

# Билеты к экзамену по функциональному программированию. Haskell

Тамарин Вячеслав

January 28, 2021

- 1 Основы программирования на Haskell. Связывание. Рекурсия. Базовые конструкции языка.
- 2 Основные встроенные типы языка Haskell. Система модулей. Частичное применение, каррирование.
- 3 Операторы и их сечения в Haskell. Бесточечный тип.
- 4 Ошибки. Основание. Строгие и нестрогие функции. Ленивое и энергичное исполнение.

## 4.1 Ошибки

Специальное обозначение ошибки исполнения:  $\perp$  — основание, дно, bottom. Библиотечная константа `undefined` — пример реализации  $\perp$ . Другая реализация:

```
1 bot = 1 + bot
2 fortyTwos = 42 : fortyTwos
```

Это пример продуктивной расходимости — `take 5 fortyTwos`.

Теперь мы можем написать более „аккуратную“ версию факториала

```
1 factorial n =
2   if n < 0
3   then error "factorial: negative argument"
4   else if n > 1
5       then n * factorial (n-1)
6       else 1
```

Использованная здесь библиотечная функция `error` — это гибкая версия `undefined` с настраиваемым сообщением об ошибке:

```
GHCi> factorial (-3)
*** Exception: factorial: negative argument
```

Тип `Bool` будет иметь три значения: `True`, `False`, `undefined`, так как вычисление могло не завершиться.

```
1 \bot :: forall {a}. a
```

## 4.2 Строгость функций

Haskell гарантирует вызов по необходимости, если мы запустим функцию, игнорирующую аргумент даже с расходимостью, то получим ответ. Функции, игнорирующие значение своего аргумента, называются *нестрогими* по этому аргументу.

Для *строгих* функций всегда результатом расходимости будет расходимость.

Небольшой неэффективностью ленивого вычисления является удержание всех незавершенных вычислений через указатели. Компилятор борется с этим с помощью анализатора строгости – если вычисления гарантировано строгие, он снимает нестрогость.

Другая неэффективность — нарастание отложенных вычислений (*think*). Чтобы форсировать вычисление можно использовать следующую функцию:

```
1 seq :: a -> b -> b
2 seq :: \bot b = \bot
3 seq a b = b, если a != \bot
```

Форсирование происходит до слабой головной нормальной формы, то есть до барьера распространения  $\perp$ : конструктор данных, лямбда-абстракция, частично примененная функция.

```
1 GHCi> seq (undefined, undefined) 42
2 42
3 GHCi> seq (\x -> undefined) 42
4 42
5 GHCi> seq ((+) undefined) 42
6 42
```

Через `seq` определяется энергичная аппликация (с вызовом-по-значению):

```
1 infixr 0 $!
2 ($!) :: (a -> b) -> a -> b
3 f $! x = x `seq` f x
```

Форсирование приводит к „худшей определенности“

```
1 GHCi> ignore undefined
2 42
3 GHCi> ignore $! undefined
4 *** Exception: Prelude.undefined
```

Пример использования `seq`: вспомним факториал с аккумулярующим параметром

```
1 factorial n = helper 1 n where
2   helper acc k | k > 1 = helper (acc * k) (k - 1)
3                   | otherwise = acc
```

Из-за ленивости `acc` будет содержать *think* вида

$$(\dots((1 * n) * (n - 1)) * (n - 2) * \dots * 2)$$

Оптимизатор GHC обычно справляется, имея встроенный анализатор строгости. Но можно, не полагаясь на него, написать

```
1 factorial n = helper 1 n where
2   helper acc k | k > 1 = (helper $! acc * k) (k - 1)
3                   | otherwise = acc
```

## 5 Алгебраические типы данных. Сопоставление с образцом, его семантика.

Из базовых кирпичиков типов можно строить новые с помощью суммы, произведения и возведения в степень.

### 5.1 Сопоставление с образцом

Пусть мы хотим поменять местами два элемента в паре:

```

1 swap :: (a, b) -> (b, a)
2 swap :: (x, y) = (y, x)

```

Здесь конструкция `(x, y)` представляет собой **образец**.

```

1 GHCi> swap (5 + 2, True)
2 (True, 7)

```

При вызове функции происходит **сопоставление с образцом**:

- проверяется, что конструктор `(, )` подходящий
- переменные `x` и `y` связываются с выражениями `5 + 2` и `True`
- осуществляется подстановка вместо переменных в теле функции `swap`

## 5.2 Тип суммы

**Перечисление** — тип с 0-арными конструкторами данных:

```

1 data CardinalDirection = North | East | South | West deriving Show

```

Конструкторы данных имеют тип `CardinalDirection`:

```

1 GHCi> dir = North
2 GHCi> :t dir
3 dir :: CardinalDirection

```

Сопоставление с образцом происходит сверху вниз

```

1 hasPole :: CardinalDirection -> Bool
2 hasPole North = True
3 hasPole South = True
4 hasPole _ = False

```

Подчеркивание (или переменная) задают неопровержимый образец.

```

1 GHCi> hasPole North
2 True
3 GHCi> hasPole West
4 False

```

Встроенные типы данных (`Char`, `Int`, `Integer`) ведут себя так, как будто определены как перечисления, поэтому можно использовать литералы как образцы.

## 5.3 Декартово произведение

Тип-произведение с одним конструктором данных

```

1 data PointDouble = PtD Double Double
2 deriving Show
3
4 GHCi> :type PtD
5 PtD :: Double -> Double -> PointDouble
6
7 midPointDouble :: PointDouble -> PointDouble
8 -> PointDouble
9 midPointDouble (PtD x1 y1) (PtD x2 y2) =
10 PtD ((x1 + x2) / 2) ((y1 + y2) / 2)
11
12 GHCi> midPointDouble (PtD 3.0 5.0) (PtD 9.0 8.0)
13 PtD 6.0 6.5

```

Можно параметризовывать типовым параметром:

```
1 data Point a = Pt a a
2 deriving Show
3
4 GHCi> :type Pt
5 Pt :: a -> a -> Point a
```

**Point** — оператор над типами, конкретный тип получается его аппликацией к некоторому типу, например, **Int**.

```
1 GHCi> :kind Point
2 Point :: * -> *
3
4 GHCi> :kind Point Int
5 Int :: *
```

**Кайнды** — система типов над системой типов Haskell.

## 5.4 Экспоненциальные типы

**Экспоненциальный** тип — это тип функции.

```
1 data Endom a = Endom (a -> a)
2 appEndom :: Endom a -> a -> a
3 appEndom (Endom f) = f
4
5 GHCi> e = Endom (\n -> 2 * n + 3)
6 GHCi> :t e
7 e :: Num a => Endom a
8 GHCi> :t appEndom e
9 appEndom e :: Num a => a -> a
10 GHCi> e `appEndom` 5
11 13
```

## 5.5 Рекурсивные типы

При объявлении типов можно использовать рекурсию. Это значит, что допустимо указывать объявляемый тип в качестве аргумента конструктора данных:

```
1 GHCi> data Nat = Zero | Suc Nat deriving Show
```

Конструктор **Suc** при этом оказывается эндоморфизмом над типом **Nat**

```
1 GHCi> :t Zero
2 Zero :: Nat
3 GHCi> :t Suc
4 Suc :: Nat -> Nat
```

Это позволяет строить неограниченное число обитателей данного типа:

```
1 GHCi> one = Suc Zero
2 GHCi> two = Suc (Suc Zero)
3 GHCi> three = Suc two
4 GHCi> four = Suc three
```

Это так называемые числа Пеано. Их можно рассматривать как способ кодирования натуральных чисел в унарной системе исчисления. Поскольку тип **Nat** это тип суммы, тотальные функции над ним, использующие опровержимые образцы, определяются несколькими равенствами

```

1 GHCi> {pred (Suc n) = n; pred Zero = Zero}
2 GHCi> pred two
3 Suc Zero

```

## 5.6 Стандартные алгебраические типы

- Тип `Maybe` а позволяет задать „необязательное” значение

```

1 data Maybe a = Nothing | Just a
2 maybe :: b -> (a -> b) -> Maybe a -> b
3 find :: (a -> Bool) -> [a] -> Maybe a

```

- Тип `Either` а b описывает одно значение из двух

```

1 data Either a b = Left a | Right b
2 either :: (a -> c) -> (b -> c) -> Either a b -> c
3
4 head' :: [a] -> Either String a
5 head' (x:_) = Right x
6 head' [] = Left "head': empty list"

```

## 5.7 Сопоставление с образцом

Сопоставление происходит сверху вниз, затем слева направо. Сопоставление бывает

- успешным (succeed);
- неудачным (fail);
- расходящимся (diverge).

```

1 bar (1, 2) = 3
2 bar (0, _) = 5

```

- (0, 7) — неудача в первом, успех во втором;
- (2, 1) — две неудачи и, как следствие, расходимость;
- (1, 5-3) — успех, так как для сопоставления нужно приходится форсировать
- (1, undefined) — расходимость, так как форсируем undefined
- (0, undefined) — неудача на первом, успех на втором

## 6 Объявления `type`, `newtype`. Метки полей.

## 7 Списки, стандартные функции работы с ними. Генерация (выделение) списков.

Списки встроены, но мы можем определить их сами:

```

1 data [] a = [] | a : ([] a)
2 infixr 5 :

```

Для удобства введён синтаксический сахар

```

1 [1,2,3] = 1:(2:(3:[])) = 1:2:3:[]

```

## 7.1 Стандартные функции из `Data.List`

```
1 head :: [a] -> a
2 head (x:_) = x
3 head [] = error "Prelude.head: empty list"
4
5 tail :: [a] -> [a]
6 tail (x:xs) = xs
7 tail [] = error "Prelude.tail: empty list"
```

Это частичная функция, в современном Haskell использовать их не рекомендуется.

Еще одним важнейшим оператором для списков служит оператор бинарной конкатенации: он делает из двух списков один, присоединяя первый к началу второго.

```
1 infixr 5 ++
2 (++) :: [a] -> [a] -> [a]
3 [] ++ ys = ys
4 (x:xs) ++ ys = x : xs ++ ys
```

Из реализации оператора видно, что его сложность (число рекурсивных вызовов до наступления терминирующего условия) линейно зависит от размера первого списка и не зависит от второго.

```
1 length :: [a] -> Int
2
3 concat :: [[a]] -> [a]
4 concat [] = []
5 concat (xs:xss) = xs ++ concat xss
6
7 infix 4 `elem`
8 elem :: (Eq a) => a -> [a] -> Bool
9 elem _ [] = False
10 elem x (y:ys) = x == y || elem x ys
```

Функция, осуществляющая поиск значения с заданным ключом в ассоциативном списке (то есть списке пар ключ-значение)

```
1 lookup :: Eq a => a -> [(a,b)] -> Maybe b
2 lookup _ [] = Nothing
3 lookup key ((k,v):kvs)
4   | key == k = Just v
5   | otherwise = lookup key kvs
```

Оператор, возвращающий элемент списка с заданным индексом

```
1 infixl 9 !!
2 (!! :: [a] -> Int -> a
3 xs    !! n | n < 0 = error "prelude.!!!: negative index"
4 []    !! _       = error "prelude.!!!: index too large"
5 (x:_) !! 0       = x
6 (_,xs) !! n      = xs !! (n-1)
```

Левая ассоциативность позволяет удобно обслуживать вложенные списки:

```
1 GHCi> ["Hello","world"] !! 0 !! 1
2 'e'
3 GHCi> ["Hello","world"] !! 1 !! 2
4 'r'
```

### 7.1.1 Подсписки

Функция `take` получает целое число `n` и список и возвращает первые `n` элементов списка. Если элементов меньше, чем `n`, возвращается сколько есть. Если `n` не положительно, возвращается пустой список.

```

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

```

```

1 drop :: Int -> [a] -> [a]
2 drop n xs | n <= 0 = xs
3 drop _ []         = []
4 drop n (_:xs)      = drop (n - 1) xs

```

### 7.1.2 Функции высших порядков над списками

В библиотеке `Data.List` много функций высших порядков. У следующих двух функций первый аргумент — функция типа `a -> Bool`, то есть унарный предикат.

```

1 filter :: (a -> Bool) -> [a] -> [a]
2 filter _ [] = []
3 filter p (x:xs)
4   | p x = x : filter p xs
5   | otherwise = filter p xs
6
7 takeWhile :: (a -> Bool) -> [a] -> [a]
8 takeWhile _ [] = []
9 takeWhile p (x:xs)
10  | p x = x : takeWhile p xs
11  | otherwise = []

```

У функции высшего порядка `map` функциональный аргумент — произвольная функция:

```

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

```

Функция `map` обрабатывает каждый элемент списка переданной функцией-обработчиком, формируя список результатов той же длины, но, возможно, другого типа.

### 7.1.3 Семейства `zip` и `zipWith`

```

1 zip :: [a] -> [b] -> [(a,b)]
2 zip [] _ = []
3 zip _ [] = []
4 zip (a:as) (b:bs) = (a,b) : zip as bs
5
6 zip3 :: [a] -> [b] -> [c] -> [(a,b,c)]
7 zip4 :: [a] -> [b] -> [c] -> [d] -> [(a,b,c,d)]
8 ...
9
10 unzip :: [(a,b)] -> ([a],[b])
11 unzip3 :: [(a,b,c)] -> ([a],[b],[c])
12 ...
13
14 zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
15 zipWith _ [] _ = []
16 zipWith _ _ [] = []
17 zipWith f (a:as) (b:bs) = f a b : zipWith f as bs
18
19 zipWith3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
20 ...

```

Все перечисленные семейства содержат функции вплоть до 7 аргументов.

### 7.1.4 Способы генерации списков

```
1 GHCi> take 2 ones
2 [1,1]
3 GHCi> take 4 (numsFrom 3)
4 [3,4,5,6]
5
6 take 2 ones -- (1)
7 take 2 (1 : ones) -- (2)
8 1 : take (2-1) ones -- (3)
9 1 : take 1 (1 : ones) -- (4)
10 1 : 1 : take (1-1) ones -- (5)
11 1 : 1 : take 0 ones -- (6)
12 1 : 1 : []
```

- (1) сопоставление с образцом форсирует вычисление `ones` до WHNF;
- (2) подходит последнее уравнение в определении `take`, используем его;
- (3) сопоставление с образцом форсирует вычисление до WHNF обоих аргументов `take`: первого, чтобы отвергнуть первое из уравнений в определении `take`, второго — чтобы отвергнуть второе;
- (4) используем последнее уравнение в определении `take`;
- (5) сопоставление с образцом форсирует вычисление до WHNF первого аргумента `take`;
- (6) подходит первое уравнение в определении `take`, используем его.

```
1 GHCi> squares = map (^2) (numsFrom 0)
2 GHCi> takeWhile (<=100) squares
3 [0,1,4,9,16,25,36,49,64,81,100]
4
5 GHCi> fibs = 0 : 1 : zipWith (+) fibs (drop 1 fibs)
6 GHCi> take 10 fibs
7 [0,1,1,2,3,5,8,13,21,34]
```

Для формирования „нелинейных” последовательностей имеется другая техника, носящая название выделение списка (**list comprehension**)

```
1 GHCi> digits = [0..9]
2 GHCi> [ x^2 | x <- digits ]
3 [0,1,4,9,16,25,36,49,64,81]
```

Часть справа от вертикальной черты носит название генератора: элементы, связываемые с переменной `x` пробегают по всему списку `digits`. Слева от вертикальной черты находится выражение, в котором можно использовать `x`.

При наличии нескольких генераторов чаще обновляется тот, что правее:

```
1 GHCi> [ [x,y] | x <- "ABC", y <- "de" ]
2 ["Ad","Ae","Bd","Be","Cd","Ce"]
```

Генераторы могут ссылаться на значения из предыдущих генераторов; кроме того можно использовать предикаты над этими значениями для фильтрации результатов.

### 7.1.5 Последовательности



```

1 GHCi> [1..10]
2 [1,2,3,4,5,6,7,8,9,10]
3 GHCi> [1,3..17]
4 [1,3,5,7,9,11,13,15,17]
5
6 ones = 1 : ones
7 numsFrom n = n : numsFrom (n + 1)

```

## 8 Специальный полиморфизм. Классы типов. Объявление представителей. Классы типов Eq, Ord, Enum, Bound.

### 8.1 Параметрический полиморфизм

Когда в определение функции входит тип как параметр, и мы вместо него можем подставить любой тип. Например,

```

1 id :: a -> a
2 id x = x
3
4 id True      :: Bool
5 id "Badger"  :: [Char]
6 id id        :: a -> a

```

Этот код универсален: можем использовать любой тип параметра и реализация не зависит от какой-то специфики типа.

Изучим функцию, определяющую принадлежность списку:

```

1 elem :: a -> [a] -> Bool
2 elem _ [] = False
3 elem x (y:ys) = x == y || elem x ys

```

Очевидно, для элементов нужно отношение равенства. Если для встроенных типов сравнение определено, для пользовательских типов тоже можно определить отношение равенства, то для функций все сложнее:

```

1 suc :: (forall a. (a -> a) -> a -> a) -> (a -> a) -> a -> a
2 suc = \n s z -> n s (s z)
3
4 suc' :: (forall a. (a -> a) -> a -> a) -> (a -> a) -> a -> a
5 suc' = \n s z -> s (n s z)
6
7 suc == suc'

```

С точки зрения  $\beta$ -эквивалентности это две разные функции в  $\beta$ -NF. За счет использования `forall` передать можно только числа Черча.

Эти функции поточечно равны, но не равны *интенционально* (совпали по  $\beta$ -NF). Такие называются *экстенционально* равными. Чтобы доказать, что функции равны придется для всех различные рассуждения использовать, поэтому в хаскеле функции несравнимы.

### 8.2 Специальный полиморфизм

Это вид полиморфизма, противоположный параметрическому: интерфейс полностью общий (полиморфный), но реализация специализирована для конкретных типов. Например, так устроено сложение чисел.

### 8.3 Классы типов

**Класс типов** — именованный набор имен функций с сигнатурами, параметризованными общим типовыми параметром:

```

1 class Eq a where
2   (==) :: a -> a -> Bool
3   (/=) :: a -> a -> Bool

```

Имя класса типов задает ограничение, называемое *контекстом*.

```

1 (==) :: Eq a => a -> a -> Bool
2
3 elem :: Eq a => a -> [a] -> Bool
4 elem _ [] = False
5 elem x (y:ys) = x == y || elem x ys

```

Чтобы сделать тип *представителем класса*, нужно реализовать требуемые функции класса:

```

1 instance Eq Bool where
2   True == True = True
3   False == False = True
4   _ == _ = False
5   x /= y = not (x == y)

```

Тип-представитель может быть полиморфным:

```

1 instance Eq a => Eq [a] where
2   [] == [] = True
3   (x:xs) == (y:ys) = x == y && xs == ys
4   _ == _ = False

```

Можно не реализовывать неравенство, оно определено по умолчанию. Для механической реализации можно использовать `deriving Eq`.

## 8.4 Расширение класса

Класс `Ord` наследует все методы `Eq` и содержит новые методы:

```

1 class Eq a => Ord a where
2   compare :: a -> a -> Ordering
3   (<) , (<=) , (>=) , (>) :: a -> a -> Bool
4   max, min :: a -> a -> a
5
6   compare x y = if x == y then EQ
7                 else if x <= y then LT
8                 else GT
9
10  x < y = case compare x y of {LT -> True; _ -> False}
11  x <= y = case compare x y of {GT -> False; _ -> True}
12  x > y = case compare x y of {GT -> True; _ -> False}
13  x >= y = case compare x y of {LT -> False; _ -> True}
14  max x y = if x <= y then y else x
15  min x y = if x <= y then x else y
16
17 sort :: Ord a => [a] -> [a]

```

Допустимо множественное наследование:

```

1 class (Eq a, Show a) => MyClass a where
2   ...

```

## 8.5 Enum, Bounded

```

1 class Enum a where
2   succ, pred    :: a -> a
3   toEnum        :: Int -> a
4   fromEnum      :: a -> Int
5
6   enumFrom      :: a -> [a]           -- [n..]
7   enumFromThen  :: a -> a -> [a]      -- [n, n'..]
8   enumFromTo    :: a -> a -> [a]      -- [n..m]
9   enumFromThenTo :: a -> a -> a -> [a] -- [n, n'..m]
10
11 class Bounded a where
12   minBound, maxBound :: a

```

Класс `Integer` потенциально бесконечен в обе стороны, поэтому он является `Enum`, но не является `Bounded`.

## 9 Внутренняя реализация классов типов.

Организована через механизм передачи словарей. Словарь для класса — запись его методов.

```

1 data Eq' a = MkEq { eq, ne :: a -> a -> Bool}

```

Здесь хранятся две *функции-селекторы*, они выбирают методы равенства и неравенства из данного словаря.

```

1 GHCi> :t eq
2 eq :: Eq' a -> a -> a -> Bool
3 GHCi> :i ne
4 eq :: Eq' a -> a -> a -> Bool

```

Теперь вместо того, что называлось контекстом, теперь просто дополнительный параметр.

Объявления представителей транслируются в функции, возвращающие словарь или в функции, принимающие словарь и возвращающие более сложный словарь:

```

1 dEqInt :: Eq' Int
2 dEqInt = MkEq {
3   eq = eqInt, -- это низкоуровневый способ сравнивать Intы
4   ne = \x y -> not $ eqInt x y
5 }
6
7 dEqList :: Eq' a -> Eq' [a]
8 dEqList (MkEq e _) = MkEq el (\x y -> not $ el x y) -- e - словарь для сравнения элементов списка
9   where el [] [] = True
10         el (x:xs) (y:ys) = x `e` y && xs `el` ys
11         el _ _ = False

```

Теперь можем написать `enum'`: он требует, чтобы ему дали словарь для сравнения элементов типа `a` первым аргументом, а остальное аналогично.

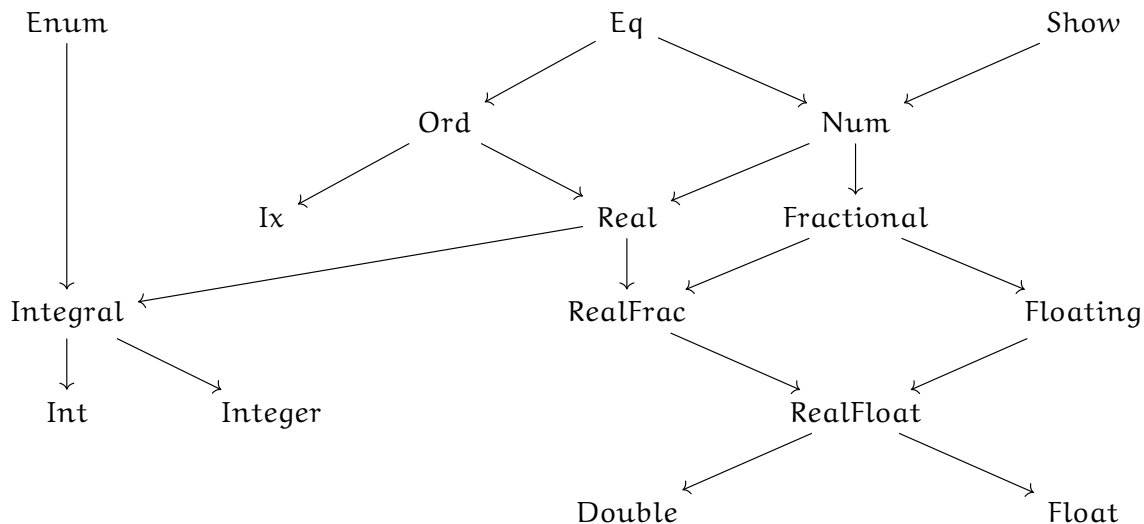
```

1 elem' :: Eq' a -> a -> [a] -> Bool
2 elem' _ _ [] = False
3 elem' d x (y:ys) = eq d x y || elem' d x ys
4
5 GHCi> elem' dEqInt 2 [3, 5, 2]
6 True
7 GHCi> elem (dEqList dEqInt) [3, 5] [[4], [1, 2, 3], [3, 5]]
8 True

```

## 10 Стандартные классы типов: Num и его наследники, Show и Read.

```
1 class (Eq a, Show a) => Num a where
2   (+), (-), (*) :: a -> a -> a
3   negate      :: a -> a
4   abs, signum  :: a -> a
5   fromInteger  :: Integer -> a
6
7   x - y = x + negate y
8   negate x = 0 - x
```



Автоматического приведения типов в Хаскеле нет. Всегда нужно использовать приведение.

## 11 Полугруппы и моноиды. Представители класса типов Semigroup и Monoid.

### 11.1 Полугруппы

Полугруппа — множество с ассоциативной бинарной операцией над ним.

```
1 infixr 6 <>
2 class Semigroup a where
3   (<>) :: a -> a -> a
4   sconcat :: NonEmpty a -> a -- конкатенация, все : заменяем на <>
5   stimes :: Integral b => b -> a -> a -- повторяем b раз и вставляем между копиями <>
6
7 infixr 5 :|
8 data NonEmpty a = a :| [a] -- произведение a и [a]
```

За счет ассоциативности операции, можно достичь логарифмической трудности операций. Для любой полугруппы должен выполняться закон:

```
1 (x <> y) <> z = x <> (y <> z)
```

Список — полугруппа относительно конкатенации:

```
1 class Semigroup [a] where
2   (<>) = (++)
```

Можем использовать в функциях, которые не работают на пустых списках:

```
1 GHCi> import Data.List.NonEmpty
2 GHCi> sconcat $ "AB" :| ["CDE", "FG"]
3 "ABCDEFGFG"
4 GHCi> stimes 5 "Ab"
5 "AbAbAbAbAb"
```

## 11.2 Моноиды

**Моноид** — множество с ассоциативной бинарной операцией над ним и нейтральным элементом для этой операции.

```
1 class Semigroup a => Monoid a where
2   mempty :: a -- нейтральный элемент
3   mappend :: a -> a -> a -- операция
4   mappend = (<>)
5   mconcat :: [a] -> a
6   mconcat = foldr mappend mempty
```

Законы:

```
1 mempty <> x = x
2 x <> mempty = x
3 (x `mappend` y) `mappend` z = x `mappend` (y `mappend` z)
```

Список — моноид относительно (++), нейтральный — пустой список:

```
1 instance Semigroup [a] where
2   (<>) = (++)
3 instance Monoid [a] where
4   mempty = []
5   mconcat = concat
```

Числа тоже моноид, причем четырежды: сложение — ноль, умножение — единица, минимум — `maxBound`, максимум — `minBound`. Для разделения этих вариантов при реализации можем обернуть числа в `newtype`-коробочку:

```
1 newtype Sum a = Sum { getSum :: a } deriving (Eq, Ord, Read, Show, Bounded)
2 instance Num a => Semigroup (Sum a) where
3   Sum x <> Sum y = Sum (x + y)
4 instance Num a => Monoid (Sum a) where
5   mempty = Sum 0
6
7 GHCi> Sum 3 <> Sum 2
8 Sum {getSum = 5}
```

`mconcat` — сумма нескольких чисел, `stimes` — умножение.

Теперь напишем для умножения, используя `coerce` — безопасное приведение, если мы имеем два одинаковых рантаймовых представления, то можем привести одно к другому автоматически:

```
1 newtype Product a = Product { getProduct :: a } deriving (Eq, Ord, Read, Show, Bounded)
2 instance Num a => Semigroup (Product a) where
3   (<>) = coerce ((* :: a -> a -> a) -- Data.Coerce
4 instance Num a => Monoid (Product a) where
5   mempty = Product 1
6
7 GHCi> Product 3 <> Product 2
8 Product {getProduct = 6}
```

Для использования `coerce`, нужно подключить расширение `ScopedTypeVariables`, чтобы расширить область видимости `a` из типа.

`Integer` не имеет максимума, поэтому, хотя это и полугруппа, моноидом быть не может. Также не существует моноида для строк.

Для решения такой проблемы можем перейти к полугрупповой операции `sconcat` и от списка к `NoEmpty`:

```
1 GHCi> (getMin . sconcat . fromList . fmap Min) ["Hello", "Hi"]
2 "Hello"
3 -- вместо
```

```

4 GHCi> (getMin . mconcat . fmap Min) ["Hello", "Hi"]
5 <interactive>: error: No instance for (Bounded [Char])

```

Булев тип тоже является моноидом относительно конъюнкции и дизъюнкции.

```

1 newtype All = All { getAll :: Bool } deriving (Eq, Ord, Read, ShowBounded)
2 instance Semigroup All where
3   (< >) = coerce (&&)
4 instance Monoid All where
5   mempty = All True
6
7 newtype Any = Any { getAny :: Bool } deriving (Eq, Ord, Read, ShowBounded)
8 instance Semigroup Any where
9   (< >) = coerce (||)
10 instance Monoid Any where
11   mempty = Any False

```

## 12 Свертки списков. Правая и левая свертки. Энергичные версии. Развертки.

### 12.1 Правая свертка

```

1 foldr :: (a -> b -> b) -> b -> [a] -> b
2 foldr f ini [] = ini
3 foldr f ini (x:xs) = x `f` (foldr f ini xs)
4
5 p : q : r : [] =====> p `f` (q `f` (r `f` ini))

```

Примеры использования:

```

1 sum :: [Integer] -> Integer
2 sum = foldr (+) 0
3
4 concat :: [[a]] -> [a]
5 concat = foldr (++) []
6
7 allOdd :: [Integer] -> Bool
8 allOdd = foldr (\n b -> odd n && b) True
9
10 id = foldr (:) []

```

Правые свертки позволяют работать с бесконечными списками.

### 12.2 Левая свертка

```

1 foldl :: (b -> a -> b) -> b -> [a] -> b
2 foldl f ini [] = ini
3 foldl f ini (x:xs) = foldl f (ini `f` x) xs
4
5 p : q : r : [] =====> ((ini `f` p) `f` q) `f` r

```

Здесь хвостовая рекурсия оптимизируется, но нарастает thunk из цепочки вызовов: запустятся вычисления только, когда список кончится, так как форсирования нет (нет `seq` или сопоставления с образцом).

```

1 foldl' :: (b -> a -> b) -> b -> [a] -> b
2 foldl' f ini [] = ini
3 foldl' f ini (x:xs) = arg `seq` foldl' f arg xs
4                       where arg = f ini x

```

Здесь thunk не нарастает, вычисление `arg` форсируется на каждом шаге, это самая эффективная из сверток, но, как и любая левая свертка, не работает с бесконечным циклом.

## 12.3 Версии без начального значения

```
1 foldr1 :: (a -> a -> a) -> [a] -> a
2 foldr1 _ [x] = x
3 foldr1 f (x:xs) = f x (foldr1 f xs)
4 foldr1 _ [] = error "foldr1: EmptyList"
```

```
1 foldl1 :: (a -> a -> a) -> [a] -> a
2 foldl1 f (x:xs) = foldl f x xs
3 foldl1 _ [] = error "foldl1: EmptyList"
```

Аналогично можно сделать строгую версию `foldl1'`.

## 12.4 Сканы

Это списки последовательных шагов свертки:

```
1 scanl :: (b -> a -> b) -> b -> [a] -> [b]
2 scanl _ z [] = [z]
3 scanl (#) z (x:xs) = z : scanl (#) (z # x) xs
4
5 GHCi> scanl (++) "!" ["a", "b", "c"]
6 ["!", "!a", "!ab", "!abc"]
7 GHCi> scanl (*) 1 [1..] !! 5 -- факториал
8 120
```

В отличие от `foldl` может работать и с бесконечными списками.

Правый скан накапливает результаты в обратном порядке:

```
1 scanr :: (a -> b -> b) -> b -> [a] -> [b]
2 scanr _ z [] = [z]
3 scanr f z (x:xs) = f x q : qs
4                      where qs@(q:_) = scanr f z xs -- здесь используем псевдоним
```

Тождества для сканов:

```
1 head (scanr f z xs) = foldr f z xs
2 last (scanl f z xs) = foldl f z xs
```

## 12.5 Развертка

Двойственная свертке операция.

```
1 unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
2 unfoldr g ini
3   | Nothing   <- next = []
4   | Just (a, b) <- next = a : unfoldr g b
5   where next = g ini
6
7 GHCi> helper x = if x == 0 then Nothing else Just (x, x - 1)
8 GHCi> unfoldr helper 10
9 [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Пример использования — возможное определение `iterate`

```
1 iterate f = unfoldr (\x -> Just (x, f x))
```

## 13 Класс типов `Foldable` и его представители.

Идея: обобщить свертки на более общий тип, то есть заменить скобки на букву.

```
1 class Foldable t where
2   fold :: Monoid m => t m -> m
3   fold = foldMap id
4
5   foldMap :: Monoid m => (a -> m) -> t a -> m
6   foldMap f = foldr (mappend . f) mempty
7
8   foldr, foldr' :: (a -> b -> b) -> b -> t a -> b
9   foldr f z t = appEndo (foldMap (Endo . f) t) z
10
11  foldl, foldl' :: (b -> a -> b) -> a -> t b -> a
12  foldl f z t = appEndo (getDual (foldMap (Dual . Endo . flip f) t)) z
13
14  foldr1, foldl1 :: (a -> a -> a) -> t a -> a
```

Минимальное определение `foldMap` или `foldr`. Изучим полезные функции:

```
1 class Foldable t where
2   toList :: t a -> [a]
3
4   null :: t a -> Bool
5   null = foldr (\_ _ -> False) True
6
7   length :: t a -> Int
8   length = foldl' (\c _ -> c + 1) 0
9
10  elem :: Eq a => a -> t a -> Bool
11  elem x = foldr f False
12          where f result elem = result || x == elem
13
14  maximum, minimum :: Ord a => t a -> a
15
16  sum, product :: Num a => t a -> a
17  sum = getSum . foldMap Sum
```

### 13.1 Представители

```
1 instance Foldable [] where
2   foldr = Prelude.foldr
3   foldl = Prelude.foldl
4   foldr1 = Prelude.foldr1
5   foldl1 = Prelude.foldl1
6
7 instance Foldable Maybe where
8   foldr _ z Nothing = z
9   foldr f z (Just x) = f x z
```

Еще много контейнеров: `Set`, `Map`, `Tree`, `Seq`. Также `Either`, `Pair`. Чтобы объявить представителя двухпараметрического типа, первый параметр нужно связать.

```
1 GHCi> foldr (+) 5 (Right 37)
2 42
3 GHCi> foldr (+) 5 (Left 37)
4 5
5 GHCi> foldr (+) 5 ("Answer", 37)
6 42
7
8 GHCi> maximum (Right 37)
```



```

9 37
10 GHCi> maximum (Left 37)
11 *** Exception: maximum: empty structure
12 GHCi> maximum (100, 42)
13 42

```

Реализация:

```

1 foldr f z t = appEndo (foldMap (Endo . f) t) z
2 foldl f z t = appEndo (getDual (foldMap (Dual . Endo . flip f) t)) z
3 fold      = foldMap id
4 length    = getSum . foldMap (Sum . const 1)
5 sum       = getSum . foldMap Sum
6 product   = getProduct . foldMap Product
7 minimum   = getMin . foldMap Min
8 maximum   = getMax . foldMap Max
9 foldr f z = foldr f z . toList
10 foldl f z = foldl f z . toList

```

Другие моноиды: `Ordering`, `SortedList`, `a -> b`.

```

1 newtype Endo a = Endo { appEndo :: a -> a }
2 instance Semigroup (Endo a) where
3   Endo f <> Endo g = Endo $ f . g
4 instance Monoid (Endo a) where
5   mempty = Endo id
6
7 newtype Dual a = Dual { getDual :: a } -- разворачивает аргументы
8 instance Semigroup a => Semigroup (Dual a) where
9   Dual a <> Dual b = Dual (b <> a)
10 instance Monoid a => Monoid (Dual a) where
11   mempty = Dual mempty

```

## 14 Класс типов `Functor` и его представители.

Функтор определяется следующим образом:

```

1 class Functor f where
2   fmap :: (a -> b) -> (f a -> f b)
3   (<$) :: a -> f b -> f a -- стрелка направлена в ту сторону, откуда берется значение
4   (<$) = fmap . const
5
6   (<$>) :: Functor f => (a -> b) -> f a -> f b
7   (<$>) fmap
8
9   ($>) :: Functor f => f a -> b -> f b
10  ($>) = flip (<$)
11
12 void :: Functor f => f a -> f () -- это unittype, ровно одно значение
13 void x = () <$ x
14
15 infixl 1 <&>
16 (<&>) :: Functor f => f a -> (a -> b) -> f b
17 xs <&> f = f <$> xs

```

Представители этого класса должны быть конструкторами типа с одним параметром: `f :: * -> *`.

```

1 instance Functor [] where
2   fmap _ [] = []
3   fmap g (x:xs) = g x : fmap g xs
4
5 instance Functor Maybe where
6   fmap _ Nothing = Nothing

```

```

7  fmap g (Just a) = Just (g a)
8
9  data Tree a = Leaf | Branch (Tree a) a (Tree a)
10 instance Functor Tree where
11   fmap :: (a -> b) -> Tree a -> Tree b
12   fmap g (Leaf x) = Leaf (g x)
13   fmap g (Branch l x r) = Branch (fmap g l) (g x) (fmap g r)

```

Структурно ничего не меняется после применения `fmap`, `<$>` — инфиксный аналог.

Расширение `InstanceSigs` позволяет указывать сигнатуры методов в представителе класса типов.

Представители для двухпараметрических типов: так как `Either`, `(,)`, `(->) :: * -> * -> *` мы должны связать первый параметр.

```

1  instance Functor (Either e) where
2    fmap :: (a -> b) -> Either e a -> Either e b
3    fmap _ (Left x) = Left x
4    fmap g (Right y) = Right (g y)
5
6  instance Functor ((,) s) where
7    fmap :: (a -> b) -> (s, a) -> (s, b)
8    fmap g (x, y) = (x, g y)
9
10 instance Functor ((->) e) where
11   fmap :: (a -> b) -> ((->) e a) -> ((->) e b)
12   fmap = (.)

```

## 14.1 Законы

Вообще, в Хаскеле из первого следует второй закон.

```

1  fmap id = id
2  fmap (f . g) = fmap f . fmap g

```

Смысл законов — сохраняем структуру контейнера, воздействуя только на элементы. Всегда нужно следить за соблюдением законов, например, есть такой нарушитель:

```

1  instance Functor [] where
2    fmap _ [] = []
3    fmap g (x:xs) = g x : g x : fmap g xs

```

## 14.2 Не-функторы

Единственная реализация не удовлетворяет первому закону:

```

1  instance Functor Endo where
2    fmap :: (a -> b) -> Endo a -> Endo b
3    fmap _ (Endo _) = Endo id

```

Вообще не допускает реализации:

```

1  newtype RevArr c a = RevArr { appRevArr :: a -> c }

```

## 15 Класс типов `Applicative` и его представители

Хотим вынуть стрелку из контейнера.

```

1 class Functor f => Applicative f where
2   {-# Minimal pure, ((<*>) / liftA2) #-}
3   pure :: a -> f a
4
5   (<*>) :: f (a -> b) -> f a -> f b
6   (<*>) = liftA2 id
7
8   liftA2 :: (a -> b -> c) -> f a -> f b -> f c
9   liftA2 g a b = g <$> a <*> b
10
11   (*>) :: f a -> f b -> f b
12   a1 *> a2 = (id <$ a1) <*> a2
13
14   (<*) :: f a -> f b -> f a
15   (<*) = liftA2 const
16
17 infixl 4 <*>, *>, <*, <***>

```

`pure` — чистая упаковка, вложение без эффекта.

Вспомогательные функции из `Control.Applicative`

```

1 liftA :: Applicative f => (a -> b) -> f a -> f b
2 liftA f a = pure f <*> a
3
4 liftA3 :: Applicative f => (a -> b -> c -> d) -> f a -> f b -> f c -> f d
5 liftA3 g a b c = g <$> a <*> b <*> c
6
7 (<***>) :: Applicative f => f a -> f (a -> b) -> f b
8 (<***>) = liftA2 (&)

```

## 15.1 Законы

- Закон, связывающий `Applicative` и `Functor`: `fmap g xs = pure g <*> xs`
- Identity: `pure id <*> v = v`
- Homomorphism: `pure g <*> pure x = pure (g x)`
- Interchange: `u <*> pure x = pure ($ x) <*> u`
- Composition: `pure (.) <*> u <*> x = u <*> (v <*> x)`

## 15.2 Представители

Контекст с возможно отсутствующим значением (примитивная обработка ошибок):

```

1 instance Applicative Maybe where
2   pure = Just
3   Nothing <*> _ = Nothing
4   (Just g) <*> x = fmap g x
5
6 GHCi> Just (+2) <*> Just 5
7 Just 7
8 GHCi> Just (+2) <*> Nothing
9 Nothing
10 GHCi> Just (+) <*> Just 2 <*> Just 5
11 Just 7

```

Для списков есть две семантики `fs <*> xs`: либо каждый с каждым (множественные результаты недетерминированного вычисления), либо `zip` (коллекция упорядоченных элементов). Список определен только для первой, так как одна расширяется до `Monad`.

```

1 instance Applicative [] where
2   pure x = [x]
3   gs <*> xs = [ g x | g <- gs, x <- xs ]
4
5 newtype ZipList a = ZipList { getZipList :: [a] }
6
7 instance Functor ZipList where
8   fmap f (ZipList xs) = ZipList (map f xs)
9 instance Applicative ZipList where
10  pure x = x : pure x -- unu repeat x us Data.List
11  ZipList gs <*> ZipList xs = ZipList (zipWith ($) gs xs)

```

Для пары выполняется семантика логгирования:

```

1 instance Monoid s => Applicative ((,), s) where
2   pure x = (mempty, x)
3   (u, f) <*> (v, x) = (u <> v, f x)
4
5 GHCi> ("Answer to ", (*)) <*> ("the Ultimate ", 6) <*> ("Question ", 7)
6 ("Answer to the Ultimate Question ", 42)

```

## 16 Классы типов `Alternative`, `MonadPlus` и их представители.

### 16.1 `Alternative`

```

1 class Applicative f => Alternative f where
2   empty :: f a
3   (<|>) :: f a -> f a -> f a
4
5   some, many :: f a -> f [a]
6   some v = (:) <$> v <*> many v -- One or more
7   many v = some v <|> pure [] -- Zero or more
8   infixl 3 <|>

```

Мы наделяем аппликативный функтор дополнительной моноидальной операцией с семантикой сложения:

```

1 instance Alternative [] where
2   empty :: [a]
3   empty = []
4   (<|>) :: [a] -> [a] -> [a]
5   (<|>) = (++)

```

```

1 instance Alternative Maybe where
2   empty :: Maybe a
3   empty = Nothing
4
5   (<|>) :: Maybe a -> Maybe a -> Maybe a
6   Nothing <|> m = m
7   m <|> _ = m
8
9 optional :: Alternative f => f a -> f (Maybe a)
10 optional v = Just <$> v <|> pure Nothing

```

Представитель `Alternative` для `Maybe` ведет себя, как упаковка `First`, возвращая первый не-`Nothing` в цепочке альтернатив:

```

1 GHCi> Nothing <|> (Just 3) <|> (Just 5) <|> Nothing
2 Just 3

```

Интерфейс для парсера

```

1 instance Alternative (Parser tok) where
2   empty :: Parser tok a
3   empty = Parser $ \_ -> Nothing
4
5   (<|>) :: Parser tok a -> Parser tok a -> Parser tok a
6   Parser u <|> Parser v = Parser f where
7     f xs = case u xs of
8       Nothing -> v xs
9       z -> z

```

СЕМАНТИКА: `empty` — парсер, всегда возвращающий неудачу; `(<|>)` — пробуем первый, при неудаче пробуем второй на исходной строке.

Рекурсивный парсер

```

1 lowers :: Parser Char String
2 lowers = (:) <$> lower <*> lowers <|> pure ""
3
4 GHCi> runParser lowers "abCd"
5 Just ("Cd", "ab")
6 GHCi> runParser lowers "abcd"
7 Just ("", "abcd")
8 GHCi> runParser lowers "Abcd"
9 Just ("Abcd", "")

```

## 16.2 MonadPlus

Расширение дополнительной моноидальной операцией. Очень похоже на `Alternative`.

```

1 class (Alternative m, Monad m) => MonadPlus m where
2   mzero :: m a
3   mzero = empty
4   mplus :: m a -> m a -> m a
5   mplus = (<|>)

```

Минимальное полное определение: ничего не делать.

```

1 instance MonadPlus Maybe
2 instance Alternative [] where
3   empty = []
4   (<|>) = (++)
5
6 instance MonadPlus []

```

### 16.2.1 Законы

Помимо „унаследованных“ законов требуют выполнения (Left and Right Zero)

```

1 mzero >= k = mzero
2 v >> mzero = mzero

```

и по крайней мере одного из двух (Left Distribution, Left Catch law)

```

1 (a `mplus` b) >= k = (a >= k) `mplus` (b >= k)
2 return a `mplus` b = return a

```

## 16.2.2 Использование

```
1 -- Haskell 2010 :: MonadPlus m => Bool -> m ()
2 guard :: Alternative f => Bool -> f ()
3 guard True = pure ()
4 guard False = empty
5
6 pythags = do
7   z <- [1..]
8   x <- [1..z]
9   y <- [x..z]
10  guard (x^2 + y^2 == z^2)
11  return (x, y, z)
12
13 GHCi> take 5 pythags
14 [(3,4,5),(6,8,10),(5,12,13),(9,12,15),(8,15,17)]
15
16 asum :: (Foldable t, Alternative f) => t (f a) -> f a
17 asum = foldr (<|>) empty
18
19 msum :: (Foldable t, MonadPlus m) => t (m a) -> m a
20 msum = asum -- foldr mplus mzero
21
22 mfilter :: MonadPlus m => (a -> Bool) -> m a -> m a
23 mfilter p ma = do
24   a <- ma
25   if p a
26   then return a
27   else mzero
```

## 17 Аппликативные парсеры

### Определение 1

Парсер — программа, принимающая на вход строку и возвращающая некоторую структуру данных, если строка удовлетворяет заданной грамматике, или сообщение об ошибке в противном случае.

Различные парсеры:

- Простой, но неудобный: `type Parser a = String -> a`
- Храним неразобранный остаток: `type Parser a = String -> (String, a)`
- Обрабатываем ошибки: `type Parser a = String -> Maybe (String, a)` или `type Parser a = String -> Either String (String, a)`
- Разбираем неоднозначные грамматики: `type Parser a = String -> [(String, a)]`

Выберем третий вариант. Также существует обобщение на список токенов, который мы обрабатываем вместо строки, если перед этим мы разбили текст на лексемы / токены.

```
1 newtype Parser tok a = Parser { runParser :: [tok] -> Maybe ([tok], a) }
```

Можно написать простой парсер, распознающий строку, начинающуюся с символа 'A':

```
1 charA :: Parser Char Char
2 charA = Parser f where
3   f (c:cs) | c == 'A' = Just (cs, c)
4   f _ = Nothing
5
6 GHCi> runParser charA "ABC"
7 Just ("BC", 'A')
```

Напишем более общую функцию, которая разбирает токен, если он удовлетворяет предикату, а иначе выдает ошибку:

```
1 satisfy :: (tok -> Bool) -> Parser tok tok
2 satisfy pr = Parser f where
3   f (c:cs) | pr c = Just (cs,c)
4   f _ = Nothing
5
6 GHCi> runParser (satisfy isUpper) "ABC" -- можно использовать функции из Data.Char, Data.Punctuation
7 Just ("BC",'A')
8 GHCi> runParser (satisfy isLower) "ABC"
9 Nothing
10
11 lower :: Parser Char Char
12 lower = satisfy isLower
13 char :: Char -> Parser Char Char
14 char c = satisfy (== c)
```

Чтобы поменять тип возвращаемого значения, например, возвращать цифру, а не символ, нужно сделать парсер функтором:

```
1 digit :: Parser Char Int
2 digit = digitToInt <$> satisfy isDigit
3
4 instance Functor (Parser tok) where
5   fmap :: (a -> b) -> Parser tok a -> Parser tok b
6   fmap g (Parser p) = Parser f where
7     f xs = case p xs of
8       Just (cs, c) -> Just (cs, g c)
9     Nothing -> Nothing
```

Можно сделать это компактнее, так как `Parser` — композиция трех функторов `(->)` `[tok]`, `Maybe`, `(,)` `[tok]`:

```
1 newtype Parser tok a = Parser { runParser :: [tok] -> Maybe ([tok], a) }
2
3 instance Functor (Parser tok) where
4   fmap :: (a -> b) -> Parser tok a -> Parser tok b
5   fmap g (Parser p) = Parser $ (fmap . fmap . fmap) g p
```

Основной эффект парсера состоит в том, что мы постепенно едем по строке и отжираем от нее кусочки.

Семантика `pure` — ничего не откусить, семантика `<*>` — получить результат первого парсера, затем второго на остатке и применить первый ко второму.

Тогда можем сделать парсер аппликативным:

```
1 instance Applicative (Parser tok) where
2   pure :: a -> Parser tok a
3   pure x = Parser $ \s -> Just (s, x)
4
5   (<*>) :: Parser tok (a -> b) -> Parser tok a -> Parser tok b
6   Parser u <*> Parser v = Parser f where
7     f xs = case u xs of
8       Nothing -> Nothing
9       Just (xs', g) -> case v xs' of
10         Nothing -> Nothing
11         Just (xs'', x) -> Just (xs'', g x)
12
13 GHCi> runParser (pure 42) "ABCD"
14 Just ("ABCD",42)
15 GHCi> runParser (pure (,) <*> digit <*> digit) "12AB"
16 Just ("AB",(1,2))
17 GHCi> runParser ((,) <$> digit <*> digit) "1AB2"
18 Nothing
```

Построим парсер, который перемножает два числа:

```
1 multiplication :: Parser Char Int
2 multiplication = (*) <$> digit <*> char '*' <*> digit -- <*> говорит, что результат звездочки просто игнорируем
3
4 GHCi> runParser multiplication "6*7"
5 Just ("",42)
```

## 18 Класс типов `Traversable` и его представители.

Пусть есть список с аппликативными функторами, а мы хотим вытащить функтор наружу:

```
1 dist :: Applicative f => [f a] -> f [a]
2 dist [] = pure []
3 dist (ax:axs) = pure (:) <*> ax <*> dist axs
4
5 GHCi> dist [Just 3,Just 5]
6 Just [3,5]
7 GHCi> dist [Just 3,Nothing]
8 Nothing
9 GHCi> getZipList $ dist $ map ZipList [[1,2],[3,4],[5,6]]
10 [[1,3,5],[2,4,6]]
```

Использование в теле определения `dist` тех же самых конструкторов, что и в образцах, приводит к сохранению трехэлементной структуры списка.

Обобщим до любого однопараметрического типа.

```
1 class (Functor t, Foldable t) => Traversable t where
2   sequenceA :: Applicative f => t (f a) -> f (t a)
3   sequenceA = traverse id
4   traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
5   traverse g = sequenceA . fmap g
```

Минимальное определение: `traverse` или `sequenceA`.

- `sequenceA`: обеспечиваем правило коммутации нашего функтора `t` с произвольным аппликативным функтором `f`.

Структура внешнего контейнера `t` сохраняется, а аппликативные эффекты внутренних `f` объединяются в результирующем `f`.

- `traverse` — это `fmap` с эффектами: проезжаем по структуре `t` а, последовательно применяя функцию к элементам типа `a` и монтируем в точности ту же структуру из результатов типа `b`, параллельно „коллекционируя” эффекты.

Если реализовать `Traversable`, то он будет и функтором и `Foldable`.

### 18.1 Представители

```
1 instance Traversable Maybe where
2   traverse :: Applicative f => (a -> f b) -> Maybe a -> f (Maybe b)
3   traverse _ Nothing = pure Nothing
4   traverse g (Just x) = Just <$> g x
5
6 instance Traversable [] where
7   traverse :: Applicative f => (a -> f b) -> [a] -> f [b]
8   traverse _ [] = pure []
9   traverse g (x:xs) = (:) <$> g x <*> traverse g xs
```

Можно заметить, что с функтором очень схожий код, просто все происходит поднятым на уровень выше.



## 18.2 Законы

```
1 newtype Identity a = Identity { runIdentity :: a }
2
3 instance Functor Identity where
4   fmap g (Identity x) = Identity (g x)
5
6 instance Applicative Identity where
7   pure = Identity
8   Identity g <*> v = fmap g v
```

Первый закон (Identity):

```
1 traverse Identity = Identity
2
3 GHCi> traverse Identity [1,2,3]
4 Identity [1,2,3]
```

Всякий **Traversable** — это **Functor**: имея **traverse** мы можем универсальным образом реализовать **fmap**, удовлетворяющий законам функтора.

```
1 fmapDefault :: Traversable t => (a -> b) -> t a -> t b
2 fmapDefault g = runIdentity . traverse (Identity . g)
```

```
1 Identity :: b -> Identity b
2 (Identity . g) :: a -> Identity b
3 traverse (Identity . g) :: Identity (t b)
4 runIdentity . traverse (Identity . g) :: t b
```

Второй закон (Composition):

```
1 traverse (Compose . fmap f2 . f1) = Compose . fmap (traverse f2) . traverse f1
```

Здесь обе части имеют тип **t a -> Compose g2 g1 (t c)** в предположении, что **f1 :: a -> g1 b** и **f2 :: b -> g2 c**.

Третий закон (Naturality):

```
1 h . traverse f = traverse (h . f)
```

Здесь **h :: (Applicative f, Applicative g) => f b -> g b** — произвольный аппликативный гомоморфизм, то есть функция удовлетворяющая требованиям:

- **h (pure x) = pure x**;
- **h (x <\*> y) = h x <\*> h y**.

В предположении, что **f :: a -> f b**, обе части имеют тип **t a -> g (t b)**.

## 18.3 Практический смысл законов

Законы **Traversable** дают следующие гарантии:

- Траверсы не пропускают элементов.
- Траверсы посещают элементы не более одного раза.
- **traverse pure** дает **pure**.
- Траверсы не изменяют исходную структуру — она либо сохраняется, либо полностью исчезает.

```

1 GHCi> traverse Just [1,2,3]
2 Just [1,2,3]
3 GHCi> traverse (const Nothing) [1,2,3]
4 Nothing

```

## 19 Монады. Класс типов `Monad`. Законы для монад `do`-нотация.

### 19.1 Эффект

Хотим вместо простой чистой функции повесить на результат некоторый контекст.

- иногда могут завершиться неудачей: `a -> Maybe b`
- могут возвращать много результатов: `a -> [b]`
- иногда могут завершиться ошибкой: `a -> (Either s) b`
- могут делать записи в лог: `a -> (s, b)`
- могут читать из внешнего окружения: `a -> ((->) e) b`
- работают с мутабельным состоянием: `a -> (State s) b`
- делают ввод/вывод (файлы, консоль): `a -> IO b`
- не делают ничего: `a -> Identity b`

Обобщая, получим стрелку Клейсли: `a -> m b`. Она обобщает зависимость *эффекта от значения*.

На оператор над типами накладываются требования:

- Должен иметься универсальный интерфейс для упаковки значения в контейнер `m`
- Должен иметься универсальный интерфейс для композиции вычислений с эффектом (стрелок Клейсли)

```

1 (<=>) :: (b -> m c) -> (a -> m b) -> (a -> m c)

```

Но используя только интерфейс функтора не реализовать, нужно сделать контекст плоским:

```

1 k1 <=> k2 = \x -> k1 <$> (k2 x) -- увы :: m (m c)

```

- Но нет универсального интерфейса для извлечения из контекста.

```

1 class Applicative m => Monad m where
2   return :: a -> m a
3   (>=) :: m a -> (a -> m b) -> m b -- производят bind
4   (>>) :: m a -> m b -> m b
5   m1 >> m2 = m1 >= \_ -> m2
6
7   fail :: String -> m a -- изгнан в MonadFail с >= 8.6
8   fail s = error s
9   infixl 1 >>, >=

```

Из обычной функции можно получить стрелку Клейсли:

```

1 toKleisli :: Monad m => (a -> b) -> (a -> m b)
2 toKleisli f = \x -> return (f x)

```

Связыватель `(>=)` похож на `(&)`, только имеет дополнительный контекст:

```

1 (>>=) m a -> (a -> m b) -> m b
2
3 ($) :: (a -> b) -> a -> b
4 (&) :: a -> (a -> b) -> b
5 (&) = flip ($)
6 infixl 1 &
7
8 GHCi> 2 & (+3) & (*9) & (+(-3))
9 42

```

Напишем представителя **Monad** для простейшего типа **Identity**, представляющего собой простую упаковку для другого типа:

```

1 newtype Identity a = Identity { runIdentity :: a }
2
3 instance Monad Identity where
4   return x = Identity x
5   Identity x >>= k = k x

```

В стрелку Клейсли *k* передается „распакованное” значение.

```

1 return :: a -> Identity a
2 (>>=) :: Identity a -> (a -> Identity b) -> Identity b

```

Нетривиальная стрелка Клейсли:

```

1 wrap'n'succ :: Integer -> Identity Integer
2 wrap'n'succ x = Identity (succ x)
3
4 GHCi> runIdentity $ wrap'n'succ 3
5 4
6 GHCi> runIdentity $ wrap'n'succ 3 >>= wrap'n'succ
7 5
8 GHCi> runIdentity $ wrap'n'succ 3 >>= wrap'n'succ >>=
9 wrap'n'succ
10 6

```

## 19.2 Законы

Первые два закона выражают тривиальную природу **return**, третий — задает некоторое подобие ассоциативности.

```

1 return a >>= k = k a
2 m >>= return = m
3 (m >>= k) >>= k' = m >>= (\x -> k x >>= k')
4
5 GHCi> runIdentity $ wrap'n'succ 3
6 4
7 GHCi> runIdentity $ return 3 >>= wrap'n'succ
8 4
9 GHCi> runIdentity $ wrap'n'succ 3 >>= return
10 4
11
12 GHCi> runIdentity $ wrap'n'succ 3 >>= wrap'n'succ >>= w
13 rap'n'succ
14 6
15 GHCi> runIdentity $ wrap'n'succ 3 >>= (\x -> wrap'n'suc
16 c x >>= wrap'n'succ)
17 6

```

Запишем в одно выражение несколько строк из примера (по третьему закону, так можно) и прицепим **return** (по второму):

```

1 goWrap0 = wrap'n'succ 3 >>=
2           wrap'n'succ >>=
3           wrap'n'succ >>=
4           return
5 goWrap1 = wrap'n'succ 3 >>= (\x ->
6           wrap'n'succ x >>= (\y ->
7           wrap'n'succ y >>= \z ->
8           return z))
9 goWrap2 = wrap'n'succ 3 >>= (\x -> -- это примерно присваивание!
10          wrap'n'succ x >>= (\y ->
11          wrap'n'succ y >>= \z ->
12          return (x,y,z)))

```

Если нас не интересует какое-то значение можем его пропускать через >>. Для упрощения этой записи есть do-нотация:

```

1 do {e}                = e
2 do {e; stmts}          = e >> do {stmts}
3 do {p <- e; stmts}      = e >>= \p -> do {stmts}
4 do {let v = exp; stmts} = let v = exp in do {stmts}

```

Здесь все `e :: m a`, а вот `exp` может быть любым. В третьем правиле сопоставление с образцом `p` могло быть неудачным, и тогда вызывается `fail`.

```

1 goWrap3 = let i = 3 in
2           wrap'n'succ i >>= (\x ->
3           wrap'n'succ x >>= (\y ->
4           wrap'n'succ y >>
5           return (i,x,y)))
6 goWrap4 = do
7   let i = 3
8   x <- wrap'n'succ i
9   y <- wrap'n'succ x
10  wrap'n'succ y
11  return (i,x,y)

```

## 20 Класс типов `MonadFail`, его история и представители.

Вызов `fail` в современном GHC (>=8.6) ссылается на функцию из класса типов `Control.Monad.Fail.MonadFail`:

```

1 class Monad m => MonadFail m where
2   fail :: String -> m a
3
4 instance MonadFail Maybe where
5   fail _ = Nothing

```

При реализации представителей `Monad` функцию `fail` в 8.6 допустимо реализовывать, но уже бессмысленно. Предназначен для обработки неудачного сопоставления с образцом слева от `<-` в do-нотации.

```

1 GHCi> do {3 <- Just 5; return 'Z'}
2 Nothing
3 GHCi> do {3 <- Identity 5; return 'Z'}
4 error: Could not deduce (MonadFail Identity) arising from a do statement with the failable pattern `3'

```

### 20.1 Трансляция в `Kernel`

do-нотация транслируется в Haskell Kernel по-разному, в зависимости от того является ли образец „failable” или нет:

```

1 GHCi> :t do {x <- return 5; return 'Z'}
2 do {x <- return 5; return 'Z'} :: Monad m => m Char
3 GHCi> :t do {3 <- return 5; return 'Z'}
4 do {3 <- return 5; return 'Z'} :: MonadFail m => m Char
5 GHCi> :t do {~3 <- return 5; return 'Z'}
6 do {~3 <- return 5; return 'Z'} :: Monad m => m Char

```

Неопровержимые образцы не являются „failable”.

`data` с одним конструктором и `newtype` не „failable” сами по себе, но могут оказаться „failable” при вложении образцов.

```

1 GHCi> :t do {(s,x) <- return ("Answer",42); return 'Z'}
2 do {(s,x) <- return ("Answer",42); return 'Z'} :: Monad m => m Char
3
4 GHCi> :t do {(s,42) <- return ("Answer",42); return 'Z'}
5 do {(s,42) <- return ("Answer",42); return 'Z'} :: MonadFail m => m Char
6
7 GHCi> :t do {Left x <- return (Left 42); return 'Z'}
8 do {Left x <- return (Left 42); return 'Z'} :: MonadFail m => m Char

```

## 21 Стандартные монады: `Maybe` и списки.

### 21.1 `Maybe`

```

1 instance Monad Maybe where
2   return = Just
3   (Just x) >>= k = k x
4   Nothing >>= _ = Nothing
5
6   (Just _) >> m = m
7   Nothing >> _ = Nothing
8
9   fail _ = Nothing

```

Рассмотрим такую „базу данных”:

```

1 type Name = String
2 type DataBase = [(Name, Name)]
3
4 fathers, mothers :: DataBase
5 fathers =
6   [("Bill", "John"), ("Ann", "John"), ("John", "Piter")]
7 mothers =
8   [("Bill", "Jane"), ("Ann", "Jane"), ("John", "Alice"), ("Jane", "Dorothy"), ("Alice", "Mary")]
9
10 getM, getF :: Name -> Maybe Name
11 getM person = lookup person mothers
12 getF person = lookup person fathers
13
14 lookup :: Eq a => a -> [(a, b)] -> Maybe b

```

Найдем прабабушку Билла по материнской линии отца:

```

1 GHCi> getF "Bill" >>= getM >>= getM
2 Just "Mary"
3 GHCi> do { f <- getF "Bill"; m <- getM f; getM m }
4 Just "Mary"

```

Функция поиска бабушек:

```

1 granmas person = do
2   m <- getM person
3   gmm <- getM m
4   f <- getF person
5   gmf <- getM f
6   return (gmm, gmf)
7
8 GHCi> granmas "Ann"
9 Just ("Dorothy","Alice")
10 GHCi> granmas "John"
11 Nothing    -- так как есть только одна бабушка

```

## 21.2 Списки

Монада списка представляет вычисление с нулём или большим числом возможных результатов.

```

1 instance Monad [] where
2   return x = [x]
3   xs >>= k = concat (map k xs)
4   fail _ = []

```

Связывание ( $\gg=$ ) отображает стрелку  $k :: a \rightarrow [b]$  на список  $xs :: [a]$  и выполняет конкатенацию получившегося списка списков типа  $[[b]]$ .

```

1 GHCi> "abc" >>= replicate 3
2 "aaabbbccc"

```

Следующие три списка — одно и то же:

```

1 list1 = [ (x, y) | x <- [1, 2, 3], y <- [1, 2, 3], x /= y ]
2
3 list2 = do
4   x <- [1, 2, 3]
5   y <- [1, 2, 3]
6   True <- return (x /= y)
7   return (x, y)
8
9 list3 =
10   [1, 2, 3] >>= (\x ->
11     [1, 2, 3] >>= (\y ->
12       return (x /= y) >>= (\r ->
13         case r of
14           True -> return (x, y)
15           _ -> fail "Will be ignored :)"))))

```

Отличие **Monad** от **Applicative** — в монадах результат предыдущего значения может влиять на структуру последующего:

```

1 GHCi> do {a <- [1..3]; b <- [a..3]; return (a,b)}
2 [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]

```

## 22 Ввод-вывод в чистых языках. Монада IO. Взаимодействие с файловой системой.

В чистых языках, где значение функции зависит только от её параметров, ввод-вывод представляет собой проблему. Функция

```

1 getCharFromConsole :: Char

```

всегда должна возвращать одно и то же!

Как с этим справиться?

```
1 getCharFromConsole :: RealWorld -> (RealWorld, Char)
```

При этом доступ к значениям типа `RealWorld` (это то, с чем связан пользователь) должен быть ограничен.

## 22.1 IO

- Значение типа `IO` — это вычисление, которое при выполнении может осуществлять действие ввода-вывода.
- Реализация в GHC (неофициально, упрощено, меняется от версии к версии)

```
1 newtype IO a = IO (RealWorld -> (RealWorld, a))
```

- Про тип `RealWorld` в документации сказано „deeply magical“, он не экспортируется из модуля, поэтому программист не имеет к нему доступа.
- И это очень хорошо, поскольку один и тот же `RealWorld` нельзя использовать два раза!
- Единственный способ выполнить действие ввода-вывода — связать его с функцией `main` программы.

```
1 newtype IO a = IO (RealWorld -> (RealWorld, a))
2
3 unIO :: IO a -> RealWorld -> (RealWorld, a)
4 unIO (IO a) = a
5
6 instance Monad IO where
7   return x = IO $ \w -> (w, x)
8   (>=) (IO m) k = IO $ \w -> case m w of (new_w, a) -> unIO (k a) new_w
```

Гарантии, которые должны выполняться:

- Побочные эффекты происходят в заданном порядке.
- Побочный эффект каждого действия происходит один раз.

## 22.2 Функции

У нас есть примитивы `getChar`, `putChar`, про которые мы не знаем реализации, это базовые кирпичики

- Ввод:

```
1 getChar :: IO Char
2 getLine :: IO String
3 getContents :: IO String
```

- Вывод:

```
1 putChar :: Char -> IO ()
2 putStr, putStrLn :: String -> IO ()
3 print :: Show a => a -> IO ()
```

- Ввод-вывод: читаем ввод, обрабатываем, пишем ответ обратно

```
1 interact :: (String -> String) -> IO ()
```

## 22.3 Пример

`main` имеет такой же тип `IO ()`

```
1 main = do
2   putStrLn "What is your name?"
3   name <- getLine
4   putStrLn $ "Nice to meet you, " ++ name ++ "!"
```

Как, имея `getChar`, сделать `getLine`?

```
1 getLine' :: IO String
2 getLine' = do
3   c <- getChar
4   if c == '\n' then
5     return []
6   else do
7     cs <- getLine'
8     return (c:cs)
```

Как, имея `putChar`, сделать `putStr`?

```
1 putStr' :: String -> IO ()
2 putStr' [] = return ()
3 putStr' (x:xs) = putChar x >> putStr' xs
```

Можно выделить общий шаблон свёртки

```
1 sequence_ :: (Foldable t, Monad m) => t (m a) -> m ()
2 sequence_ = foldr (>>) (return ())
```

Тогда

```
1 putStr'' :: String -> IO ()
2 putStr'' = sequence_ . map putChar
```

`sequence_ . map` тоже можно обобщить:

```
1 mapM_ :: (Foldable t, Monad m) => (a -> m b) -> t a -> m ()
2 mapM_ f = sequence_ . fmap f
3
4 putStr''' :: String -> IO ()
5 putStr''' = mapM_ putChar
```

```
1 sequence :: (Traversable t, Monad m) => t (m a) -> m (t a)
2 sequence = sequenceA
3
4 mapM :: (Traversable t, Monad m) => (a -> m b) -> t a -> m (t b)
5 mapM f = sequence . fmap f -- traverse
6
7 GHCi> mapM_ putChar "Hello"
8 HelloGHCi> mapM putChar "Hello"
9 Hello[(),(),(),(),()]
```

Еще более полновесные аналоги



## 23 Монада Reader.

Вычисление, допускающее чтение значений из разделяемого окружения.

```
1 instance Monad ((->) r) where
2   return :: a -> (r -> a) -- просто игнорим окружение
3   retrum x = \_ -> x
4
5   (>=>) :: (r -> a) -> (a -> (r -> b)) -> (r -> b) -- передаем полученное окружение в оба вычисления
6   m >=> k = \e -> k (m e) e
7
8 GHCi> do {a <- (^2); b <- (*5); return (a + b)} $ 3
9 24
```

Для большей универсальности введем тип:

```
1 newtype Reader r a = Reader { runReader :: r -> a }
```

На самом деле этот тип определен по-другому, а нам интересен публичный интерфейс сборки и разборки:

```
1 reader :: (r -> a) -> Reader r a
2 runReader :: Reader r a -> r -> a
3
4 instance Monad (Reader r) where
5   return x = reader $ \_ -> x
6   m >=> k = reader $ \e -> let v = runReader m e in runReader (k v) e
```

Напишем простейший **Reader**, отчитывающийся о значении окружения:

```
1 simpleReader :: Show r => Reader r Strins
2 simpleReader = reader (\e -> "Enviroment is " ++ show e)
3
4 GHCi> runReader simpleReader 42
5 "Enviroment is 42"
```

### 23.1 Функция ask

Функция **ask :: Reader r r** возвращает окружение

```
1 type User = String
2 type Password = String
3 type UsersTable = [(User,Password)]
4
5 pwds :: UsersTable
6 pwds = [("Bill", "123"), ("Ann", "qwerty"), ("John", "2sR")]
7 firstUser = do
8   e <- ask
9   let name = fst (head e)
10  return name
11
12 GHCi> runReader firstUser pwds
13 "Bill"
```

### 23.2 Функция asks

Функция **asks :: (r -> a) -> Reader r a** возвращает результат выполнения функции над окружением:

```

1 getPwdLen :: User -> Reader UsersTable Int
2 getPwdLen person = do
3   mbPwd <- asks $ lookup person
4   let mbLen = fmap length mbPwd
5   let len = fromMaybe (-1) mbLen
6   return len
7 GHCi> runReader (getPwdLen "Ann") pws
8 6
9 GHCi> runReader (getPwdLen "Ann") []
10 -1

```

## 24 Монада `Writer`.

Это вычисление, допускающее запись в лог. В первом элементе пары лежит значение, во втором — лог.

```

1 newtype Writer w a = Writer { runWriter :: (a, w) }
2
3 writer :: (a, w) -> Writer w a
4 runWriter :: Writer w a -> (a, w)

```

Контекст `Monoid` обеспечивает полноценное конструирование логa.

```

1 instance Monoid w => Monad (Writer w) where
2   return x = writer (x, mempty)
3   m >>= k = let ~(x, u) = runWriter m -- ~ для ленивого вычисления
4             ~(y, v) = runWriter $ k x
5             in writer (y, u `mappend` v)

```

Есть два способа запустить вычисление

```

1 runWriter :: Writer w a -> (a, w)
2 execWriter :: Writer w a -> w -- если хотим только логгирование получить
3
4 GHCi> runWriter (return 3 :: Writer String Int)
5 (3, "")
6 GHCi> runWriter (return 3 :: Writer (Sum Int) Int)
7 (3, Sum {getSum = 0})
8 GHCi> execWriter (return 3 :: Writer (Product Int) Int)
9 Product {getProduct = 1}

```

База данных для интернет-магазина «Овощи-Фрукты».

```

1 type Vegetable = String
2
3 type Price = Double
4 type Qty = Double
5 type Cost = Double
6
7 type PriceList = [(Vegetable, Price)]
8 prices :: PriceList
9 prices = [("Potato", 13), ("Tomato", 55), ("Apple", 48)]

```

### 24.1 Функция `tell`

Функция `tell :: Monoid w => w -> Writer w ()` позволяет задать вывод, здесь возвращаемый `unitType ()` подчеркивает, что задача функции породить эффект, а не вернуть что-то.

```

1 addVegetable :: Vegetable -> Qty -> Writer (Sum Cost) (Vegetable, Price)
2 addVegetable veg qty = do
3   let pr = fromMaybe 0 $ lookup veg prices
4   let cost = qty * pr
5   tell $ Sum cost
6   return (veg, pr)
7
8 GHCi> runWriter $ addVegetable "Apple" 100
9 (("Apple",48.0), Sum {getSum = 4800.0})
10 GHCi> runWriter $ addVegetable "Pear" 100
11 (("Pear",0.0), Sum {getSum = 0.0})

```

## 24.2 Функция listen

Если хотим знать промежуточные стоимости, используем

```
listen :: Monoid w => Writer w a -> Writer w (a, w)
```

```

1 myCart1 :: Writer (Sum Cost) [((Vegetable, Price), Sum Cost)]
2 myCart1 = do
3   x1 <- listen $ addVegetable "Potato" 3.5
4   x2 <- listen $ addVegetable "Tomato" 1.0
5   x3 <- listen $ addVegetable "AGRH!!" 1.6
6   return [x1,x2,x3]
7
8 GHCi> runWriter myCart1
9 ([((("Potato",13.0),Sum {getSum = 45.5}),(("Tomato",55.0),
10 Sum {getSum = 55.0}),(("AGRH!!",0.0),Sum {getSum = 0.0})),
11 Sum {getSum = 100.5})]

```

## 24.3 Функция listens

Есть более гибкая альтернатива, можем обрабатывать то, что выводим

```
listens :: Monoid w => (w -> b) -> Writer w a -> Writer w (a, b)
```

```

1 myCart1' :: Writer (Sum Cost) [((Vegetable, Price), Cost)]
2 myCart1' = do
3   x1 <- listens getSum $ addVegetable "Potato" 3.5
4   x2 <- listens getSum $ addVegetable "Tomato" 1.0
5   x3 <- listens getSum $ addVegetable "AGRH!!" 1.6
6   return [x1,x2,x3]
7
8 GHCi> runWriter myCart1'
9 ([((("Potato",13.0),45.5),(("Tomato",55.0),55.0),(("AGRH!!",0.0),0.0)),Sum {getSum = 100.5})]

```

## 24.4 Функция censor

Для модификации лога используем

```
censor :: Monoid w => (w -> w) -> Writer w a -> Writer w a
```

```

1 myCart0' :: Writer (Sum Cost) [(Vegetable, Price)]
2 myCart0' = censor (discount 10) myCart0
3
4 discount :: Double -> Sum Cost -> Sum Cost
5 discount proc s@(Sum x)
6   | x < 100 = s
7   | x >= 100 = Sum $ x * (100 - proc) / 100
8
9 GHCi> execWriter myCart0
10 Sum {getSum = 100.5}

```

```

11 GHCi> execWriter myCart0'
12 Sum {getSum = 90.45}

```

## 25 Монада State.

Вычисление, позволяющее работать с изменяемым состоянием.

```

1 newtype State s a = State { runState :: s -> (a, s) }
2
3 state :: (s -> (a, s)) -> State s a
4 runState :: State s a -> s -> (a, s)
5
6 instance Monad (State s) where
7   return x = state $ \st -> (x, st)
8   m >>= k = state $ \st -> let (x, st') = runState m st
9                               m' = k x
10                              in runState m' st'

```

`return` упаковывает значение в функцию, не меняющую состояние. `(>>=)` передает обновленное первым вычислением состояние во второе вычисление.

Помимо пары, в вычислении с состоянием можно получить либо только итоговое состояние, либо только итоговое значение вычисления:

```

1 runState :: State s a -> s -> (a,s)
2 execState :: State s a -> s -> s
3 evalState :: State s a -> s -> a
4
5 GHCi> runState (return 3 :: State String Int) "Hi, State!"
6 (3, "Hi, State!")
7 GHCi> execState (return 3 :: State String Int) "Hi, State!"
8 "Hi, State!"
9 GHCi> evalState (return 3 :: State String Int) "Hi, State!"
10 3

```

### 25.1 Функции стандартного интерфейса

```

1 get :: State s s -- ask
2 get = state $ \s -> (s, s)
3
4 put s :: s -> State s () -- tell, но мы портим прошлое состояние
5 put = state $ \_ -> ((), s)
6
7 modify :: (s -> s) -> State s () -- обработка
8 modify f = do s <- get
9             put (f s)
10
11 gets :: (s -> a) -> State s a
12 gets f = do s <- get
13         return (f s)

```

### 25.2 Примеры

```

1 tick :: State Int Int -- увеличиваем состояние на 1, возвращаемое значение оставляем
2 tick = do n <- get
3         put (n + 1)
4         return n
5 GHCi> runState tick 3
6 (3,4)

```

```

7
8 succ' :: Int -> Int
9 succ' n = execState tick n
10
11 plus :: Int -> Int -> Int
12 plus n x = execState (sequence $ replicate n tick) x

```

Конструкция `sequence . replicate n` встречается довольно часто, поэтому в `Control.Monad` определено

```

1 replicateM :: Applicative m => Int -> m a -> m [a] -- (Раньше контекст был Monad m)
2 replicateM n = sequenceA . replicate n
3
4 plus' :: Int -> Int -> Int
5 plus' n x = execState (replicateM n tick) x

```

## Другие инструменты для работы с состоянием

- `State` это чистая функциональная конструкция.
- Монада `ST` позволяет локально работать с настоящим изменяемым состоянием. Имеется удобный вспомогательный ссылочный тип `STRef`. Локальность обеспечивается типом второго ранга  
`runST :: (forall s. ST s a) -> a`
- `IORef` это `STRef` без локальности и соответствующих гарантий безопасности.
- `MVar` это `IORef` с блокировками, поддерживающими конкурентный доступ.
- `TVar` это изменяемые ячейки памяти в рамках `STM` (Software transactional memory).

## 26 Монада `Except`.

Вычисление, допускающее возбуждение и перехват исключений.

```

1 newtype Except e a = Except {runExcept :: Either e a} -- как обычно такого на самом деле нет
2
3 except :: Either e a -> Except e a
4 except = Except
5
6 instance Monad (Except e) where
7   return a = Except $ Right a
8   m >>= k = case runExcept m of
9     Left e -> Except $ Left e
10    Right x -> k x

```

### 26.1 Стандартный интерфейс

```

1 throwE :: e -> Except e a
2 throwE = except . Left
3
4 catchE :: Except e a -> (e -> Except e' a) -> Except e' a
5 m `catchE` h = case runExcept m of
6   Left l -> h l
7   Right r -> Except $ Right r

```

## 26.2 Использование

```
1 do { action1; action2; action3 } `catchE` handler
```

В библиотеке mtl по историческим причинам `throwError` и `catchError`.

## 26.3 Примеры

```
1 data DivByError = ErrZero | Other String deriving (Eq,Show)
2
3 (/?) :: Double -> Double -> Except DivByError Double
4 _ /? 0 = throwE ErrZero
5 x /? y = return $ x / y
6
7 example0 :: Double -> Double -> Except DivByError String
8 example0 x y = action `catchE` handler where
9   action = do q <- x /? y
10   return $ show q
11   handler = return . show
12 GHCi> runExcept $ example0 5 2
13 Right "2.5"
14 GHCi> runExcept $ example0 5 0
15 Right "ErrZero"
```

Можем захотеть расширить функциональность, работать с ошибками хитрее, можем использовать функциональность `MonadPlus`, то есть `guard`, `msum`, `mfilter`, для этого нужно сделать `Except` е представителем: (это, например, ошибка по умолчанию, операция накапливания ошибок)

```
1 instance Monoid e => MonadPlus (Except e) where
2   mzero = Except $ Left mempty
3   Except x `mplus` Except y = Except $
4     case x of
5       Left e -> either (Left . mappend e) Right y
6       r -> r
```

Семантика:

- `mzero` — ошибка по умолчанию для `guard`, задается `mempty`;
- `mplus` — накапливает ошибки слева направо, но если происходит удачная попытка, то возвращает удачу.

```
1 instance Monoid DivByError where
2   mempty = Other ""
3   Other s1 `mappend` Other s2 = Other $ s1 ++ s2
4   Other s1 `mappend` ErrZero = Other $ s1 ++ "zero;"
5   ErrZero `mappend` Other s2 = Other $ "zero;" ++ s2
6   ErrZero `mappend` ErrZero = Other $ "zero;zero;"
7
8 example2 :: Double -> Double -> Except DivByError String
9 example2 x y = action `catchE` handler where
10   action = do
11     q <- x /? y
12     guard $ y >= 0
13     return $ show q
14   handler = \err -> return $ show err
```

Посмотрим, как это работает

```
1 GHCi> runExcept $ example2 5 0
2 Right "ErrZero"
3 GHCi> runExcept $ example2 5 (-2)
4 Right "Other \"\"\"
5 GHCi> runExcept $ msum [5/?0, 7/?0, 2/?0]
6 Left (Other "zero;zero;zero;")
7 GHCi> runExcept $ msum [5/?0, 7/?0, 2/?4]
8 Right 0.5
```