева: `data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Show`, запросите и объясните тип функций-конструкторов-данных `Leaf` и `Branch`, постройте с помощью этих функций простейшие деревья. Объясните, какие деревья можно представить такой структурой данных, а какие нельзя.

Определите тип данных дерева: `data Tree a = Empty | Branches a [Tree a] deriving Show`, запросите и объясните тип функций-конструкторов-данных `Empty` и `Branches`, постройте с помощью этих функций простейшие деревья. Объясните, какие деревья можно представить такой структурой данных, а какие нельзя.

Напишите функцию `fringeMy :: Tree a -> [a]`, возвращающую в произвольном порядке все листья дерева.

Напишите функцию `mapTreeMy :: (a -> b) -> Tree a -> Tree b`, применяющую к каждому элементу дерева заданную функцию без изменения структуры самого дерева.

Напишите функцию `depthMy :: Tree a -> Integer`, находящую глубину дерева - количество поддеревьев на пути к самой глубокой вершине.

Напишите функцию `dfsMy :: Tree a -> [a]`, возвращающую все вершины дерева в порядке обхода дерева в глубину.

> Напишите функцию `bfsMy :: Tree a -> [a]`, возвращающую все вершины дерева в порядке обхода дерева в ширину.
>
> Напишите функцию `showTreeMy :: Tree a -> String`, возвращающую текстовое представление дерева в виде последовательности строк с отступами (примерно как команда tree в командной строке MS Windows).

Определите тип данных для бинарного поискового дерева: `data Ord a => SearchTree a = Empty | Branches a (SearchTree a) (SearchTree a) deriving (Show, Eq)`, запросите и объясните тип функций-конструкторов-данных `Empty` и `Branches`, постройте с помощью этих функций простейшие деревья.

Напишите функцию `elemTreeMy :: Ord a => SearchTree a -> Bool`, проверяющую правильность бинарного поискового дерева.

Напишите функцию `checkTreeMy :: Ord a, Eq a => a -> SearchTree a -> Bool`, проверяющую, есть ли конкретный элемент в дереве.

Напишите функцию `putValMy :: Ord a => a -> SearchTree a -> SearchTree a`, добавляющую в правильное бинарное поисковое дерево очередной элемент в нужное место.

Напишите функцию `list2treeMy :: Ord a => [a] -> SearchTree a`, создающую из пустого дерева правильное поисковое дерево путем добавления в дерево последовательно всех элементов списка.

Напишите функцию `heightMy :: Ord a => SearchTree a -> Integer`, находящую глубину дерева.

> Напишите функцию `checkhMy :: Ord a => SearchTree a -> Bool`, проверяющую, является ли текущее бинарное поисковое дерево идеально сбалансированным.
>
> Напишите функцию `balanceMy :: Ord a => SearchTree a -> SearchTree a`, создающую из поискового дерева идеально сбалансированное дерево.
>
> Объясните, что вычисляет выражение `and $ map checkhMy $ map balanceMy $ map list2treeMy $ permutations "abcdefgh"`.

Деревья вычислений

Один из вариантов представления некоторого выражения - это бинарное дерево, в узлах которого - бинарные операции, а в листьях - значения. Нижеследующий код позволяет строить всевозможные деревья вычислений для заданного набора значений и четырех базовых операций.

```
module EvalTrees where
```

```
-- импортируем модуль для работы с рациональными числами
import Ratio

-- дерево вычислений (или значение - или операция над двумя деревьями)
data EvalTree = Val Integer | Oper Char EvalTree EvalTree    deriving Show

-- операции
ops = "+-*/"

-- результат вычисления дерева
eval :: EvalTree -> Rational
eval (Val x) = toRational x
eval (Oper '+' x y) = eval x + eval y
eval (Oper '-' x y) = eval x - eval y
eval (Oper '*' x y) = eval x * eval y
eval (Oper '/' x y)
                  | eval y == 0 = 0 -- вообще-то говоря неверно
                  | otherwise = eval x / eval y

-- множество разбиений множества на подмножества и дополнения
complements :: [a] -> [([a],[a])]
complements [] = [([],[])]
complements (x:xs) = concat $ map (inject x) (complements xs) where
      inject x (xs,ys) = [(x:xs,ys),(xs,x:ys)]

-- всевозможные деревья для заданного списка чисел
trees :: [Integer] -> [EvalTree]
trees [] = []
trees [x] = [Val x]
trees values = concat
            [allTreesWith op lvs rvs |
                  (lvs,rvs) <- complements values,
                  op <- ops,
                  not (null lvs), not (null rvs)
            ] where
      -- всевозможные деревья для операции и зафиксированных чисел слева и справа
      allTreesWith op lvs rvs = [Oper op lt gt | lt <- trees lvs, gt <- trees rvs]

showratio x | denominator x == 1     = show (numerator x)
showratio x | otherwise        = show (numerator x)++ "/"++ show (denominator x)

-- пример: вывести все деревья для 1,2,3 ?
-- trees [1,2,3]
-- а первое из них?
-- head $ trees [1,2,3]
-- а вычислить его?
-- eval $ head $ trees [1,2,3]
-- а в компактной форме?
-- showratio $ eval $ head $ trees [1,2,3]
```

Используя все цифры из множества `{1, 5, 6, 7}` ровно один раз, а так же скобки и арифметические операции `(+, -, *, /)` получить число `21`. Цифры должны использоваться отдельно, "слеплять" их (`12 + 34`) нельзя. Вот, например, какие числа (это только часть из всех возможных) можно получить из множества `{1, 2, 3, 4}`:

```
 8 = (4 * (1 + 3)) / 2
 14 = 4 * (3 + 1 / 2)
```

```
19 = 4 * (2 + 3)  1
36 = 3 * 4 * (2 + 1)
```

С помощью приведенных функций найдите выражение, с помощью которого можно получить число `21` из множества `{1, 5, 6, 7}`.

> Найдите выражения, не вычисляющиеся из-за деления на ноль. Возможно, придется модифицировать какие-то функции.

Дополнительные задания для самостоятельной работы

Задания с Project Euler

Задания взяты с сайта Project Euler (http://projecteuler.net/problems). Каждое задание требует решения определенной вычислительной задачи. Необходимо написать решение на языке Haskell. Можно зарегистрироваться на сайте и проверять там корректность ответов. Ограничений на решение не накладывается. Нумерация заданий соответствует задачам Project Euler. Задач там, конечно, намного больше – я выбрал только некоторые, чтобы было, с чего начать.

```
ID 1. Натуральные числа меньше 10, которые делятся на 3 или на 5 - это 3, 5, 6 и 9. Их
сумма равна 23. Найдите сумму всех чисел, делящихся на 3 или на 5, которые меньше
1000.

ID 2. В ряде Фибоначчи каждое следующее число равно двум предыдущим:
1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
Найдите сумму всех четных чисел Фибоначчи, не превышающих 4000000.

ID 3. Простые делители числа 13195 - это 5, 7, 13 and 29. А какой самый большой
простой делитель числа 600851475143?

ID 4. Числа-палиндромы - это те, что читаются одинаково как слева направо, так и
справа налево. Самый большой палиндром, получаемый произведением двух двузначных чисел
- это 9009 = 91 * 99. Найдите самый большой палиндром, получаемый произведением двух
трехзначных чисел.

ID 5. 2520 - это самое маленькое число из тех, что делятся на все числа от 1 до 10 без
остатка. А какое самое маленькое число делится без остатка на все числа от 1 до 20?

ID 6. Сумма квадратов первых 10 натуральных чисел:
1^2 + 2^2 + ... + 10^2 = 385
А квадрат суммы этих чисел:
(1 + 2 + ... + 10)^2 = 55^2 = 3025
Разница между квадратом суммы и суммой квадратов для первых 10 натуральных чисел равна
3025 - 385 = 2640.
Найдите подобную разницу для первых 100 натуральных чисел.

ID 7. Первые 6 простых чисел: 2, 3, 5, 7, 11, и 13. А чему равно 10001-е простое
число?

ID 8. Найдите самое большое произведение из 5 последовательно идущих цифр этого
1000-значного числа.
 7316717653133062491922511967442657474235534919493
 9698352031277450632623957831801698480186947885184
```

858615607891129494954595017379583319528532088055 11
125406987471585238630507156932909632952274430435 57
668966489504452445231617318564030987111217223831 13
622298934233803081353362766142828064444866452387 49
303589072962904915604407723907138105158593079608 66
701724271218839987979087922749219016997208880937 76
657273330010533678812202354218097512545405947522 43
525849077116705560136048395864467063244157221553 97
536978179778461740649551492908625693219784686224 82
839722413756570560574902614079729686524145351004 74
821663704844031998900088952434506585412275886668 81
164271714799244429282308634656748139191231628245 86
178664583591245665294765456828489128831426076900 42
242190226710556263211111093705442175069416589604 08
071984038509624554443629812309878799272442849091 88
845801561660979191338754992005240636899125607176 06
058861164671094050775410022569831552000559357297 25
716362695618826704282524836008232575304207529634 50

ID 9. Пифагорова тройка - это три такие натуральные числа a < b < c, что
a^2 + b^2 = c^2
Например, 3^2 + 4^2 = 9 + 16 = 25 = 5^2.
Найдите единственную Пифагорову тройку, такую что a + b + c = 1000 и вычислите произведение abc.

ID 10. Сумма простых чисел, меньших 10, равна 2 + 3 + 5 + 7 = 17. Найдите сумму всех простых чисел, меньших 2000000.

ID 11. В матрице 20 на 20, приведенной ниже, выделены 4 числа, идущих по диагонали. Их произведение равно 26 * 63 * 78 * 14 = 1788696.

08 02 22 97 38 15 00 40 00 75 04 05 07 78 52 12 50 77 91 08
49 49 99 40 17 81 18 57 60 87 17 40 98 43 69 48 04 56 62 00
81 49 31 73 55 79 14 29 93 71 40 67 53 88 30 03 49 13 36 65
52 70 95 23 04 60 11 42 69 24 68 56 01 32 56 71 37 02 36 91
22 31 16 71 51 67 63 89 41 92 36 54 22 40 40 28 66 33 13 80
24 47 32 60 99 03 45 02 44 75 33 53 78 36 84 20 35 17 12 50
32 98 81 28 64 23 67 10 26 38 40 67 59 54 70 66 18 38 64 70
67 26 20 68 02 62 12 20 95 63 94 39 63 08 40 91 66 49 94 21
24 55 58 05 66 73 99 26 97 17 78 78 96 83 14 88 34 89 63 72
21 36 23 09 75 00 76 44 20 45 35 14 00 61 33 97 34 31 33 95
78 17 53 28 22 75 31 67 15 94 03 80 04 62 16 14 09 53 56 92
16 39 05 42 96 35 31 47 55 58 88 24 00 17 54 24 36 29 85 57
86 56 00 48 35 71 89 07 05 44 44 37 44 60 21 58 51 54 17 58
19 80 81 68 05 94 47 69 28 73 92 13 86 52 17 77 04 89 55 40
04 52 08 83 97 35 99 16 07 97 57 32 16 26 26 79 33 27 98 66
88 36 68 87 57 62 20 72 03 46 33 67 46 55 12 32 63 93 53 69
04 42 16 73 38 25 39 11 24 94 72 18 08 46 29 32 40 62 76 36
20 69 36 41 72 30 23 88 34 62 99 69 82 67 59 85 74 04 36 16
20 73 35 29 78 31 90 01 74 31 49 71 48 86 81 16 23 57 05 54
01 70 54 71 83 51 54 69 16 92 33 48 61 43 52 01 89 19 67 48

А какое максимальное произведение четырех подряд идущих чисел (по горизонтали, по диагонали или по вертикали) можно получить в этой матрице?

ID 16. 2^15 = 32768, и сумма цифр этого числа равна 3 + 2 + 7 + 6 + 8 = 26. Чему равна сумма цифр числа 2^1000?

Приложения

Приложение 1. Простейший инструментарий

Установка WinHugs и начало работы

Для обучения языку вполне достаточно будет простого интерпретатора языка Haskell, который называется Hugs. Последнюю версию лучше всего брать здесь:

`http://cvs.haskell.org/Hugs/pages/downloading.htm`

На момент написания этого текста, последняя версия, доступная по приведенной выше ссылке, называется `WinHugs-Sep2006.exe`. Установка типична и проблем не вызывает.

Вообще говоря, интерпретатор Hugs не поддерживается разработчиками уже несколько лет (по крайней мере, не выходят обновления). Достоинством его является маленький размер и простота использования, поэтому для обучения я рекомендую пользоваться именно им.

Тем же, кто хочет с самого начала работать со "взрослыми" пакетами разработки на языке Haskell, прямая дорога к The Haskell Platform с его GHC (Glasgow Haskell Compiler) и набором стандартных библиотек и утилит (в том числе и очень похожей на Hugs маленькой интерпретируемой средой WinGHCi). Взять все необходимое можно здесь:

`http://www.haskell.org/platform/`

Интересующихся более полным обзором имеющегося инструментария для разработки на языке Haskell, отсылаю к отдельной книге [3].

Работа с интерпретатором WinHugs в интерактивном режиме

> Примечание: текст этого и нескольких следующих разделов в своей большей части представляет собой перевод частей документа "The Hugs 98 User Manual".

При запуске программа отображает главное окно программы с информацией о версии и приглашением к работе:

```
Type :? for help
Hugs>
```

Интерпретатор представляет собой диалоговую среду, в которой пользователь печатает предложение, а система это предложение вычисляет и выдает ответ.

Вообще говоря, Hugs – это что-то вроде очень сложного калькулятора. Интерпретатор просто вычисляет значение введенного пользователем выражения и возвращает результат этого вычисления:

```
Hugs> (2+3)*8
40
Hugs > sum [1..10]
55
```

```
Hugs >
```

Надпись `Hugs>` в начале первой, третьей и пятой строк – это приглашение к работе, отображаемое интерпретатором. Оно показывает, что система готова к вводу пользователем выражений. В первом случае пользователь ввел выражение `(2+3)*8`, которое было вычислено и результат `40` возвращен пользователю. В ответ на второе приглашение системы пользователь ввел выражение `sum [1..10]`. Запись `[1..10]` означает список целых чисел от `1` до `10`, а `sum` – это функция из стандартных модулей, которая возвращает сумму чисел в заданном списке. Таким образом, интерпретатор возвращает:

```
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10  →  55
```

На самом деле, чтобы посчитать эту сумму, можно было прямо так и набрать в строке ввода:

```
Hugs > 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
55
Hugs >
```

Однако в отличие от калькуляторов, Hugs не ограничивается работой с числами. Выражения могут включать значения многих типов: числа, логические значения, символы, строки, списки, функции и другие типы, в том числе и определенные пользователем. Например:

```
Hugs > (not True) || False
False
Hugs > reverse "Hugs is cool"
"looc si sguH"
Hugs > filter even [1..10]
[2, 4, 6, 8, 10]
Hugs > take 10 fibs where fibs = 0:1:zipWith (+) fibs (tail fibs)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
Hugs >
```

Однако в командной строке интерпретатора можно только вычислять значения выражений, но нельзя вводить новые определения: их нужно помещать в файлы и загружать (как это сделать, будет объяснено совсем скоро).

Определение `fibs` в последнем примере создается только при вычислении выражения и не сохраняется для последующего использования. К тому же выражение не может содержать переносов строк (хотя и может занимать больше, чем одну строку на экране).

Команды интерпретатору

Каждая вводимая пользователем строка расценивается Hugs как команда интерпретатору. Если вводимая строка представляет из себя выражение, то, как было показано раньше, она рассматривается как команда интерпретатору вычислить значение этого выражения и вернуть его пользователю. Кроме выражений пользователь может использовать набор заранее определенных команд, самой главной из которых является ":?".
При вводе такой команды Hugs вернет список команд, которые могут быть использованы, все они начинаются с двоеточия.

Некоторые команды приведены ниже. Самой важной и полезной командой здесь является команда ":t", возвращающая тип функции.

```
:l <filenames>   загрузить модули из файлов
:r               перезагрузить текущий модуль
:e <filename>    редактировать модуль
:e               редактировать текущий модуль
<expr>           вычислить значение выражения
:t <expr>        вывести тип выражения
:?               вывести список доступных команд
:n [pat]         вывести функции с именем, подходящим под шаблон [pat]
:q               выйти из интерпретатора
```

Работа с модулями

Функции типа `sum`, `(+)`, `take` и прочие, использованные в приведенных выше примерах являются стандартными и определены в стандартных модулях. В этом загружаемом по умолчанию модуле определено огромное количество полезных функций; но, разумеется, чтобы сделать что-то полезное, необходимо уметь создавать свои собственные функции. Лучше всего при этом создавать их в собственном модуле (или модулях) и загружать в Hugs. Модуль – это всего-лишь набор определений, хранимых в каком-то файле. Разумеется, в каждом модуле может храниться множество определений функций.
Например, создадим следующий модуль и сохраним его в файле fact.hs. (Модули Hugs обычно имеют расширение ".hs", а имя модуля должно совпадать с именем файла и начинаться с прописного символа.

```
module Fact where
fact  :: Integer -> Integer
fact n = product [1..n]
```

Функция `product`, использованная здесь, является стандартной и определена в стандартном модуле; она используется для вычисления произведения всех числен из списка, так же как функция `sum` используется для вычисления суммы списка чисел.
Таким образом, приведенные выше строки определяют функцию, которая берет число, обозначаемое `n` и вычисляет его факториал. В стандартной математической записи `fact n = n!` определяется следующим образом:

```
n! = 1 * 2 * ... * (n-1) * n
```

Когда вы привыкнете к используемой Hugs нотации, то заметите, что определения функций в Hugs очень часто близки к неформальным математическим определениям. Еще одно часто используемое математическое определение факториала выглядит следующим образом:

```
f(0) = 1
f(n) = n * f(n-1)
```

Совершенно аналогичным образом можно было определить функцию в Hugs. Для этого добавим три строчки в конец определенного ранее модуля Fact:

```
f  :: Integer -> Integer
f 0 = 1
f n = n * f (n-1)
```

Для того, чтобы можно было использовать введенные определения в среде интерпретатора, модуль необходимо загрузить. Для этого используется команда :load (или :l , потому что все команды можно сокращать до первой буквы):

```
Hugs> :l fact.hs
Fact>
```

То же самое можно было выполнить с помощью кнопки в командном окне интерпретатора, если файл находится не в каталоге программы. Заметьте, что к модулю Hugs, используемому интерпретатором, теперь добавился файл Fact.hs. Приглашение к работе теперь тоже изменилось, и оно говорит о том, что теперь мы можем использовать определенные в модуле функции:

```
Fact> fact 6
720
Fact> fact 6 + fact 7
5760
Fact> fact 7 `div` fact 6
7
Fact>
```

В одном модуле можно подключить функциональность из другого модуля. Для этого нужно написать в модуле, например:

```
import Char
```

Теперь, после того, как весь основной инструментарий готов, пора приступать к изучению собственно теории и практики функционального программирования на языке Haskell.

Приложение 2. Список рекомендуемой литературы и электронных ресурсов

1. "Функциональное программирование на языке Haskell", Роман Душкин, ДМК Пресс, 2007.
2. "Изучай Haskell во имя добра!", Миран Липовача, ДМК Пресс, 2012.
3. "Практика работы на языке Haskell", Роман Душкин, ДМК Пресс, 2010.
4. "The Haskell School of Expression", Paul Hudak, Cambridge University Press, 2000.
5. http://www.haskell.org/
6. http://www.haskell.org/hoogle/