Лекция 5: функции как значения

Функциональное программирование на Haskell

Алексей Романов
6 марта 2023 г.
МИЭТ

Функции как значения

- Как упоминалось в начале курса, одно из оснований ФП состоит в том, что функции могут использоваться как значения.
- В Haskell можно выразиться сильнее:

Функции как значения

- Как упоминалось в начале курса, одно из оснований ФП состоит в том, что функции могут использоваться как значения.
- В Haskell можно выразиться сильнее:
- Функции это и есть просто значения, тип которых имеет форму `ТипПараметра -> ТипРезультата` для каких-то `ТипПараметра` и `ТипРезультата`.

Функции как значения

- Как упоминалось в начале курса, одно из оснований ФП состоит в том, что функции могут использоваться как значения.
- В Haskell можно выразиться сильнее:
- Функции это и есть просто значения, тип которых имеет форму `ТипПараметра -> ТипРезультата` для каких-то `ТипПараметра` и `ТипРезультата`.
- Мы уже видели примеры этого в равноправии функций и переменных.

Функции как значения

- Как упоминалось в начале курса, одно из оснований ФП состоит в том, что функции могут использоваться как значения.
- В Haskell можно выразиться сильнее:
- Функции это и есть просто значения, тип которых имеет форму `ТипПараметра -> ТипРезультата` для каких-то `ТипПараметра` и `ТипРезультата`.
- Мы уже видели примеры этого в равноправии функций и **других** переменных.

Функции высших порядков

- В частности, функции могут принимать на вход функции.
- То есть тип параметра сам может быть функциональным типом.
- Тривиальный пример:

```
foo :: (Char -> Bool) -> Bool
foo f = f 'a'

ghci> foo isLetter
True
```

- Скобки вокруг типа параметра здесь необходимы.
- Функции, параметры которых — функции, называются *функциями высших порядков (ФВП)*.
- Часто ими также считают функции, возвращающие функции, но не в Haskell (скоро увидим почему).

Функции высших порядков

- В частности, функции могут принимать на вход функции.
- То есть тип параметра сам может быть функциональным типом.
- Тривиальный пример:

```haskell
foo :: (Char -> Bool) -> Bool
foo f = f 'a'

ghci> foo isLetter
```

Функции высших порядков

- В частности, функции могут принимать на вход функции.
- То есть тип параметра сам может быть функциональным типом.
- Тривиальный пример:

```haskell
foo :: (Char -> Bool) -> Bool
foo f = f 'a'

ghci> foo isLetter
True
```

- Скобки вокруг типа параметра здесь необходимы.

Функции высших порядков

- В частности, функции могут принимать на вход функции.
- То есть тип параметра сам может быть функциональным типом.
- Тривиальный пример:

```
foo :: (Char -> Bool) -> Bool
foo f = f 'a'

ghci> foo isLetter
True
```

- Скобки вокруг типа параметра здесь необходимы.
- Функции, параметры которых — функции, называются *функциями высших порядков (ФВП)*.

Функции высших порядков

- В частности, функции могут принимать на вход функции.
- То есть тип параметра сам может быть функциональным типом.
- Тривиальный пример:

```
foo :: (Char -> Bool) -> Bool
foo f = f 'a'

ghci> foo isLetter
True
```

- Скобки вокруг типа параметра здесь необходимы.
- Функции, параметры которых — функции, называются *функциями высших порядков (ФВП)*.
- Часто ими также считают функции, возвращающие функции, но не в Haskell (скоро увидим почему).

Лямбда-выражения

- В `foo` с прошлого слайда можем также передать своюю новую функцию, определив её локально:

```
foo isCyrillic where
    isCyrillic c = let lc = toLower c
                   in 'a' <= lc && lc <= 'я'
```

Лямбда-выражения

- В `foo` с прошлого слайда можем также передать свою новую функцию, определив её локально:
  ```
  foo isCyrillic where
      isCyrillic c = let lc = toLower c
                     in 'a' <= lc && lc <= 'я'
  ```
- Но имя этой функции на самом деле не нужно.
- Как и вообще функциям, которые создаются только как аргументы для других (или как результаты).

Лямбда-выражения

- В `foo` с прошлого слайда можем также передать свою новую функцию, определив её локально:

```
foo isCyrillic where
    isCyrillic c = let lc = toLower c
                   in 'a' <= lc && lc <= 'я'
```

- Но имя этой функции на самом деле не нужно.
- Как и вообще функциям, которые создаются только как аргументы для других (или как результаты).
- Вместо этого зададим её через *лямбда-выражение*

```
foo (\c -> let lc = toLower c
           in 'a' <= lc && lc <= 'я')
```

Лямбда-выражения

- В `foo` с прошлого слайда можем также передать свою новую функцию, определив её локально:

```
foo isCyrillic where
    isCyrillic c = let lc = toLower c
                   in 'a' <= lc && lc <= 'я'
```

- Но имя этой функции на самом деле не нужно.
- Как и вообще функциям, которые создаются только как аргументы для других (или как результаты).
- Вместо этого зададим её через *лямбда-выражение*

```
foo (\c -> let lc = toLower c
           in 'a' <= lc && lc <= 'я')
```

- Кстати, почему это определение неверно?

Лямбда-выражения

- В `foo` с прошлого слайда можем также передать свою новую функцию, определив её локально:

```
foo isCyrillic where
    isCyrillic c = let lc = toLower c
                   in 'а' <= lc && lc <= 'я'
```

- Но имя этой функции на самом деле не нужно.
- Как и вообще функциям, которые создаются только как аргументы для других (или как результаты).
- Вместо этого зададим её через *лямбда-выражение*

```
foo (\c -> let lc = toLower c
           in 'а' <= lc && lc <= 'я')
```

- Кстати, почему это определение неверно?
- Как ни странно, в Юникоде `'ё' > 'я'`.
- И не все буквы кириллицы используются в русском.

Лямбда-выражения

- Вообще, два определения

  ```
  функция образец = результат

  функция = \образец -> результат
  ```

  эквивалентны.

Лямбда-выражения

- Вообще, два определения

  ```
  функция образец = результат

  функция = \образец -> результат
  ```

  эквивалентны.
- Одно исключение: для второго может быть выведен менее общий тип.

Лямбда-выражения

- Вообще, два определения

```
функция образец = результат

функция = \образец -> результат
```

эквивалентны.

- Одно исключение: для второго может быть выведен менее общий тип.
- Лямбда-выражение для функции с несколькими параметрами пишется

```
\образец1 ... образецN -> результат
```

Лямбда-выражения

- Вообще, два определения

  ```
  функция образец = результат

  функция = \образец -> результат
  ```

  эквивалентны.
- Одно исключение: для второго может быть выведен менее общий тип.
- Лямбда-выражение для функции с несколькими параметрами пишется

  ```
  \образец1 ... образецN -> результат
  ```
- Например,

  ```
  Data.List.sortBy (\x y -> compare y x) list
  ```
- Что делает это выражение?

Лямбда-выражения

- Вообще, два определения

  ```
  функция образец = результат

  функция = \образец -> результат
  ```

  эквивалентны.
- Одно исключение: для второго может быть выведен менее общий тип.
- Лямбда-выражение для функции с несколькими параметрами пишется

  ```
  \образец1 ... образецN -> результат
  ```
- Например,

  ```
  Data.List.sortBy (\x y -> compare y x) list
  ```
- Что делает это выражение?
- Сортирует список по убыванию.

Лямбда-выражения и case

- Если в обычном определении функции несколько уравнений, например

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

  то в лямбда-выражении придётся использовать `case`:

Лямбда-выражения и case

- Если в обычном определении функции несколько уравнений, например

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

  то в лямбда-выражении придётся использовать case:

  ```
  not = \x -> case x of
      True  -> False
      False -> True
  ```

Лямбда-выражения и case

- Если в обычном определении функции несколько уравнений, например

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

то в лямбда-выражении придётся использовать `case`:

```
not = \x -> case x of
    True  -> False
    False -> True
```

или с расширением `LambdaCase`

```
not = \case
    True  -> False
    False -> True
```

Несколько стандартных ФВП

- В `Prelude` есть три функции второго порядка, которые очень часто встречаются в коде Haskell:

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

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

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

- И ещё две в `Data.Function`:

```haskell
(&) = flip (.)
(&) :: (a -> b) -> (b -> c) -> (a -> c)

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

Несколько стандартных ФВП

- В `Prelude` есть три функции второго порядка, которые очень часто встречаются в коде Haskell:

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

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

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

- И ещё две в `Data.Function`:

```
(&) = flip (.)
(&) :: (a -> b) -> (b -> c) -> (a -> c)

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

Несколько стандартных ФВП

- В `Prelude` есть три функции второго порядка, которые очень часто встречаются в коде Haskell:

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

Несколько стандартных ФВП

- В `Prelude` есть три функции второго порядка, которые очень часто встречаются в коде Haskell:

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

Несколько стандартных ФВП

- В `Prelude` есть три функции второго порядка, которые очень часто встречаются в коде Haskell:

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

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

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

- И ещё две в `Data.Function`:

```
(&) = flip (.)
(&) :: (a -> b) -> (b -> c) -> (a -> c)

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

Несколько стандартных ФВП

- В `Prelude` есть три функции второго порядка, которые очень часто встречаются в коде Haskell:

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

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

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

- И ещё две в `Data.Function`:

```
(&) = flip (.)
(&) :: (a -> b) -> (b -> c) -> (a -> c)

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

Несколько стандартных ФВП

- В `Prelude` есть три функции второго порядка, которые очень часто встречаются в коде Haskell:

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

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

Несколько стандартных ФВП

- В `Prelude` есть три функции второго порядка, которые очень часто встречаются в коде Haskell:

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

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

Несколько стандартных ФВП

- В `Prelude` есть три функции второго порядка, которые очень часто встречаются в коде Haskell:

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

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

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

- И ещё две в `Data.Function`:

```
(&) = flip (.)
(&) :: (a -> b) -> (b -> c) -> (a -> c)

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

Несколько стандартных ФВП

- В `Prelude` есть три функции второго порядка, которые очень часто встречаются в коде Haskell:

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

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

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

- И ещё две в `Data.Function`:

```
(&) = flip (.)
(&) :: (a -> b) -> (b -> c) -> (a -> c)

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

Несколько стандартных ФВП

- В `Prelude` есть три функции второго порядка, которые очень часто встречаются в коде Haskell:

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

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

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

Несколько стандартных ФВП

- В `Prelude` есть три функции второго порядка, которые очень часто встречаются в коде Haskell:

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

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

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

Несколько стандартных ФВП

- В `Prelude` есть три функции второго порядка, которые очень часто встречаются в коде Haskell:

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

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

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

- И ещё две в `Data.Function`:

```
(&) = flip (.)
(&) :: (a -> b) -> (b -> c) -> (a -> c)

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

Несколько стандартных ФВП

- В `Prelude` есть три функции второго порядка, которые очень часто встречаются в коде Haskell:

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

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

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

- И ещё две в `Data.Function`:

```
(&) = flip (.)
(&) ::
```

Несколько стандартных ФВП

- В `Prelude` есть три функции второго порядка, которые очень часто встречаются в коде Haskell:

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

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

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

- И ещё две в `Data.Function`:

```
(&) = flip (.)
(&) :: (a -> b) -> (b -> c) -> (a -> c)

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

Избавление от скобок с помощью `$` и `.`

- Казалось бы, в чём смысл `$`: зачем писать `f $ x` вместо `f x`?

Избавление от скобок с помощью `$` и `.`

- Казалось бы, в чём смысл `$`: зачем писать `f $ x` вместо `f x`?
- Этот оператор имеет минимальный возможный приоритет, так что `f (x + y)` можно записать как `f $ x + y`.

Избавление от скобок с помощью `$` и `.`

- Казалось бы, в чём смысл `$`: зачем писать `f $ x` вместо `f x`?
- Этот оператор имеет минимальный возможный приоритет, так что `f (x + y)` можно записать как `f $ x + y`.
- И он правоассоциативен, так что `f (g (h x))` можно записать как `f $ g $ h x`.

Избавление от скобок с помощью `$` и `.`

- Казалось бы, в чём смысл `$`: зачем писать `f $ x` вместо `f x`?
- Этот оператор имеет минимальный возможный приоритет, так что `f (x + y)` можно записать как `f $ x + y`.
- И он правоассоциативен, так что `f (g (h x))` можно записать как `f $ g $ h x`.
- Но более принято `f . g . h $ x`.

Избавление от скобок с помощью `$` и `.`

- Казалось бы, в чём смысл `$`: зачем писать `f $ x` вместо `f x`?
- Этот оператор имеет минимальный возможный приоритет, так что `f (x + y)` можно записать как `f $ x + y`.
- И он правоассоциативен, так что `f (g (h x))` можно записать как `f $ g $ h x`.
- Но более принято `f . g . h $ x`.
- `.` тоже правоассоциативен, но уже с максимальным приоритетом.

Избавление от скобок с помощью `$` и `.`

- Казалось бы, в чём смысл `$`: зачем писать `f $ x` вместо `f x`?
- Этот оператор имеет минимальный возможный приоритет, так что `f (x + y)` можно записать как `f $ x + y`.
- И он правоассоциативен, так что `f (g (h x))` можно записать как `f $ g $ h x`.
- Но более принято `f . g . h $ x`.
- `.` тоже правоассоциативен, но уже с максимальным приоритетом.

- Пока, наверное, проще читать и писать код со скобками, но стандартный стиль Haskell предпочитает их избегать.
- Только не перестарайтесь!

Правда о функциях многих переменных

- Настала пора раскрыть тайну функций многих переменных в Haskell:

Правда о функциях многих переменных

- Настала пора раскрыть тайну функций многих переменных в Haskell:
- Их не существует.

Правда о функциях многих переменных

- Настала пора раскрыть тайну функций многих переменных в Haskell:
- Их не существует.
- `->` — правоассоциативный оператор, так что `Тип1 -> Тип2 -> Тип3` это на самом деле `Тип1 -> (Тип2 -> Тип3)`: функция, возвращающая функцию.

Правда о функциях многих переменных

- Настала пора раскрыть тайну функций многих переменных в Haskell:
- Их не существует.
- `->` — правоассоциативный оператор, так что `Тип1 -> Тип2 -> Тип3` это на самом деле `Тип1 -> (Тип2 -> Тип3)`: функция, возвращающая функцию.
- `\x y -> результат` это сокращение для `\x -> \y -> результат`, а со скобками `\x -> (\y -> результат)`.

Правда о функциях многих переменных

- Настала пора раскрыть тайну функций многих переменных в Haskell:
- Их не существует.
- `->` — правоассоциативный оператор, так что `Тип1 -> Тип2 -> Тип3` это на самом деле `Тип1 -> (Тип2 -> Тип3)`: функция, возвращающая функцию.
- `\x y -> результат` это сокращение для `\x -> \y -> результат`, а со скобками `\x -> (\y -> результат)`.
- Применение функций (не `$`, а пробел), наоборот, левоассоциативно. То есть `f x y` читается как `(f x) y`.

Каррирование

- Обычно в математике для сведения функций нескольких аргументов к функциям одного аргумента используется декартово произведение: $A \times B \to C$, а не $A \to (B \to C)$.

Каррирование

- Обычно в математике для сведения функций нескольких аргументов к функциям одного аргумента используется декартово произведение: $A \times B \to C$, а не $A \to (B \to C)$.
- В Haskell тоже можно было бы писать

  ```
  foo :: (Int, Int) -> Int
  foo (x, y) = x + y
  ```

  для определения функций и `foo (1, 2)` для вызова.

Каррирование

- Обычно в математике для сведения функций нескольких аргументов к функциям одного аргумента используется декартово произведение: $A \times B \to C$, а не $A \to (B \to C)$.
- В Haskell тоже можно было бы писать

  ```
  foo :: (Int, Int) -> Int
  foo (x, y) = x + y
  ```

  для определения функций и `foo (1, 2)` для вызова.
- Но Haskell здесь следует традиции λ-исчисления.

Каррирование

- Обычно в математике для сведения функций нескольких аргументов к функциям одного аргумента используется декартово произведение: $A \times B \to C$, а не $A \to (B \to C)$.
- В Haskell тоже можно было бы писать

  ```
  foo :: (Int, Int) -> Int
  foo (x, y) = x + y
  ```

  для определения функций и `foo (1, 2)` для вызова.
- Но Haskell здесь следует традиции λ-исчисления.
- Эти подходы эквивалентны, так как множества $A \times B \to C$ и $A \to (B \to C)$ всегда изоморфны

Каррирование

- Обычно в математике для сведения функций нескольких аргументов к функциям одного аргумента используется декартово произведение: $A \times B \to C$, а не $A \to (B \to C)$.
- В Haskell тоже можно было бы писать

  ```
  foo :: (Int, Int) -> Int
  foo (x, y) = x + y
  ```

  для определения функций и `foo (1, 2)` для вызова.
- Но Haskell здесь следует традиции λ-исчисления.
- Эти подходы эквивалентны, так как множества $A \times B \to C$ и $A \to (B \to C)$ всегда изоморфны ($C^{A \cdot B} = C^{B^A}$).
- В Haskell этот изоморфизм реализуют функции

Каррирование

- Обычно в математике для сведения функций нескольких аргументов к функциям одного аргумента используется декартово произведение: $A \times B \to C$, а не $A \to (B \to C)$.
- В Haskell тоже можно было бы писать

  ```
  foo :: (Int, Int) -> Int
  foo (x, y) = x + y
  ```

  для определения функций и `foo (1, 2)` для вызова.
- Но Haskell здесь следует традиции λ-исчисления.
- Эти подходы эквивалентны, так как множества $A \times B \to C$ и $A \to (B \to C)$ всегда изоморфны ($C^{A \cdot B} = C^{B^A}$).
- В Haskell этот изоморфизм реализуют функции

  ```
  curry :: ((a, b) -> c) -> a -> b -> c
  uncurry :: (a -> b -> c) -> (a, b) -> c
  ```

Частичное применение

- Преимущество каррированных функций в том, что естественным образом появляется частичное применение.
- То есть мы можем применить функцию «двух аргументов» только к первому и останется функция одного аргумента:

```
ghci> :t sortBy
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
ghci> :t sortBy (flip compare)
sortBy (flip compare) :: Ord a => [a] -> [a]
```

- Заметьте, что здесь частичное применение в двух местах: `flip` можно теперь рассматривать как функцию трёх аргументов!

Частичное применение

- Преимущество каррированных функций в том, что естественным образом появляется частичное применение.
- То есть мы можем применить функцию «двух аргументов» только к первому и останется функция одного аргумента:

```
ghci> :t sortBy
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
ghci> :t sortBy (flip compare)
```

Частичное применение

- Преимущество каррированных функций в том, что естественным образом появляется частичное применение.
- То есть мы можем применить функцию «двух аргументов» только к первому и останется функция одного аргумента:

```
ghci> :t sortBy
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
ghci> :t sortBy (flip compare)
sortBy (flip compare) :: Ord a => [a] -> [a]
```

Частичное применение

- Преимущество каррированных функций в том, что естественным образом появляется частичное применение.
- То есть мы можем применить функцию «двух аргументов» только к первому и останется функция одного аргумента:

```
ghci> :t sortBy
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
ghci> :t sortBy (flip compare)
sortBy (flip compare) :: Ord a => [a] -> [a]
```

- Заметьте, что здесь частичное применение в двух местах:

Частичное применение

- Преимущество каррированных функций в том, что естественным образом появляется частичное применение.
- То есть мы можем применить функцию «двух аргументов» только к первому и останется функция одного аргумента:

```
ghci> :t sortBy
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
ghci> :t sortBy (flip compare)
sortBy (flip compare) :: Ord a => [a] -> [a]
```

- Заметьте, что здесь частичное применение в двух местах: `flip` можно теперь рассматривать как функцию трёх аргументов!

Сечения операторов

- Для применения бинарного оператора только к первому аргументу можно использовать его префиксную форму:

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

Сечения операторов

- Для применения бинарного оператора только к первому аргументу можно использовать его префиксную форму:

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

- А ко второму? Можно использовать лямбду

```
\x -> x / 2
```

или `flip`

```
flip (/) 2
```

Сечения операторов

- Для применения бинарного оператора только к первому аргументу можно использовать его префиксную форму:

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

- А ко второму? Можно использовать лямбду

```
\x -> x / 2
```

  или `flip`

```
flip (/) 2
```

- Но есть специальный синтаксис `(арг оп)` и `(оп арг)`:

```
ghci> (1 `div`) 2
0
ghci> (/ 2) 4
2.0
```

Сечения операторов

- Для применения бинарного оператора только к первому аргументу можно использовать его префиксную форму:

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

- А ко второму? Можно использовать лямбду

```
\x -> x / 2
```

  или `flip`

```
flip (/) 2
```

- Но есть специальный синтаксис `(арг оп)` и `(оп арг)`:

```
ghci> (1 `div`) 2
```

Сечения операторов

- Для применения бинарного оператора только к первому аргументу можно использовать его префиксную форму:

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

- А ко второму? Можно использовать лямбду

```
\x -> x / 2
```

  или `flip`

```
flip (/) 2
```

- Но есть специальный синтаксис `(арг оп)` и `(оп арг)`:

```
ghci> (1 `div`) 2
0
ghci> (/ 2) 4
2.0
```

Сечения операторов

- Для применения бинарного оператора только к первому аргументу можно использовать его префиксную форму:

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

- А ко второму? Можно использовать лямбду

```
\x -> x / 2
```

  или `flip`

```
flip (/) 2
```

- Но есть специальный синтаксис `(арг оп)` и `(оп арг)`:

```
ghci> (1 `div`) 2
0
ghci> (/ 2) 4
```

Сечения операторов

- Для применения бинарного оператора только к первому аргументу можно использовать его префиксную форму:

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

- А ко второму? Можно использовать лямбду

```
\x -> x / 2
```

или `flip`

```
flip (/) 2
```

- Но есть специальный синтаксис `(арг оп)` и `(оп арг)`:

```
ghci> (1 `div`) 2
0
ghci> (/ 2) 4
2.0
```

Сечения кортежей

- Расширение `TupleSections` позволяет частично применять конструкторы кортежей:

```
ghci> :set -XTupleSections
ghci> (, "I", , "Love", True) 1 False
(1,"I",False,"Love",True)
ghci> :t (, "I", , "Love", True)
(, "I", , "Love", True)
  :: t1 -> t2 -> (t1, [Char], t2, [Char], Bool)
```

Сечения кортежей

- Расширение `TupleSections` позволяет частично применять конструкторы кортежей:

```
ghci> :set -XTupleSections
ghci> (, "I", , "Love", True) 1 False
```

Сечения кортежей

- Расширение `TupleSections` позволяет частично применять конструкторы кортежей:

```
ghci> :set -XTupleSections
ghci> (, "I", , "Love", True) 1 False
(1,"I",False,"Love",True)
ghci> :t (, "I", , "Love", True)
(, "I", , "Love", True)
  :: t1 -> t2 -> (t1, [Char], t2, [Char], Bool)
```

Сечения кортежей

- Расширение `TupleSections` позволяет частично применять конструкторы кортежей:

```
ghci> :set -XTupleSections
ghci> (, "I", , "Love", True) 1 False
(1,"I",False,"Love",True)
ghci> :t (, "I", , "Love", True)
```

Сечения кортежей

- Расширение `TupleSections` позволяет частично применять конструкторы кортежей:

```
ghci> :set -XTupleSections
ghci> (, "I", , "Love", True) 1 False
(1,"I",False,"Love",True)
ghci> :t (, "I", , "Love", True)
(, "I", , "Love", True)
  :: t1 -> t2 -> (t1, [Char], t2, [Char], Bool)
```

η-эквивалентность (сокращение аргументов)

- Рассмотрим два определения

```haskell
foo' x = foo x
-- или foo' = \x -> foo x
```

- `foo' y == foo y`, какое бы `y` (и `foo`) мы не взяли.

η-эквивалентность (сокращение аргументов)

- Рассмотрим два определения

```haskell
foo' x = foo x
-- или foo' = \x -> foo x
```

- `foo' y == foo y`, какое бы `y` (и `foo`) мы не взяли.
- Поэтому мы можем упростить определение:

```haskell
foo' = foo
```

η-эквивалентность (сокращение аргументов)

- Рассмотрим два определения

```
foo' x = foo x
-- или foo' = \x -> foo x
```

- `foo' y == foo y`, какое бы `y` (и `foo`) мы не взяли.
- Поэтому мы можем упростить определение:

```
foo' = foo
```

- Это также относится к случаям, когда совпадают часть аргументов в конце:

```
foo' x y z w = foo (y + x) z w
```

эквивалентно

```
foo' x y = foo (y + x)
```

η-эквивалентность (сокращение аргументов)

- И когда само определение такой формы не имеет, но может быть к ней преобразовано

  ```
  root4 x = sqrt (sqrt x)
  ```

  можно переписать как

  ```
  root4 x = (sqrt . sqrt) x

  root4 = sqrt . sqrt
  ```

- Ещё пример:

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

  ($) f x = f x
  ($) f = f
  ($) f = id f
  ($) = id
  ```

η-эквивалентность (сокращение аргументов)

- И когда само определение такой формы не имеет, но может быть к ней преобразовано

  ```
  root4 x = sqrt (sqrt x)
  ```

  можно переписать как

  ```
  root4 x = (sqrt . sqrt) x

  root4 = sqrt . sqrt
  ```

- Ещё пример:

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

  ($) f x = f x
  ```

η-эквивалентность (сокращение аргументов)

- И когда само определение такой формы не имеет, но может быть к ней преобразовано

  ```
  root4 x = sqrt (sqrt x)
  ```

  можно переписать как

  ```
  root4 x = (sqrt . sqrt) x

  root4 = sqrt . sqrt
  ```

- Ещё пример:

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

  ($) f x = f x
  ($) f = f
  ($) f = id f
  ($) = id
  ```

η-эквивалентность (сокращение аргументов)

- И когда само определение такой формы не имеет, но может быть к ней преобразовано

  ```
  root4 x = sqrt (sqrt x)
  ```

  можно переписать как

  ```
  root4 x = (sqrt . sqrt) x

  root4 = sqrt . sqrt
  ```

- Ещё пример:

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

  ($) f x = f x
  ($) f = f
  ```

η-эквивалентность (сокращение аргументов)

- И когда само определение такой формы не имеет, но может быть к ней преобразовано

  ```
  root4 x = sqrt (sqrt x)
  ```

  можно переписать как

  ```
  root4 x = (sqrt . sqrt) x

  root4 = sqrt . sqrt
  ```

- Ещё пример:

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

  ($) f x = f x
  ($) f = f
  ($) f = id f
  ($) = id
  ```

η-эквивалентность (сокращение аргументов)

- И когда само определение такой формы не имеет, но может быть к ней преобразовано

  ```
  root4 x = sqrt (sqrt x)
  ```

  можно переписать как

  ```
  root4 x = (sqrt . sqrt) x

  root4 = sqrt . sqrt
  ```

- Ещё пример:

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

  ($) f x = f x
  ($) f = f
  ($) f = id f
  ```

η-эквивалентность (сокращение аргументов)

- И когда само определение такой формы не имеет, но может быть к ней преобразовано

```
root4 x = sqrt (sqrt x)
```

  можно переписать как

```
root4 x = (sqrt . sqrt) x

root4 = sqrt . sqrt
```

- Ещё пример:

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

($) f x = f x
($) f = f
($) f = id f
($) = id
```

Бесточечный стиль

- Как видим, с помощью композиции и других операций можно дать определение некоторых функций без использования переменных.
- Это называется *бесточечным стилем* (переменные рассматриваются как точки в пространстве значений).

Бесточечный стиль

- Как видим, с помощью композиции и других операций можно дать определение некоторых функций без использования переменных.
- Это называется *бесточечным стилем* (переменные рассматриваются как точки в пространстве значений).
- Оказывается, что очень многие выражения в Haskell имеют бесточечный эквивалент.
- На Haskell Wiki можно найти инструменты, позволяющие переводить между стилями.
- Опять же, не перестарайтесь:

```
> pl \x y -> compare (f x) (f y)
((. f) . compare .)
```

Бесточечный стиль

- Как видим, с помощью композиции и других операций можно дать определение некоторых функций без использования переменных.
- Это называется *бесточечным стилем* (переменные рассматриваются как точки в пространстве значений).
- Оказывается, что очень многие выражения в Haskell имеют бесточечный эквивалент.
- На Haskell Wiki можно найти инструменты, позволяющие переводить между стилями.
- Опять же, не перестарайтесь:

```
> pl \x y -> compare (f x) (f y)
((. f) . compare .)
```

- В языках семейств Forth и APL бесточечный стиль является основным.

Программирование, направляемое типами

- При реализации полиморфных функций очень часто их тип подсказывает реализацию.
- Пример:

  ```
  foo :: (a -> b, b -> Int) -> (a -> Int)
  ```

  Какую часть реализации можно написать сразу?

  ```
  foo (f, g) = \x -> ?
  ```

  Типы переменных:

  ```
  f :: a -> b, g :: b -> Int, x :: a, ? :: Int
  ```

  Можно увидеть, что `?` может быть `g ??`, где `?? :: b = f x`.

Программирование, направляемое типами

- При реализации полиморфных функций очень часто их тип подсказывает реализацию.
- Пример:

  `foo :: (a -> b, b -> Int) -> (a -> Int)`

  Какую часть реализации можно написать сразу?

  `foo (f, g) = \x -> ?`

  Типы переменных:

  `f :: a -> b, g :: b -> Int, x :: a, ? :: Int`

  Можно увидеть, что `?` может быть `g ??`, где `?? :: b = f x`.

Программирование, направляемое типами

- При реализации полиморфных функций очень часто их тип подсказывает реализацию.
- Пример:

  ```
  foo :: (a -> b, b -> Int) -> (a -> Int)
  ```

  Какую часть реализации можно написать сразу?

  ```
  foo (f, g) = \x -> ?
  ```

  Типы переменных:

  ```
  f :: a -> b, g :: b -> Int, x :: a, ? :: Int
  ```

  Можно увидеть, что `?` может быть `g ??`, где `?? :: b = f x`.

Программирование, направляемое типами

- При реализации полиморфных функций очень часто их тип подсказывает реализацию.
- Пример:

  ```
  foo :: (a -> b, b -> Int) -> (a -> Int)
  ```

  Какую часть реализации можно написать сразу?

  ```
  foo
  ```

Программирование, направляемое типами

- При реализации полиморфных функций очень часто их тип подсказывает реализацию.
- Пример:

  ```
  foo :: (a -> b, b -> Int) -> (a -> Int)
  ```

  Какую часть реализации можно написать сразу?

  ```
  foo (f, g) =
  ```

Программирование, направляемое типами

- При реализации полиморфных функций очень часто их тип подсказывает реализацию.
- Пример:

```
foo :: (a -> b, b -> Int) -> (a -> Int)
```

Какую часть реализации можно написать сразу?

```
foo (f, g) = \x ->
```

Программирование, направляемое типами

- При реализации полиморфных функций очень часто их тип подсказывает реализацию.
- Пример:

  ```
  foo :: (a -> b, b -> Int) -> (a -> Int)
  ```

  Какую часть реализации можно написать сразу?

  ```
  foo (f, g) = \x -> ?
  ```

  Типы переменных:

Программирование, направляемое типами

- При реализации полиморфных функций очень часто их тип подсказывает реализацию.
- Пример:

  `foo :: (a -> b, b -> Int) -> (a -> Int)`

  Какую часть реализации можно написать сразу?

  `foo (f, g) = \x -> ?`

  Типы переменных:

  `f :: a -> b, g :: b -> Int, x :: a, ? :: Int`

  Можно увидеть, что `?` может быть `g ??`, где `?? :: b = f x`.

Программирование, направляемое типами

- При реализации полиморфных функций очень часто их тип подсказывает реализацию.
- Пример:

  ```
  foo :: (a -> b, b -> Int) -> (a -> Int)
  ```

  Какую часть реализации можно написать сразу?

  ```
  foo (f, g) = \x -> ?
  ```

  Типы переменных:

  ```
  f :: a -> b, g :: b -> Int,
  ```

Программирование, направляемое типами

- При реализации полиморфных функций очень часто их тип подсказывает реализацию.
- Пример:

  `foo :: (a -> b, b -> Int) -> (a -> Int)`

  Какую часть реализации можно написать сразу?

  `foo (f, g) = \x -> ?`

  Типы переменных:

  `f :: a -> b, g :: b -> Int, x :: a, ? :: Int`

  Можно увидеть, что `?` может быть `g ??`, где `?? ::`

Программирование, направляемое типами

- При реализации полиморфных функций очень часто их тип подсказывает реализацию.
- Пример:

  ```
  foo :: (a -> b, b -> Int) -> (a -> Int)
  ```

  Какую часть реализации можно написать сразу?

  ```
  foo (f, g) = \x -> ?
  ```

  Типы переменных:

  ```
  f :: a -> b, g :: b -> Int, x :: a, ? :: Int
  ```

  Можно увидеть, что `?` может быть `g ??`, где

  ```
  ?? :: b =
  ```

Программирование, направляемое типами

- При реализации полиморфных функций очень часто их тип подсказывает реализацию.
- Пример:

  ```
  foo :: (a -> b, b -> Int) -> (a -> Int)
  ```

  Какую часть реализации можно написать сразу?

  ```
  foo (f, g) = \x -> ?
  ```

  Типы переменных:

  ```
  f :: a -> b, g :: b -> Int, x :: a, ? :: Int
  ```

  Можно увидеть, что `?` может быть `g ??`, где `?? :: b = f x`.

Типизированные дыры в коде

- Эти рассуждения не обязательно делать вручную.
- Если в выражении (а не в образце) использовать дыру _ (или _название), то увидим ожидаемый в этом месте тип.
- ```
  foo :: (a -> b, b -> Int) -> (a -> Int)
  foo (f, g) = \x -> g _
  ```
  выдаст сообщение
  ```
  Found hole `_' with type: b
  Where: `b' is a rigid type variable bound by
  the type signature for foo :: (a -> b, b -> Int) ->
  ↪  (a -> Int)
  Relevant bindings include ...
  ```
- Дыр может быть несколько.
  ```