

Практика 6. Классы типов

осень 2026

Андрей Стоян, Илья Колегов

осень 2026

Летучка 6

Летучка 6 · 7 минут · без конспектов · Ваше ФИ: _____

1. `data Choice = A Int | B Int Int | C.`

Результат `case B 1 2 of { A x -> x + 3; B x y -> x + y; C -> 42 }`: _____

2. Объявление `Either`: _____

3. Существует ли значение типа `Maybe` без типового аргумента? _____

4. Наиболее общий тип `Left 42`: _____

5. Результат `map not [True, False]`: _____

Простые классы типов

Классы типов как механизм перегрузки

Перегрузки C++: одно имя, две конкретные функции.

Int

```
string show(int x) { ... }
```

Data

```
string show(data d) { ... }
```

Классы типов как механизм перегрузки

Общая сигнатура: подставляем тип и узнаём знакомую перегрузку.

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

a = Int

```
string show(int x) { ... }
```

```
show :: Int -> String
```

a = Data

```
string show(data d) { ... }
```

```
show :: Data -> String
```

Классы типов как механизм перегрузки

Конкретная реализация становится методом соответствующего инстанса.

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

a = Int

```
instance Show Int where  
  show :: Int -> String  
  show x = ...
```

a = Data

```
instance Show Data where  
  show :: Data -> String  
  show d = ...
```

Как тип аргумента определяет нужную реализацию?

Классы типов как механизм перегрузки

Тип аргумента `Int` выбирает реализацию из `instance Show Int`.

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

a = Int

```
instance Show Int where  
  show :: Int -> String  
  show x = ...
```

a = Data

```
instance Show Data where  
  show :: Data -> String  
  show d = ...
```

```
show (3 :: Int)
```

Классы типов как механизм перегрузки

Тип аргумента `Data` выбирает другую реализацию для того же символа `show`.

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

a = Int

```
instance Show Int where  
  show :: Int -> String  
  show x = ...
```

```
show (3 :: Int)
```

```
show (d :: Data)
```

a = Data

```
instance Show Data where  
  show :: Data -> String  
  show d = ...
```

Задачи: свой инстанс Show

✎ Допишите инстанс: конструкторы отображаются строками "ERROR", "WARNING", "DEBUG", "INFO".

```
data LogLevel = Error | Warning | Debug | Info

instance Show LogLevel where
  show Error    = "ERROR"
  show Warning  = -----
  show Debug    = -----
  show Info     = -----
```

⚡ **Решение.**

```
instance Show LogLevel where
  show Error    = "ERROR"
  show Warning  = "WARNING"
  show Debug    = "DEBUG"
  show Info     = "INFO"
```

Что требуется от типового параметра?

✎ Допишите ограничение в сигнатуре. Предскажите результат вызова.

```
showWithPrefix :: _____ => a -> String -> String
showWithPrefix x prefix = prefix ++ show x

showWithPrefix Warning "level: "
```

⚡ **Решение.**

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

Результат: "level: WARNING".

⚠ `Show a =>` — требование: для типа `a` должна быть доступна реализация `Show`.

Задачи: инстанс для пары

✎ Напишите целиком инстанс `Show` для пары `(a, b)`, включая необходимые ограничения. Формат результата: `"(True,3)"` для `(True, 3 :: Int)`.

⚡Решение.

```
instance (Show a, Show b) => Show (a, b) where
  show (x, y) = "(" ++ show x ++ "," ++ show y ++ ")"
```

Задачи: свой класс

✎ Придумайте и напишите целиком класс для парсинга данных `MyRead`. Его метод должен принимать строку и возвращать разобранное значение либо отсутствие результата при неудаче.

⚡ Решение.

```
class MyRead a where  
  readMaybe :: String -> Maybe a
```

Варьируемый типовой параметр может использоваться где угодно.

Все значения типа

✎ Напишите сигнатуру и реализацию `allValues`, перебирающей все значения конечного перечислимого типа. Какие классы нужны для границ и перехода к следующему значению?

⚡ *Решение.*

```
allValues :: (Enum a, Bounded a) => [a]
allValues = [minBound .. maxBound]
```

`allValues :: [Bool]` даёт `[False, True]`.

Равенство функций на конечной области

✎ Реализуйте проверку равенства функций: напишите целиком инстанс `Eq` для `(a -> b)` с необходимыми ограничениями. Область входов конечная и небольшая; используйте `allValues`.

⚡ Решение.

```
instance (Enum a, Bounded a, Eq b) => Eq (a -> b) where
    (==) :: (a -> b) -> (a -> b) -> Bool
    f == g = map f allValues == map g allValues
```

NB Полный перебор полезен лишь для небольшой конечной области, например `Bool`. Для `Int` такой способ непрактичен.

Один класс требует другой

✎ **Устно:** Нужны $x == y$ и $x < y$. Достаточно ли `Ord a` или нужно `(Eq a, Ord a)`? Проверьте по объявлению класса.

⚡ **Решение.**

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  -- ...
```

`Ord a` уже требует `Eq a` — отдельно повторять его не нужно.

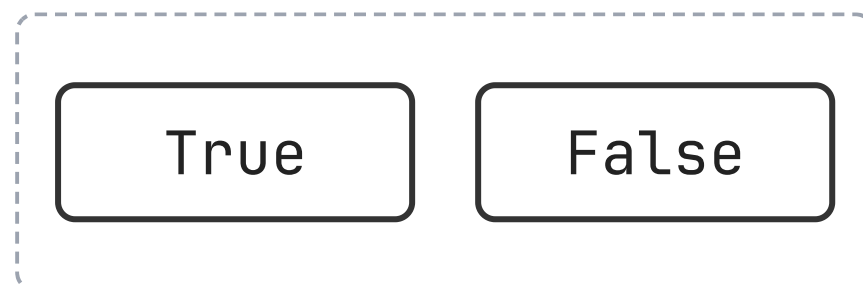
Классы типов высших кайндов

map — одна идея для разных структур

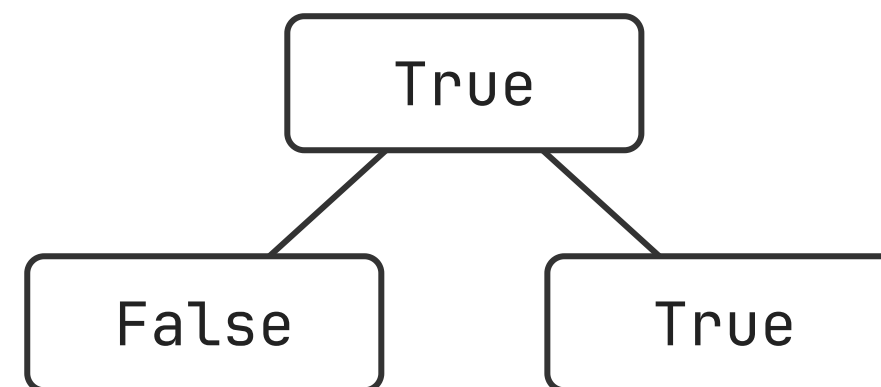
Знакомый `map`, `transform` из П5 и преобразование значения внутри `Maybe`.

```
map      :: (a -> b) -> [a] -> [b]
transform :: (a -> b) -> Tree a -> Tree b
mapMaybe :: (a -> b) -> Maybe a -> Maybe b
```

СПИСОК



дерево из П5



Maybe

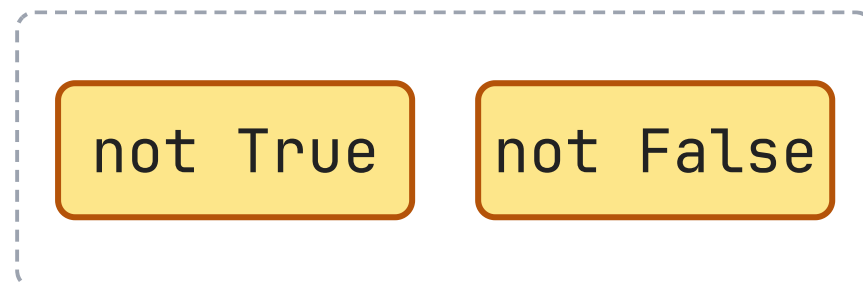


map — одна идея для разных структур

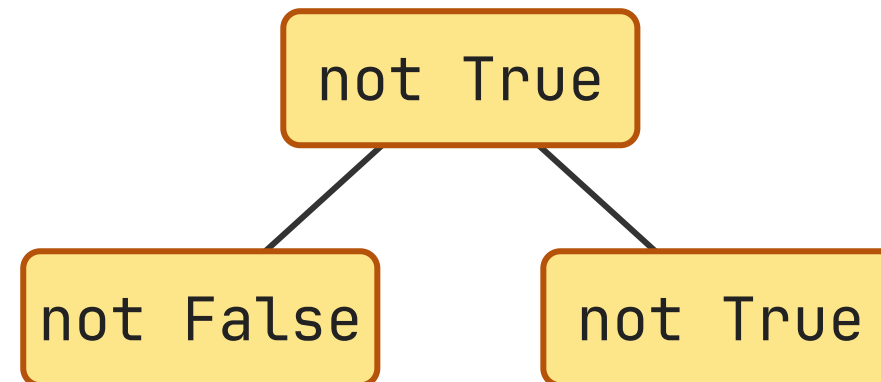
В каждой структуре применяем `not` к значениям; форма структуры сохраняется.

```
map      :: (a -> b) -> [a] -> [b]
transform :: (a -> b) -> Tree a -> Tree b
mapMaybe :: (a -> b) -> Maybe a -> Maybe b
```

СПИСОК



дерево из П5



Maybe

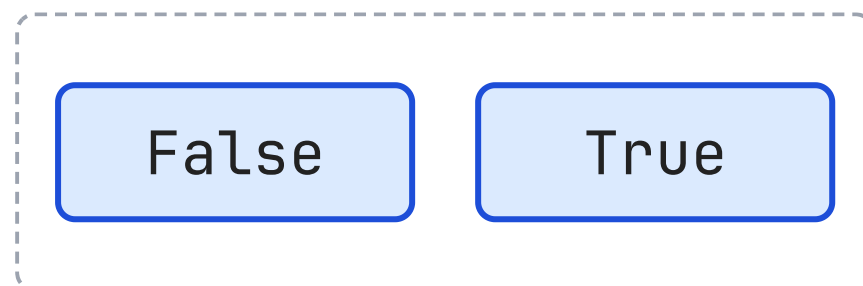


map — одна идея для разных структур

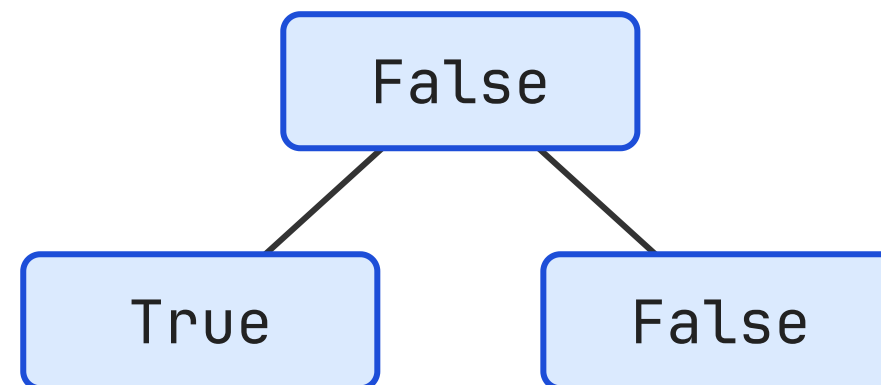
Одно имя `fmap`: конкретный конструктор типа выбирает реализацию.

```
fmap      :: (a -> b) -> [a] -> [b]
fmap      :: (a -> b) -> Tree a -> Tree b
fmap      :: (a -> b) -> Maybe a -> Maybe b
```

СПИСОК



дерево из П5



Maybe

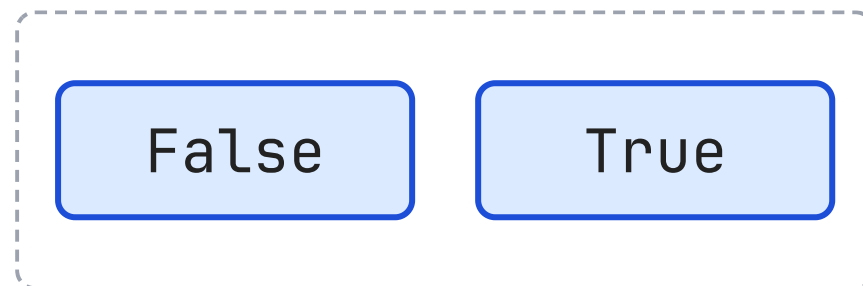


map — одна идея для разных структур

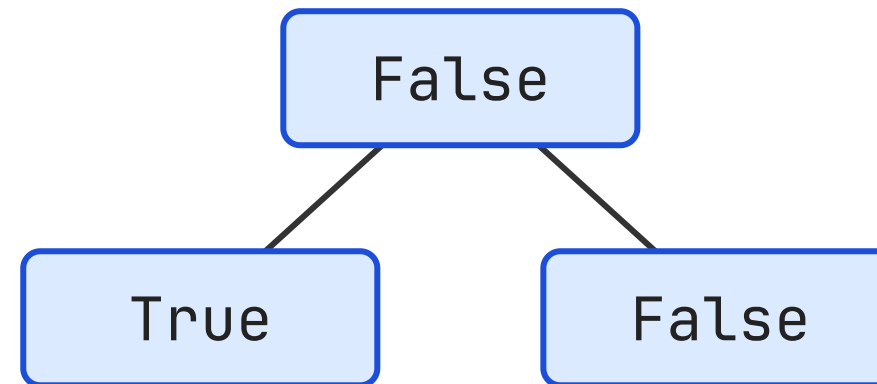
Functor объединяет эти операции: параметр **f** задаёт структуру.

```
class Functor (f :: * -> *) where  
  fmap :: (a -> b) -> f a -> f b
```

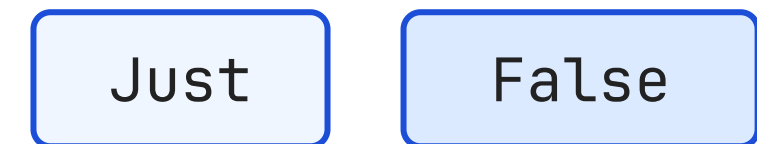
f = []



f = Tree



f = Maybe



fmap: меняется значение, остаётся структура

✎ **Устно:** Предскажите `fmap show (Just True)` и `fmap show Nothing`. Что должно сохраняться?

`fmap show (Just True)`

Just

структура

True

значение типа Bool

fmap: меняется значение, остаётся структура

Функция `show` применяется к значению внутри `Just`: из `Bool` получается `String`.

```
Just (show True)
```

Just

show True

структура

преобразуем только значение

fmap: меняется значение, остаётся структура

Конструктор **Just** сохраняется; значение и его тип меняются.

Just "True"

Just

структура

"True"

результат типа String

Функторы: от разбора к пропускам

e.g. Разбор `Maybe`:

```
instance Functor Maybe where
  fmap f Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

 Допишите инстанс списка по его конструкторам.

```
instance Functor [] where
  fmap f []      = -----
  fmap f (x:xs) = -----
```

 **Решение.**

```
instance Functor [] where
  fmap f []      = []
  fmap f (x:xs) = f x : fmap f xs
```

Это `map`: можно написать `fmap = map`.

Частичное применение типовых конструкторов

✎ Определите кайнды `(,)`, `(->)` и `(,) Int`. Реализуйте функтор для пары.

⚡ *Решение.*

```
(,) :: * -> * -> *  
(->) :: * -> * -> *  
(,) Int :: * -> *
```

⚡ *Решение.*

```
instance Functor ((,) e) where -- (,) :: * -> * -> *  
  fmap :: (a -> b) -> ((,) e) a -> ((,) e) b  
  --    :: (a -> b) -> (e, a)      -> (e, b)  
  fmap f (w, x) = (w, f x)
```

Что теперь умеем

- Использовать один символ для разных типов
- Определять классы типов и их инстансы
- Использовать частичное применение типовых конструкторов

Реализация классов типов в GHC

⚙ Полиморфная сортировка: базовые классы типов

Хотим сортировать элементы разных типов. Как их сравнивать?

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

⚙ Полиморфная сортировка: базовые классы типов

Передаём сравнение явно: для каждого типа — своя реализация.

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

⚙ Полиморфная сортировка: базовые классы типов

Объединяем операции в запись — словарь.

```
data MyOrd a = MyOrd { less :: a -> a -> Bool }  
sort :: MyOrd a -> [a] -> [a]
```

```
intMyOrd :: MyOrd Int  
intMyOrd = MyOrd { less = (<) }
```

```
sort intMyOrd [3,2,1]
```

Результат: [1,2,3].

⚙ Свойство упорядоченности

Через `data`

1. Определение словаря функций.

```
data MyOrd a = MyOrd
  { less :: a -> a -> Bool }
```

2. Экземпляр словаря для конкретного типа — именованное значение.

```
intMyOrd :: MyOrd Int
intMyOrd = MyOrd { less = (<) }
```

3. Явный параметр функции.

```
sort :: MyOrd a -> [a] -> [a]
```

4. Передаётся пользователем.

```
test = sort intMyOrd [3, 2, 1]
```

1. Нужен импорт модуля, даже пустой, с определением нужного инстанса: `import Module ()`.

Через `class`

1. Определение класса типов.

```
class MyOrd a where
  less :: a -> a -> Bool
```

2. Объявление типа представителем класса — не имеет имени.

```
instance MyOrd Int where
  less = (<)
```

3. Неявный параметр функции.

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

4. Передаётся компилятором.¹

```
test = sort [3, 2, 1]
```

⚙ От словаря элемента к словарю списка

Строитель словаря и зависимый инстанс выражают одно требование.

Явно

```
listMyOrd :: MyOrd a -> MyOrd [a]
listMyOrd aOrd = MyOrd
  { less = ... }
```

Неявно

```
instance MyOrd a => MyOrd [a] where
  less = ...
```

```
sort (listMyOrd intMyOrd) [[3,2], [], [1]]
```

⇒ `intMyOrd` даёт сравнение элементов; `listMyOrd intMyOrd` — сравнение списков.

⚙ От словаря элемента к словарю списка

Строитель словаря и зависимый инстанс выражают одно требование.

Явно

```
listMyOrd :: MyOrd a -> MyOrd [a]
listMyOrd aOrd = MyOrd
  { less = ... }
```

Неявно

```
instance MyOrd a => MyOrd [a] where
  less = ...
```

```
sort [[3,2], [], [1 :: Int]]
```

⇒ Компилятор ищет `MyOrd [Int]`: нужен `MyOrd Int`, затем инстанс списка.

Материалы

Частичный и полный порядок

Иногда одно свойство влечёт другое как более слабое. В Haskell инстанс расширенного класса типов просто требует наличия инстанса базового.

Пример: частичный порядок — работает не для каждой пары элементов.

```
class MyPartialOrd a where
  mbLess :: a -> a -> Maybe Bool
```

Частичный порядок обязателен для полного.¹²

```
class MyPartialOrd a => MyOrd a where
  less :: a -> a -> Bool
```

При объявлении типа представителем базового класса можно пользоваться объявлением для расширенного.

```
instance MyPartialOrd Int where
  mbLess x y = Just (less x y)
```

1. Например, числа с плавающей точкой по стандарту IEEE 754 упорядочены только частично (NaN).
2. В Haskell так не сделано, но сделано в Rust. Менее удобно, но более безопасно.

Похожие концепции в других языках

- А ля классы типов, диспатч по нулевому аргументу функций:
 - e.g. Rust traits;
 - e.g. Swift protocols;
 - e.g. Scala traits + implicits.
- Контроль за инстанциацией шаблонов:
 - e.g. C++20 концепты.
- Неявные ресиверы:
 - e.g. Kotlin context receivers.
- Шаблон visitor для динамического диспатча по нескольким аргументам одновременно.
- Мультиметоды:
 - e.g. Julia.

Полезные ссылки

[Typeclassopedia: шпаргалка по зоопарку классов типов](#)

Серьёзные материалы

- Tagless-Final Style
- Олег Нижников — Современное ФП с Tagless Final
- The Death of Tagless Final by John A. De Goes
- Review: Codata in Action
- Tagless Final style by Oleg
- Lecture notes on Tagless Final
- Philip Wadler, Stephen Blott. *How to make ad-hoc polymorphism less ad hoc*. POPL, 1989.