Оглавление

Глава 1

Функции высшего порядка

Функцией высшего порядка называют функцию, которая может принимать на вход функции или возвращать функции в качестве результата. За счёт частичного применения в Haskell все функции, которые принимают более одного аргумента, являются функциями высшего порядка.

В этой главе мы подробно обсудим способы составления функций, недаром Haskell – функциональный язык. В Haskell функции являются очень гибким объектом, они позволяют выделять сложные способы комбинирования значений. Часто за счёт развитых средств составления новых функций в Haskell пользователь определяет лишь базовые функции, получая остальные “на лету” применением двух-трёх операций, это выглядит примерно как `(2+3)*5`, где вместо чисел стоят базовые функции, а операции `+` и `*` составляют новые функции из простейших.

1.1 Обобщённые функции

В этом разделе мы познакомимся с несколькими функциями, которые принимают одни функции и составляют по ним другие. Эти функции используются в Haskell очень часто. Все они живут в модуле `Data.Function`. Модуль `Prelude` экспортирует их из этого модуля.

1.1.1 Функция тождества

Начнём с самой простой функции. Это функция `id`. Она ничего не делает с аргументом, просто возвращает его:

```
id :: a -> a
id x = x
```

Зачем нам может понадобиться такая функция? Сама по себе она бесполезна. Она приобретает ценность при совместном использовании с другими функциями, поэтому пока мы не будем приводить примеров.

1.1.2 Константная функция

Следующая функция `const` принимает значение и возвращает постоянную функцию. Эта функция будет возвращать константу для любого переданного в неё значения:

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

Функция `const` является конструктором постоянных функций, так например мы получаем пятёрки на любой аргумент:

```
Prelude> let onlyFive = const 5
Prelude> :t onlyFive
onlyFive :: b -> Integer
Prelude> onlyFive "Hi"
5
Prelude> onlyFive (1,2,3)
5
Prelude> map onlyFive "abracadabra"
[5,5,5,5,5,5,5,5,5,5,5]
```

С её помощью мы можем легко построить и постоянную функцию двух аргументов:

```
const2 a = const (const a)
```

Вспомним определение для &&:

```
(&&) :: Bool -> Bool -> Bool
(&&) True   x  = x
(&&) False  _  = False
```

С помощью функций id и const мы можем сократить число аргументов и уравнений:

```
(&&) :: Bool -> Bool -> Bool
(&&) a = if a then id else (const False)
```

Также мы можем определить и логическое “или”:

```
(||) :: Bool -> Bool -> Bool
(||) a = if a then (const True) else id
```

1.1.3 Функция композиции

Функция композиции принимает две функции и составляет из них последовательное применение функций:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g = \x -> f (g x)
```

Это очень полезная функция. Она позволяет нанизывать функции друг на друга. Мы перехватываем выход второй функции, сразу подставляем его в первую и возвращаем её выход в качестве результата. Например перевернём список символов и затем сделаем все буквы заглавными:

```
Prelude> :m +Data.Char
Prelude Data.Char> (map toUpper . reverse) ”abracadabra”
”ARBADACARBA”
```

Приведём пример посложнее:

```
add :: Nat -> Nat -> Nat
add  a  Zero     = a
add  a  (Succ b) = Succ (add a b)
```

Если мы определим функцию свёртки для Nat, которая будет заменять в значении типа Nat конструкторы на соответствующие по типу функции:

```
foldNat :: a -> (a -> a) -> Nat -> a
foldNat zero succ Zero     = zero
foldNat zero succ (Succ b) = succ (foldNat zero succ b)
```

То мы можем переписать с помощью функции композиции эту функцию так:

```
add :: Nat -> Nat -> Nat
add = foldNat  id  (Succ . )
```

Куда делись аргументы? Они выражаются через функции id и (.). Поведение этой функции лучше проиллюстрировать на примере. Пусть у нас есть два числа типа Nat:

```
two     = Succ (Succ Zero)
three   = Succ (Succ (Succ Zero))
```

Вычислим

```
add two three
```

Вспомним о частичном применении:

```
    add two three
=>  (add two) three
=>  (foldNat id (Succ . ) (Succ (Succ Zero))) three
```

Теперь функция свёртки заменит все конструкторы `Succ` на `(Succ . )`, а конструкторы `Zero` на `id`:

```
=>  ((Succ . ) ((Succ . ) id)) three
```

Что это за монстр?

```
((Succ . ) ((Succ . ) id))
```

Функция `(Succ . )` это левое сечение операции `(.)`. Эта функция, которая принимает функции и возвращает функции. Она принимает функцию и навешивает на её выход конструктор `Succ`. Давайте упростим это большое выражение с помощью определений функций `(.)` и `id`:

```
    ((Succ . ) ((Succ . ) id))
=>   (Succ . ) (\x -> Succ (id x))
=>   (Succ . ) (\x -> Succ x)
=>   \x -> Succ (Succ x)
```

Теперь нам осталось применить к этой функции наше второе значение:

```
    (\x -> Succ (Succ x)) three
=>  Succ (Succ three)
=>  Succ (Succ (Succ (Succ x)))
```

Так мы получили, что и ожидалось от сложения. За каждый конструктор `Succ` в первом аргументе мы добавляем применение `Succ` к результату, а вместо `Zero` протаскиваем через `id` второй аргумент.

1.1.4 Аналогия с числами

С помощью функции композиции мы можем нанизывать друг на друга списки функций. Попробуем в интерпретаторе:

```
Prelude> let f = foldr (.) id [sin, cos, sin, cos, exp, (+1), tan]
Prelude> f 2
0.6330525927559899
Prelude> f 15
0.7978497904127007
```

Функция `foldr` заменит в списке каждый конструктор `(:)` на функцию композиции, а пустой список на функцию `id`. В результате получается композиция из всех функций в списке.

Это очень похоже на сложение или умножение чисел в списке. При этом в качестве нуля (для сложения) или единицы (для умножения) мы используем функцию `id`. Мы пользуемся тем, что по определению для любой функции `f` выполнены тождества:

```
f  . id  ==  f
id . f   ==  f
```

Поэтому мы можем использовать `id` в качестве накопителя результата композиции, как в случае:

```
Prelude> foldr (*) 1 [1,2,3,4]
24
```

Если сравнить `(.)` с умножением, то `id` похоже на единицу, а `(const a)` на ноль. В самом деле для любой функции `f` и любого значения `a` выполнено тождество:

```
const a  .  f  == const a
```

Мы словно умножаем функцию на ноль, делая её вычисление бессмысленным.

1.1.5 Функция перестановки

Функция перестановки `flip` принимает функцию двух аргументов и меняет аргументы местами:

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

К примеру:

```
Prelude> foldr (-) 0 [1,2,3,4]
-2
Prelude> foldr (flip (-)) 0 [1,2,3,4]
-10
```

Иногда это бывает полезно.

1.1.6 Функция on

Функция on (от англ.~на) перед применением бинарной функции пропускает аргументы через унарную функцию:

```
on :: (b -> b -> c) -> (a -> b) -> a -> a -> c
(.*.) `on` f = \x y -> f x .*. f y
```

Она часто используется в сочетании с функцией `sortBy` из модуля `Data.List`. Эта функция имеет тип:

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

Она сортирует элементы списка согласно некоторой функции упорядочивания `f :: (a -> a -> Ordering)`. С помощью функции on мы можем легко составить такую функцию на лету:

```
let xs = [(3, "John"), (2, "Jack"), (34, "Jim"), (100, "Jenny"), (-3, "Josh")]
Prelude> :m +Data.List Data.Function
Prelude Data.List Data.Function>
Prelude Data.List Data.Function> sortBy (compare `on` fst) xs
[(-3,"Josh"),(2,"Jack"),(3,"John"),(34,"Jim"),(100,"Jenny")]
Prelude Data.List Data.Function> map fst (sortBy (compare `on` fst) xs)
[-3,2,3,34,100]
Prelude Data.List Data.Function> map snd (sortBy (compare `on` fst) xs)
["Josh","Jack","John","Jim","Jenny"]
```

Мы импортировали в интерпретатор модуль `Data.List` для функции `sortBy` а также модуль `Data.Function` для функции on. Они не импортируются модулем `Prelude`.

Выражением `(compare `on` fst)` мы составили функцию

```
\a b -> compare (fst a) (fst b)

fst = \(a, b) -> a
```

Тем самым ввели функцию упорядочивания на парах, которая будет сравнивать пары по первому элементу. Отметим, что аналогичного эффекта можно добиться с помощью функции `comparing` из модуля `Data.Ord`.

1.1.7 Функция применения

Ещё одной очень полезной функцией является функция применения `($)`. Посмотрим на её определение:

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

На первый взгляд её определение может показаться бессмысленным. Зачем нам специальный знак для применения, если у нас уже есть пробел? Для ответа на этот вопрос нам придётся познакомиться с приоритетом инфиксных операций.

1.2 Приоритет инфиксных операций

В Haskell очень часто используются бинарные операции для составления функций “на лету”. В этом помогает и частичное применение, мы можем в одном выражении применить к функции часть аргументов, построить из неё новую функцию с помощью какой-нибудь такой бинарной операции и всё это передать в другую функцию!

Для сокращения числа скобок нам понадобится разобраться в понятии приоритета операции. Так например в выражении

```
> 2 + 3 * 10
32
```

Мы полагаем, что умножение имеет больший приоритет чем сложение и со скобками это выражение будет выглядеть так:

```
> 2 + (3 * 10)
32
```

Фраза “больший приоритет” означает: сначала умножение потом сложение. Мы всегда можем изменить поведение по умолчанию с помощью скобок:

```
> (2 + 3) * 10
50
```

В Haskell приоритет функций складывается из двух понятий: старшинство и ассоциативность. Старшинство определяется числами, они могут быть от 0 до 9. Чем больше это число, тем выше приоритет функций.

Старшинство используется вычислителем для группировки разных операций, например `(+)` имеет старшинство 6, а `(*)` имеет старшинство 7. Поэтому интерпретатор сначала ставит скобки вокруг выражения с `(*)`, а затем вокруг `(+)`. Считается, что обычное префиксное применение имеет высший приоритет 10. Нельзя задать приоритет выше применения, это значит, что операция “пробел” будет всегда выполняться первой.

Ассоциативность используется для группировки одинаковых операций, например мы видим:

```
1+2+3+4
```

Как нам быть? Мы можем группировать скобки слева направо:

```
((1+2)+3)+4
```

Или справа налево:

```
1+(2+(3+4))
```

Ответ на этот вопрос даёт ассоциативность, она бывает левая и правая. Например операции `(+)` `(-)` и (*) являются лево-ассоциативными, а операция возведения в степень `(^)` является право-ассоциативной.

```
1 + 2 + 3 == (1 + 2) + 3
1 ^ 2 ^ 3 ==  1 ^ (2 ^ 3)
```

Приоритет функции можно узнать в интерпретаторе с помощью команды `:i`:

```
*FunNat> :m Prelude
Prelude> :i (+)
class (Eq a, Show a) => Num a where
  (+) :: a -> a -> a
  ...
          -- Defined in GHC.Num
infixl 6 +
Prelude> :i (*)
class (Eq a, Show a) => Num a where
  ...
  (*) :: a -> a -> a
  ...
          -- Defined in GHC.Num
infixl 7 *
Prelude> :i (^)
(^) :: (Num a, Integral b) => a -> b -> a         -- Defined in GHC.Real
infixr 8 ^
```

Приоритет указывается в строчках `infixl 6 +` и `infixl 7 *`. Цифра указывает на старшинство операции, а суффикс `l` (от англ.~left – левая) или `r` (от англ.~right – правая) на ассоциативность.

Если мы создали свою функцию, мы можем определить для неё ассоциативность. Для этого мы пишем в коде:

```
module Fixity where

import Prelude(Num(..))

infixl 4 ***
infixl 5 +++
infixr 5 `neg`

(***) = (*)
(+++) = (+)
neg   = (-)
```

Мы ввели новые операции и поменяли старшинство операций сложения и умножения местами и изменили ассоциативность у вычитания. Проверим в интерпретаторе:

```
Prelude> :l Fixity
[1 of 1] Compiling Fixity           ( Fixity.hs, interpreted )
Ok, modules loaded: Fixity.
*Fixity> 1 + 2 * 3
7
*Fixity> 1 +++ 2 *** 3
9
*Fixity> 1 - 2 - 3
-4
*Fixity> 1 `neg` 2 `neg` 3
2
```

Посмотрим как это вычислялось:

```
1   +   2   *  3  ==   1   +   (2    *   3)
1  +++  2  *** 3  ==  (1  +++   2)  ***  3

1   -   2   -  3  ==  (1   -    2)   -   3
1 `neg` 2 `neg 3` ==   1 `neg` (2  `neg` 3)
```

Также в Haskell есть директива `infix` это тоже самое, что и `infixl`.

1.2.1 Приоритет функции композиции

Посмотрим на приоритет функции композиции:

```
Prelude> :i (.)
(.) :: (b -> c) -> (a -> b) -> a -> c         -- Defined in GHC.Base
infixr 9 .
```

Она имеет высший приоритет. Она очень часто используется при определении функции в бесточечном стиле. Такая функция похожа на конвейер функций:

```
fun a = fun1 a . fun2 (x1 + x2) . fun3 . (+x1)
```

1.2.2 Приоритет функции применения

Теперь посмотрим на полное определение функции применения:

```
infixr 0 $

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

Ответ на вопрос о полезности этой функции кроется в её приоритете. Ей назначен самый низкий приоритет. Она будет исполняться в последнюю очередь. Очень часто возникают ситуации вроде:

```
foldNat zero succ (Succ b) = succ (foldNat zero succ b)
```

С помощью функции применения мы можем переписать это определение так:

```
foldNat zero succ (Succ b) = succ $ foldNat zero succ b
```

Если бы мы написали без скобок:

```
... = succ foldNat zero succ b
```

То выражение было бы сгруппировано так:

```
... = (((succ foldNat) zero) succ) b
```

Но поскольку мы поставили барьер в виде операции (`$`) с низким приоритетом, группировка скобок произойдёт так:

```
... = (succ $ ((foldNat zero) succ) b)
```

Это как раз то, что нам нужно. Преимущество этого подхода проявляется особенно ярко если у нас несколько вложенных функций на конце выражения:

```
xs :: [Int]
xs = reverse $ map ((+1) . (*10)) $ filter even $ ns 40

ns :: Int -> [Int]
ns 0  = []
ns n  = n : ns (n - 1)
```

В списке `xs` мы сначала создаём в функции `ns` убывающий список чисел, затем оставляем лишь чётные, потом применяем два арифметических действия ко всем элементам списка, затем переворачиваем список.

Проверим работает ли это в интерпретаторе, заодно поупражняемся в композиционном стиле:

```
Prelude> let ns n = if (n == 0) then [] else n : ns (n - 1)
Prelude> let even x = 0 == mod x 2
Prelude> let xs = reverse $ map ((+1) . (*10)) $ filter even $ ns 20
Prelude> xs
[21,41,61,81,101,121,141,161,181,201]
```

Если бы не функция применения нам пришлось бы написать это выражение так:

```
xs = reverse (map ((+1) . (*10)) (filter even (ns 40)))
```

1.3 Функциональный калькулятор

Мне бы хотелось сделать акцент на одном из вступительных предложений этой главы:

> За счёт развитых средств составления новых функций в Haskell пользователь определяет лишь базовые функции, получая остальные “на лету” применением двух-трёх операций, это выглядит примерно как `(2+3)*5`, где вместо чисел стоят базовые функции, а операции `+` и `*` составляют новые функции из простейших.

Такие обобщённые функции как `id`, `const`, `(.)`, `map` `filter` позволяют очень легко комбинировать различные функции. Бесточечный стиль записи функций превращает функции в простые значения или значения-константы, которые можно подставлять в другие функции. В этом разделе мы немного потренируемся в перегрузке численных значений и превратим числа в функции, функции и в самом деле станут константами. Мы определим экземпляр `Num` для функций, которые возвращают числа. Смысл этих операций заключается в том, что теперь мы применяем обычные операции сложения умножения к функциям, аргумент которых совпадает по типу. Например для того чтобы умножить функции `\t -> t+2` и `\t -> t+3` мы составляем новую функцию `\t -> (t+2) * (t+3)`, которая получает на вход значение `t` применяет его к каждой из функций и затем умножает результаты:

```haskell
module FunNat where

import Prelude(Show(..), Eq(..), Num(..), error)

instance Show (t -> a) where
    show _ = error "Sorry, no show. It's just for Num"
instance Eq (t -> a) where
    (==) _ _ = error "Sorry, no Eq. It's just for Num"

instance Num a => Num (t -> a) where
    (+) = fun2 (+)
    (*) = fun2 (*)
    (-) = fun2 (-)

    abs    = fun1 abs
    signum = fun1 signum

    fromInteger = const . fromInteger

fun1 :: (a -> b) -> ((t -> a) -> (t -> b))
fun1 = (.)

fun2 :: (a -> b -> c) -> ((t -> a) -> (t -> b) -> (t -> c))
fun2 op a b = \t -> a t `op` b t
```

Функции `fun1` и `fun2` превращают функции, которые принимают значения, в функции, которые принимают другие функции.

Из-за контекста класса `Num` нам пришлось объявить два фиктивных экземпляра для классов `Show` и `Eq`. Загрузим модуль `FunNat` в интерпретатор и посмотрим что же у нас получилось:

```
Prelude> :l FunNat.hs
[1 of 1] Compiling FunNat           ( FunNat.hs, interpreted )
Ok, modules loaded: FunNat.
*FunNat> 2 2
2
*FunNat> 2 5
2
*FunNat> (2 + (+1)) 0
3
*FunNat> ((+2) * (+3)) 1
12
```

На первый взгляд кажется что выражение `2  2` не должно пройти проверку типов, ведь мы применяем значение к константе. Но на самом деле 2 это не константа, а значение `2 :: Num a => a` и подспудно к двойке применяется функция `fromInteger`. Поскольку в нашем модуле мы определили экземпляр `Num` для функций, второе число 2 было конкретизировано по умолчанию до `Integer`, а первое число 2 было конкретизировано до `Integer -> Integer`. Компилятор вывел из контекста, что под 2 мы понимаем функцию. Функция была создана с помощью метода `fromInteger`. Эта функция принимает любое значение и возвращает двойку.

Далее мы складываем и перемножаем функции словно это обычные значения. Что интересно мы можем составлять и такие выражения:

```
*FunNat> let f = ((+) - (*))
*FunNat> f 1 2
1
```

Как была вычислена эта функция? Мы определили экземпляр функций для значений типа `Num a => t -> a`. Если мы вспомним, что функция двух аргументов на самом деле является функцией одного аргумента: `Num a => t1 -> (t2 -> a)`, мы заметим, что тип `Num a => (t2 -> a)` принадлежит `Num`, теперь если мы обозначим его за `a'`, то мы получим тип `Num a' => t1 -> a'`, это совпадает с нашим исходным экземпляром.

Получается, что за счёт механизма частичного применения мы одним махом определили экземпляры `Num` для функций *любого* числа аргументов, которые возвращают значение типа `Num`.

Итак функция `f` имеет вид:

```haskell
\t1 t2 -> (t1 + t2) - (t1 * t2)
```

Подставим значения:

```
(\t1 t2 -> (t1 + t2) - (t1 * t2)) 1 2
(\t2 -> (1 + t2) - (1 * t2) 2
(1 + 2) - (1 * 2)
3 - 2
1
```

Теперь давайте составим несколько выражений с обобщёнными функциями. Для этого добавим в модуль `FunNat` директиву импорта функций из модуля `Data.Function`. Также добавим несколько основных функций для списков и класс `Ord`:

```
module FunNat where

import Prelude(Show(..), Eq(..), Ord(..), Num(..), error)

import Data.Function(id, const, (.), ($), flip, on)
import Prelude(map, foldr, filter, zip, zipWith)

...
```

и загрузим модуль в интерпретатор:

```
Prelude> :load FunNat
[1 of 1] Compiling FunNat           ( FunNat.hs, interpreted )
Ok, modules loaded: FunNat.
```

Составим функцию, которая принимает один аргумент, умножает его на два, вычитает 10 и берёт модуль числа.

```
*FunNat> let f = abs $ id * 2 - 10
*FunNat> f 2
6
*FunNat> f 10
10
```

Давайте посмотрим как была составлена эта функция:

```
    abs $ id * 2 - 10

=>  abs $ (id * 2) - 10                 -- приоритет умножения
=>  abs $ (\x -> x * \x -> 2) - 10      -- развернём id и 2
=>  abs $ (\x -> x * 2) - 10            -- по определению (*) для функций
=>  abs $ (\x -> x * 2) - \x -> 10      -- развернём 10
=>  abs $ \x -> (x * 2) - 10            -- по определению (-) для функций
=>  \x -> abs x . \x -> (x * 2) - 10    -- по определению abs для функций
=>  \x -> abs ((x * 2) - 10)            -- по определению (.)

=>  \x -> abs ((x * 2) - 10)
```

Функция возведения в квадрат:

```
*FunNat> let f = id * id
*FunNat> map f [1,2,3,4,5]
[1,4,9,16,25]
*FunNat> map (id * id - 1) [1,2,3,4,5]
[0,3,8,15,24]
```

Обратите внимание на краткость записи. В этом выражении `(id * id - 1)` проявляется основное преимущество бесточечного стиля, избавившись от аргументов, мы можем пользоваться функциями так, словно это простые значения. Этот приём используется в Haskell очень активно. Пока нам встретились лишь две инфиксных операции для функций (это композиция и применение с низким приоритетом), но в будущем вы столкнётесь с целым морем подобных операций. Все они служат одной цели, они прячут аргументы функции, позволяя быстро составлять функции на лету из примитивов. Чтобы не захлебнуться в этом море помните, что скорее всего новый символ означает либо композицию либо применение для функций специального вида.

Возведём в четвёртую степень:

```
*FunNat> map (f . f) [1,2,3,4,5]
[1,16,81,256,625]
```

Составим функцию двух аргументов, которая будет вычислять сумму квадратов двух аргументов:

```
*FunNat> let x = const id
*FunNat> let y = flip $ const id
*FunNat> let d = x * x + y * y
*FunNat> d 1 2
5
*FunNat> d 3 2
13
```

Так мы составили функцию, ни прибегая к помощи аргументов. Эти выражения могут стать частью других выражений:

```
*FunNat> filter  ((<10) . d 1) [1,2,3,4,5]
[1,2]
*FunNat> zipWith d [1,2,3] [3,2,1]
[10,8,10]
*FunNat> foldr (x*x - y*y) 0 [1,2,3,4]
3721610024
*FunNat> zipWith ((-) * (-) + const id) [1,2,3] [3,2,1]
[7,2,5]
```

В последнем выражении трудно предугадать результат. В таких выражениях всё-таки лучше пользоваться синонимами. В бесточечном стиле мы можем несколькими операциями собрать из базовых функций сложную функцию и передать её аргументом в другую функцию, которая также может поучаствовать в комбинации других функций!

1.4 Функции, возвращающие несколько значений

Как было сказано ранее функции, которые возвращают несколько значений, реализованы в Haskell с помощью кортежей. Например функция, которая расщепляет поток на голову и хвост выглядит так:

```
decons :: Stream a -> (a, Stream a)
decons (a :& as) = (a, as)
```

Здесь функция возвращает сразу два значения. Но всегда ли уместно пользоваться кортежами? Для композиции функций, которые возвращают несколько значений нам придётся разбирать возвращаемые значения с помощью сопоставления с образцом и затем использовать эти значения в других функциях. Посудите сами если у нас есть функции:

```
f :: a  -> (b1, b2)
g :: b1 -> (c1, c2)
h :: b2 -> (c3, c4)
```

Мы уже не сможем комбинировать их так просто как если бы это были обычные функции без кортежей.

```
q x = (\(a, b) -> (g a, h b)) (f x)
```

В случае пар нам могут прийти на помощь функции `first` и `second`:

```
q = first g . second h . f
```

Если мы захотим составить какую-нибудь другую функцию из `q`, то ситуация заметно усложнится. Функции, возвращающие кортежи, сложнее комбинировать в бесточечном стиле. Здесь стоит вспомнить правило Unix.

> Пишите функции, которые делают одну вещь, но делают её хорошо.

Функция, которая возвращает кортеж пытается сделать сразу несколько дел. И теряет в гибкости, ей трудно взаимодействовать с другими функциями. Старайтесь чтобы таких функций было как можно меньше.

Если функция возвращает несколько значений, попытайтесь разбить её на несколько, которые возвращают лишь одно значение. Часто бывает так, что эти значения тесно связаны между собой и такую функцию не удаётся разбить на несколько составляющих. Если у вас появляется много таких функций, то это повод задуматься о создании нового типа данных.

Например в качестве точки на плоскости можно использовать пару `(Float, Float)`. В этом случае, если вы начнёте писать модуль на геометрическую тему у вас появится много функций, которые принимают и возвращают точки:

```
rotate      :: Float -> (Float, Float) -> (Float, Float)
norm        :: (Float, Float) -> (Float, Float)
translate   :: (Float, Float) -> (Float, Float) -> (Float, Float)
...
```

Все они стараются делать несколько дел одновременно, возвращая кортежи. Но мы можем изменить ситуацию определением новых типов:

```
data Point  = Point  Float Float
data Vector = Vector Float Float
data Angle  = Angle  Float
```

Объявления функций станут более краткими и наглядными.

```
rotate      :: Angle  -> Point -> Point
norm        :: Point  -> Point
translate   :: Vector -> Point -> Point
...
```

1.5 Комбинатор неподвижной точки

Познакомимся с функцией `fix` или комбинатором неподвижной точки. По хорошему об этой функции следовало бы рассказать в разделе обобщённые функции. Но я пропустил её нарошно, для простоты изложения. В этом разделе градус сложности резко подскакивает, если вы ранее не встречались с этой функцией она может показаться вам очень необычной. Для начала посмотрим на её тип:

```
Prelude> :m +Data.Function
Prelude Data.Function> :t fix
fix :: (a -> a) -> a
```

Странно `fix` принимает функцию и возвращает значение, обычно всё происходит наоборот. Теперь посмотрим на определение:

```
fix f = let x = f x
        in  x
```

Если вы запутались, то посмыслу это определение равносильно такому:

```
fix f = f (fix f)
```

Функция `fix` берёт функцию и начинает бесконечно нанизывать её саму на себя. Так мы получаем, что-то вроде:

```
f (f (f (f (...))))
```

Зачем нам такая функция? Помните в самом конце четвёртой главы в упражнениях мы составляли бесконечные потоки. Мы делали это так:

```
data Stream a = a :& Stream a

constStream :: a -> Stream a
constStream a = a :& constStream a
```

Если смотреть на функцию `constStream` очень долго, то рано или поздно в ней проглянет функция `fix`. Я нарошно не буду выписывать, а вы мысленно обозначьте `(a :&)` за `f` и `constStream a` за `fix f`. Получилось?

Через `fix` можно очень просто определить бесконечность для `Nat`, бесконечность это цепочка `Succ`, которая никогда не заканчивается `Zero`. Оказывается, что в Haskell мы можем составлять выражения с такими значениями (как это получается мы обудим попозже):

```
ghci Nat
*Nat>m + Data.Function
*Nat Data.Function> let infinity = fix Succ
*Nat Data.Function> infinity < Succ Zero
False
```

С помощью функции `fix` можно выразить любую рекурсивную функцию. Посмотрим как на примере функции `foldNat`, у нас есть рекурсивное определение:

```
foldNat :: a -> (a -> a) -> Nat -> a
foldNat z  s  Zero      = z
foldNat z  s  (Succ n)  = s (foldNat z s n)
```

Необходимо привести его к виду:

```
x = f x
```

Слева и справа мы видим повторяются выражения `foldNat z s`, обозначим их за `x`:

```
x :: Nat -> a
x Zero      = z
x (Succ n)  = s (x n)
```

Теперь перенесём первый аргумент в правую часть, сопоставление с образцом превратится в `case`-выражение:

```
x :: Nat -> a
x = \nat -> case nat of
               Zero    -> z
               Succ n  -> s (x n)
```

В правой части вынесем `x` из выражения с помощью лямбда функции:

```
x :: Nat -> a
x = (\t -> \nat -> case nat of
                      Zero    -> z
                      Succ n  -> s (t n)) x
```

Смотрите мы обозначили вхождение `x` в выражении справа за `t` и создали лямбда-функцию с таким аргументом. Так мы вынесли `x` из выражения.

Получилось, мы пришли к виду комбинатора неподвижной точки:

```
x :: Nat -> a
x = f x
    where f = \t -> \nat -> case nat of
                      Zero    -> z
                      Succ n  -> s (t n)
```

Приведём в более человеческий вид:

```
foldNat :: a -> (a -> a) -> (Nat -> a)
foldNat z s = fix f
    where f t = \nat -> case nat of
                          Zero    -> z
                          Succ n  -> s (t n)
```

1.6 Краткое содержание

1.6.1 Основные функции высшего порядка

Мы познакомились с функциями из модуля `Data.Function`. Их можно разбить на несколько типов:

- Примитивные функции (генераторы функций).

```
id      = \x -> x
const a = \_ -> a
```

- Функции, которые комбинируют функции или функции и значения:

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

(.*.) `on` f = \x y -> f x .*. f y
```

- Преобразователи функций, принимают функцию и возвращают функцию:

```
flip f = \x y -> f y x
```

- Комбинатор неподвижной точки:

```
fix f = let x = f x
        in  x
```

1.6.2 Приоритет инфиксных операций

Мы узнали о специальном синтаксисе для задания приоритета применения функций в инфиксной форме:

```
infixl 3 #
infixr 6 `op`
```

Приоритет складывается из двух частей: старшинства (от 1 до 9) и ассоциативности (бывает левая и правая). Старшинство определяет распределение скобок между разными функциями:

```
infixl 6 +
infixl 7 *

1 + 2 * 3 == 1 + (2 * 3)
```

А ассоциативность – между одинаковыми:

```
infixl 6 +
infixr 8 ^

1 + 2 + 3 == (1 + 2) + 3
1 ^ 2 ^ 3 ==  1 ^ (2 ^ 3)
```

Мы узнали, что функции `($)` и `(.)` стоят на разных концах шкалы приоритетов функций и как этим пользоваться.

1.7 Упражнения

- Просмотрите написанные вами функции, или функции из примеров. Можно ли их переписать с помощью основных функций высшего порядка? Если да, то перепишите. Попробуйте определить их в бесточечном стиле.
- В прошлой главе у нас было упражнение о потоках. Сделайте поток экземпляром класса `Num`. Для этого поток должен содержать значения из класса `Num`. Методы из класса `Num` применяются поэлементно. Так сложение двух потоков будет выглядеть так:

```
(a1 :& a2 :& a3 :& ...) + (b1 :& b2 :& b3) ==
==  (a1 + b1 :& a2 + b2 :& a3 + b3 :& ...)
```

- Определите приоритет инфиксной операции (`:&`)
  так чтобы вам было удобно использовать её в сочетании с арифметическими операциями.

- Рассмотрим такой тип:

  ```
  data St a b = St (a -> (b, St a b))
  ```

  Этот тип хранит функцию, которая позволяет преобразовывать потоки значений. Определите функцию применения:

  ```
  ap :: St a b -> [a] -> [b]
  ```

  Она принимает ленту входящих значений и возвращает ленту выходов. Определите для этого типа несколько основных функций высшего порядка. Чтобы не возникало конфликта имён с модулем `Data.Function` мы не будем его импортировать. Вместо него мы импортируем модуль `Control.Category`. Он содержит класс:

  ```
  class Category cat where
          id  :: cat a a
          (.) :: cat b c -> cat a b -> cat a c
  ```

  Если присмотреться к типам функций, можно понять, что тип-экземпляр `cat` принимает два параметра. Совсем как тип функции `(a -> b)`. Формально его можно записать в префиксной форме так `(->) a b`. Получается, что тип `cat` это что-то вроде функции. Это некоторые сущности, у которых есть понятия тождества и композиции.

  Для обычных функций экземпляр класса `Category` уже определён. Но в этом модуле у нас есть ещё и необычные функции, функции которые преобразуют ленты значений. Функции `id` и `(.)` мы определим, сделав наш тип `St` экземпляром класса `Category`. Также определите постоянный преобразователь. Он на любой вход возвращает одно и то же число, и преобразователь, который будет накапливать сумму поступающих на вход значений, по-другому такой преобразователь называют интегратором:

  ```
  const    :: a -> St b a
  integral :: Num a => St a a
  ```

- Перепишите с помощью `fix` несколько стандартных функций для списков. Например `map`, `foldr`, `foldl`, `zip`, `repeat`, `cycle`, `iterate`.

  Старайтесь найти наиболее краткое выражение, пользуйтесь функциями высшего порядка и частичным применением. Например рассмотрим функцию `repeat`:

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

  Запишем с `fix`:

  ```
  repeat a = fix $ \xs -> a : xs
  ```

  Заметим, что мы можем избавиться от аргумента `xs` с помощью сечения:

  ```
  repeat a = fix (a:)
  ```

  Но мы можем пойти ещё дальше, если вспомним, что функция двух аргументов `(:)` является функцией от одного аргумента `(:) :: a -> ([a] -> [a])`, которая возвращает функцию одного аргумента:

  ```
  repeat = fix . (:)
  ```

  Смотрите в этом выражении мы составили композицию двух функций. Функция `(:)` примет первый аргумент и вернёт функцию, как раз то, что и нужно для `fix`.