**ФУНКЦІОНАЛЬНЕ ПРОГРАМУВАННЯ**

Лекція 14

# ЧИСТО ФУНКЦІОНАЛЬНІ МОВИ ПРОГРАМУВАННЯ - 2

**2020**

**Повний текст лекції буде розміщений на сайті baklaniv.at.ua**

## Композиция

Если необходимо применить функцию к результату вычисления другой, то мы используем композицию:

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

Запись `f.g` напоминает математическое обозначение $f \circ g$.

Функции должны иметь подходящие типы. Если `g :: X -> Y` и `f :: Y -> Z`, то `(f.g) :: X -> Z`.

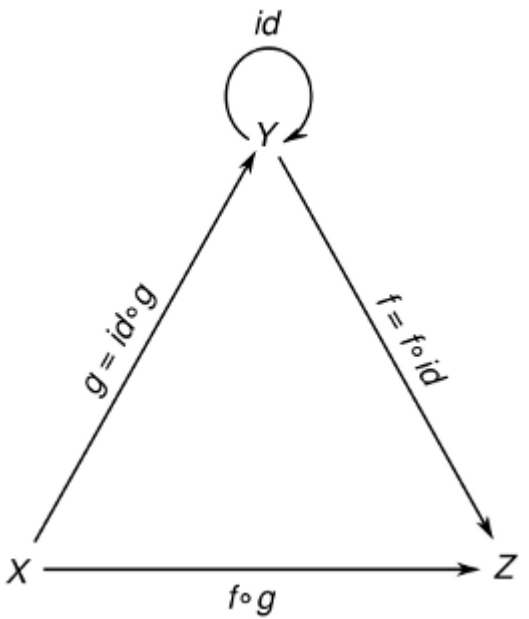
Обратите внимание: если записано `(f.g)`, то сначала к аргументу применяется **`g`**, а затем к результату применяется **`f`**, не наоборот.

В Prelude имеется тождественное отображение:

```
id x = x
(f.id) = f и (id.g) = g, где id :: Y -> Y.
```

Все это можно изобразить на диаграмме (рисунок 1), где стрелка от X к Y обозначает функцию X → Y.

**Рисунок 1. Функции g :: X -> Y, f :: Y -> Z, (f.g) :: X -> Z, id :: Y -> Y.**



Если нужно определить функцию, возводящую аргумент в четвертую степень, то можно записать:

```
fourth :: Double -> Double
fourth x = square (square x)
```

Через композицию функций это выглядит так:

```
fourth x = (square . square) x
```

Но `(square . square)` и есть интересующая нас функция:

```
ghci> (square . square) 2
16
```

Поэтому лучше использовать определение

```
fourth = square . square
```

Если мы хотим фиксировать первый аргумент функции `f`, то можно записать:

```
g x = f a x
```

Более компактная запись, обеспечиваемая частичным применением:

```
g = f a
```

Это называется **бесточечной записью** (point-free). Под точками здесь понимаются аргументы *x*, *y*, *z*, … (по аналогии с точками пространства), а не оператор `.`.

При разумном использовании бесточечная запись позволяет выражаться более ясно и лаконично и мыслить в терминах композиции функций.

## Анонимные функции

Функции могут быть анонимными. Для них используется запись вида

```
1\ x y -> sqrt (x*x + y*y)
```

Здесь **** – символ, похожий на греческую букву λ. После него перечисляются аргументы. **->** обозначает отображение.

Считается, что выражение после **->** простирается как можно дальше, так что если анонимную функцию нужно вставить в другое выражение, то ее запись берется в скобки:

```
1… (\ x y -> sqrt (x*x + y*y)) …
```

Не будем забывать о каррировании по умолчанию: приведенное выше выражение на самом деле является удобным сокращением для

```
1\ x -> (\ y -> sqrt (x*x + y*y))
```

Анонимные функции дают возможность указать в выражении функцию, не записывая предварительно для нее уравнение.

```
1ghci> (\ x y -> sqrt (x^2 + y^2)) 3 4
25.0
```

## Операторы и функции

Операторы не являются особыми элементами языка – это те же функции. Если взять инфиксный оператор в скобки, то мы получим соответствующую каррированную функцию двух аргументов.

Если в скобках записать до или после оператора значение, то мы получим функцию одного аргумента, где другой аргумент фиксирован (таблица 1). Такую запись называют **сечением** (section).

| Запись | Значение |
|---|---|
| **(+)** | (\ x y -> x+y) |
| **(x+)** | \ y -> x+y |
| **(+y)** | \ x -> x+y |

Таблица 1. Сечения инфиксного оператора на примере **+**

```
ghci> (+) 2 3
5
ghci> (2+) 3
5
ghci> (+3) 2
5
```

Это порождает некоторую путаницу с инфиксным минусом. Запись вроде `(-23)` – это число, а не функция `(\ x -> x - 23)`. При необходимости используйте предопределенные `subtract` и `negate`:

```
subtract x y =  x - y
negate   x   = -x
```

Помните, что запись `f -23` обозначает разность; применение функции должно записываться со скобками: `f (-23)`.

Кроме того, имеется возможность использовать имя функции как инфиксный оператор; для этого нужно взять имя в обратные кавычки: `x `f` y` – это то же самое, что и `f x y`.

Но этим не стоит злоупотреблять.

## Оператор $

Применение функции имеет самый высокий приоритет. Так, `f x + g x` обозначает `(f x) + (g x)`.

Для удобства имеется оператор `$`, который также обозначает применение функции, но имеет самый низкий приоритет и правоассоциативен.

Особенности оператора `$`:

- `($)` можно использовать как функцию, которая принимает первый аргумент ко второму.
- Благодаря низкому приоритету, `f $ x + g x` обозначает `f (x + g x)`.
- Благодаря правоассоциативности, `f $ g $ h x` обозначает `f (g (h x))`.

Главным образом, `$` позволяет писать меньше скобок.

# Списки

**Список** – это последовательность элементов одного типа. Если тип элементов — `T`, то тип списка обозначается в виде `[T]`.

Список имеет рекурсивную структуру:

- Списком является пустой список `[]`.
- Списком является `x : xs`, где `x :: T` – голова, `xs :: [T]` – хвост.

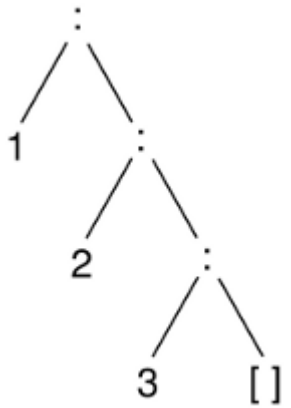
Голова присоединяется в начало к хвосту конструктором списка `(:)`.

Пусть нужно построить список из целых чисел 1, 2, 3 (рисунок 2). Он будет иметь тип `[Int]`. Нужно начать с пустого списка `[]`. Далее к нему присоединяются значения по одному:

```
ghci> 3 : []
[3]
ghci> 2 : it
[2,3]
ghci> 1 : it
[1,2,3]
```

`[1,2,3]` – это краткая запись для `1 : (2 : (3 : []))`. Конструктор `:` правоассоциативен, можно писать просто `1 : 2 : 3 : []`.

**Рисунок 2. Дерево списка [1,2,3].**



В таблице 2 перечислены полезные функции для работы со списками, которые можно найти в Prelude (не волнуйтесь, скоро вы убедитесь, что их очень легко определить самостоятельно).

| | | |
|---|---|---|
| **`null`** | **Проверка на пустой список** | **`ghci> null []`**<br>**`True`** |
| `head` | Выбор головы в непустом списке | `ghci> head [1,2,3]`<br>`1` |
| `tail` | Выбор хвоста в непустом списке | `tail [1,2,3]`<br>`[2,3]` |
| `length` | Количество элементов | `ghci> length [1,2,3]`<br>`3` |
| `(!!)` | Доступ по индексу (элементы нумеруются с 0) | `ghci> [1,2,3] !! 1`<br>`2` |
| `take` | Выбор первых *n* элементов | `ghci> take 2 [1,2,3]`<br>`[1,2]` |
| `drop` | Удаление первых *n* элементов | `ghci> drop 2 [1,2,3]`<br>`[3]` |
| `(++)` | Объединение двух списков | `ghci> [1,2,3] ++`<br>`[4,5,6]`<br>`[1,2,3,4,5,6]` |
| `concat` | Объединение списка списков | `ghci> concat`<br>`[[1,2],[3,4],[5,6]]`<br>`[1,2,3,5,6]` |
| `reverse` | Обращение | `ghci> reverse` |

| | | |
|---|---|---|
| **`null`** | **Проверка на пустой список** | **`ghci> null []`**<br>**`True`**<br>`[1,2,3]`<br>`[3,2,1]`<br>`ghci> sum [1,2,3,4]`<br>`10` |
| `sum` | Сумма элементов | `ghci> product`<br>`[1,2,3,4]`<br>`24` |
| `product` | Произведение элементов | `ghci> zip`<br>`['a','b','c']`<br>`[1,2,3]`<br>`[('a',1),('b',2),`<br>`('c',3)]` |
| `zip` | Объединение двух списков в список пар | |

Таблица 2. Основные функции для работы со списками

## Строки

Тип строк `String` соответствует списку символов `[Char]`. Строковые значения, как и в других языках, можно записывать в двойных кавычках: `"abc"`, но это просто удобное сокращение для `['a','b','c']`. Пустая строка – `""`.

```
ghci> "Bob"++" "++"Dobbs"
"Bob Dobbs"
ghci> concat ["Bob", " ", "Dobbs"]
"Bob Dobbs"
ghci> reverse "Bob Dobbs"
"sbboD boB"
```

# Полиморфизм

Выше упоминались функции, действующие для разных типов. Например, аргумент `curry` может иметь тип `(Int,Int)->Int, (Int,Char)->String, и т.д.`

Чтобы отразить это, тип `curry` описывается с использованием **ти́повых переменных**:

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

Ти́повые переменные, в отличие от самих типов, записываются со строчной буквы. Для них принято использовать короткие имена, чаще однобуквенные.

Если какая-то переменная встречается в типе несколько раз, то подразумевается один и тот же, заранее не фиксированный тип. Типы, в которых фигурируют переменные, называются полиморфными.

Таким образом, типы, как и функции, имеют параметры. Это называется параметрическим полиморфизмом.

Теперь можно посмотреть на типы уже упомянутых функций:

```
ghci> :type id
id :: a -> a
ghci> :type fst
fst :: (a, b) -> a
ghci> :type (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
ghci> :type zip
zip :: [a] -> [b] -> [(a, b)]
```

# Частичные функции

Функции не обязаны быть полными. Выражение, которое не вычисляется успешно (вызывает ошибку или зацикливается), обозначается `⊥`. Считается, что элемент `⊥` содержится в любом типе, поэтому он называется дном (bottom).

В Prelude есть функция `error :: String -> a` для выдачи сообщения об ошибке и полиморфное неопределенное значение `undefined :: a`.

```
ghci> error "Something wrong"
*** Exception: Something wrong
ghci> undefined
*** Exception: Prelude.undefined
```

# Классы типов

Если вы решите узнать тип арифметических функций, то будете озадачены:

```
ghci> :type (+)
(+) :: (Num a) => a -> a -> a
ghci> :type (/)
(/) :: (Fractional a) => a -> a -> a
```

Здесь **a**, как и прежде, означает переменную типа. Но `(C a) =>` в сигнатуре – ограничение на класс, означающее, что **a** должно принадлежать классу **C**.

Классы дают структурированный подход к перегрузке. Каждый класс предоставляет набор методов, которые должны перегружаться для типов-представителей.

Числа также перегружены. Например, `23` можно рассматривать и как целое число, и как число с плавающей точкой:

```
ghci> :type 23
23 :: (Num t) => t
```

Перечислим основные классы, которые можно найти в Prelude.

## Eq

`Eq` – класс типов, в которых значения можно проверять на равенство и неравенство.

**Представители:** `Bool, Char, Double, Float, Int, Integer`; кортежи и списки элементов, чей тип принадлежит `Eq`.

**Методы**: `(==) :: a -> a -> Bool` и `(/=) :: a -> a -> Bool`.

Как и прежде, `(==)` и `(/=)` могут использоваться в качестве инфиксных операторов `==` и `/=`.

```
ghci> True /= True
False
ghci> (1,2,3) == (1,1+1,1+1+1)
True
```

(Не путайте `/=` с оператором `!=` в других языках.)

## Ord

`Ord` – класс типов, в которых значения линейно упорядочены (для любых $x$ и $y$ имеем $x \leq y$ или $y \leq x$).

**Представители**: `Bool, Char, Double, Float, Int, Integer`; кортежи и списки элементов, чей тип принадлежит `Ord` (они сравниваются лексикографически).

**Методы**:

- Функции сравнения с типом `a -> a -> Bool: (<), (>=), (>), (<=)`
- `>min :: a -> a -> a` и `max :: a -> a -> a`.

```
ghci> "Bob" < "Dobbs"
True
ghci> min "Bob" "Dobbs"
"Bob"
```

## Enum

`Enum` – класс последовательностей.

**Представители**: `(), Bool, Char, Double, Float, Int, Integer`.

**Методы**:

- Все элементы последовательностей занумерованы, начиная с 0. По элементу можно получить порядковый номер, а по порядковому номеру – элемент:
  ```
  toEnum :: Int -> a, fromEnum :: a -> Int
  ```
- Если элемент не 0-й, то для него имеется предшествующий; если элемент не последний, то для него имеется последующий:
  ```
  pred :: a -> a, succ :: a -> a
  ```

```
ghci> fromEnum 'c'
99
ghci> toEnum 99 :: Char
'c'
ghci> pred 'b'
'a'
ghci> succ 'b'
'c'
```

Обратите внимание: `toEnum` – полиморфная функция с типом `Int -> a`, и из выражения `toEnum 99` невозможно установить тип, поэтому мы явно указываем `toEnum 99 :: Char`.

## Num

`Num` – класс чисел.

**Представители:** `Double, Float, Int, Integral`.

**Методы**:

- Противоположное число, абсолютное значение, знак (тип `a -> a -> a`):
  ```
  negate, abs, signum
  ```
- Сложение, умножение, вычитание (тип `a -> a -> a`):
  ```
  (+), (*), (-)
  ```
- Преобразование из целого числа: `fromInteger :: Integer -> a`

```
ghci> [signum (-23), signum 0, signum 23]
[-1,0,1]
ghci> abs (-23)
23
ghci> negate 23
-23
ghci> [23 + 23, 23 * 23, 23 - 23]
[46,529,0]
```

## Integral

`Integral` – класс целых чисел.

**Представители**: `Int, Integral`.

**Методы**:

- Целочисленное деление с округлением к 0: `quot :: a -> a -> a`
- Остаток от деления `quot`: `rem :: a -> a -> a`
- Целочисленное деление с округлением к −∞: `div :: a -> a -> a`
- Остаток от деления `div`: `mod :: a -> a -> a`
- Пара чисел (частное, остаток): `quotRem :: a -> a -> (a, a)`, `divMod :: a -> a -> (a, a)`
- Преобразование в целое число типа `Integer`: `toInteger :: a -> Integer`

```
ghci> quotRem (-23) 3
(-7,-2)
ghci> divMod (-23) 3
(-8,1)
```

Важная функция:

```
fromIntegral :: (Integral a, Num b) => a -> b
fromIntegral = fromInteger . toInteger
```

Пример использования:

```
ghci> let mean xs = sum xs / (fromIntegral.length) xs
ghci> mean [1,2,3,4,5]
3.0
```

## Fractional

`Fractional` – класс дробных чисел.

**Представители**: `Double, Float`.

**Методы**: `(/) :: a -> a -> a` и `recip :: a -> a`

```
ghci> 3 / 2
1.5
ghci> recip 3
0.3333333333333333
```

## Floating

`Floating` – класс чисел с плавающей точкой.

**Представители**: `Double, Float`.

**Методы**:

- Квадратный корень `sqrt :: a -> a`
- Возведение в степень `(**) :: a -> a -> a`
- Число π = 3.1415926…: `pi :: a`
- Экспонента `exp :: a -> a` и натуральный логарифм `log :: a -> a`
- Логарифм по данному основанию `logBase :: a -> a -> a`
- Тригонометрические функции типа `a -> a`: `sin, tan, cos, asin, atan, acos, sinh, tanh, cosh, asinh, atanh, acosh`.

```
ghci> exp 1
2.718281828459045
ghci> logBase 2 64
6.0
ghci> sqrt 2
1.4142135623730951
ghci> sin $ pi/2
1.0
ghci> 1.99 ** 3.99
15.574846491761065
```

## Show

Основной интересующий нас метод класса `Show – show :: a -> String` – используется для вывода значений типа **a**:

```
ghci> show 23
"23"
ghci> show [1,2,3]
"[1,2,3]"
```

ghci печатает значения, если они принадлежат классу `Show`. Иначе появляется сообщение об ошибке. Часто оно возникает при попытке напечатать функцию (например, при частичном применении):

```
ghci> (+) 1

<interactive>:1:0:
No instance for (Show (t -> t))
arising from a use of `print' at <interactive>:1:0-4
Possible fix: add an instance declaration for (Show (t -> t))
In a stmt of a 'do' expression: print it
```

## Read

Для класса `Read` нас будет интересовать функция `read :: (Read a) => String -> a`, по сути противоположная `show`.

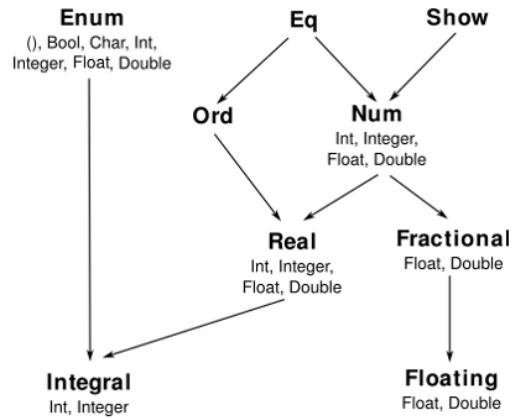
```
ghci> read "23" :: Integer
23
ghci> read "[('a',23)]" :: [(Char,Integer)]
[('a',23)]
```

# Заключение

Итак, мы узнали о строгой статической типизации и основных типах и классах Haskell. Преимущества и особенности системы типов вы еще оцените при дальнейшем изучении и применении языка.

Рассмотренные классы образуют иерархию (рисунок 3).

**Рисунок 3. Иерархия классов**



Интересные ресурсы для самостоятельного изучения:

- Библиотека Prelude: http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html (EN) (Для начала изучите набор функций и их типы.)
- Hoogle – Web-приложение для поиска функций в стандартных библиотеках, в том числе по сигнатуре: http://haskell.org/hoogle/.
- Команда `:info` выводит информацию о данном имени. Попробуйте применить ее к функциям, типам, методам и классам.

# Соглашения по именованию

Для начала подытожим использующиеся в Haskell соглашения по именованию.

- Язык чувствителен к регистру, т. е. идентификаторы `foobar`, `fooBar` и `FooBar` различаются.
- Имена функций, аргументов и ти́повых переменных должны начинаться со строчной буквы. Далее могут следовать 0 и более букв любого регистра, цифр, подчеркиваний и штрихов '.
- Имена типов и классов должны начинаться с заглавной буквы.
- Зарезервирован ряд ключевых слов, которые не могут использоваться в качестве идентификаторов: `case`, `class`, `data`, `default`, `deriving`, `do`, `else`, `if`, `import`, `in`, `infix`, `infixl`, `infixr`, `instance`, `let`, `module`, `newtype`, `of`, `then`, `type`, `where`, `_`.

Подробности см. в Haskell 98 Report.

Некоторые неформальные соглашения:

- Принято использовать так называемый camelCase, когда слова в идентификаторе пишутся слитно, но каждое новое слово начинается с заглавной буквы.
- Как и в математике, часто используются имена со штрихом на конце: `x, x', x''`, …
- На конце имен списков часто указывается s, как при образовании множественного числа в английском языке: `xs, ys, zs,` …
- Для ти́повых переменных принято использовать короткие имена, чаще однобуквенные.

# Комментарии и грамотное программирование

В обычном файле .hs многострочные комментарии располагаются между символами `{-` и `-}`, а однострочные – после `--`. Многострочные комментарии могут быть вложенными.

```
… -- комментарий

{-
  …
  {-
    комментарий
  -}
  …
-}
```

Дональд Кнут предложил «грамотное программирование» («literate programming»), в котором текст программы сопровождается подробным и последовательным объяснением логики работы на естественном языке. При этом код в первую очередь предназначается для чтения человеком, и лишь затем – для выполнения компьютером.

Если вместо расширения .hs использовать .lhs, то будет использоваться «грамотный» подход, и всё содержимое по умолчанию будет считаться комментариями, а код необходимо помечать отдельно: либо начальным символом `>`, либо блоками между `\begin{code}` и `\end{code}`:

```
Комментарии

> код
> …
> …

Комментарии

\begin{code}
код
…
…
\end{code}
```

При компиляции комментарии и метки комментариев будут игнорироваться. Вариант с `\begin{code}` и `\end{code}` предназначается для набора текстов в системе LaTeX.

Подробности см. по адресу http://haskell.org/haskellwiki/Literate_programming

# Сопоставление с образцом и условные конструкции

## if-then-else

Условные конструкции в Haskell могут записываться различным образом. Среди прочего, имеется стандартная конструкция:

```
1if условие then выражение1 else выражение2
```

Условие должно иметь тип `Bool`. Если его значение – `True`, то результатом будет выражение после `then`, иначе – выражение после `else`.

Пример:

```
1abs x = if x >= 0 then x else -x
```

Обе ветви должны иметь один и тот же тип, который и будет считаться типом условного выражения.

Ветвь `else` обязательна. Как следствие, любое выражение с if-then-else разбирается однозначно (иначе возникали бы ситуации, когда не ясно, к какому открывающему `if` относится закрывающее `else`).

## Уравнения с условиями

Часто условий и ветвей несколько. Как, например, в случае [signum.png]

Тогда можно использовать уравнения с условиями:

```
signum x | x >  0  =  1
         | x == 0  =  0
         | x <  0  = -1
```

**Условия** стоят после знака `|` и до знака `=`. Они просматриваются по порядку: если значение первого – `True`, то результатом считается следствие после знака `=`; если значение – `False`, то проверяется второе условие, и т. д.

При этом условия могут пересекаться и даже не покрывать в совокупности все возможные ситуации. Если все условия будут просмотрены и ни одно не даст `True`, мы получим ошибку времени выполнения.

Часто последним условием специально указывается такое, которое всегда равно `True`. Для удобства в Prelude определено

```
otherwise = True
```

(по-английски `otherwise` означает «иначе»).

```
abs x | x >= 0     =  x
      | otherwise  = -x
```

## Сопоставление с образцом

Многие функции естественно определяются через разбор различных случаев. Например, булевы функции:

```
1 not :: Bool -&gt; Bool
2 not False  =  True
3 not True   =  False
4
5 (&&) :: Bool -&gt; Bool -&gt; Bool
6 False && False  =  False
7 False && True   =  False
8 True  && False  =  False
9 True  && True   =  True
10
11(||) :: Bool -&gt; Bool -&gt; Bool
12False || False  =  False
13False || True   =  True
14True  || False  =  True
15True  || True   =  True
```

**Образцами** называются выражения в левой части уравнений, с которыми **сопоставляются** фактические аргументы функций. Сначала происходит сопоставление с образцом в первом уравнении, в случае удачного сопоставления используется правая часть; в противном случае происходит сопоставление с образцом во втором уравнении и т. д.

В образцах могут использоваться переменные, значения, а также **заполнитель _**. Переменная успешно сопоставляется с любым фактическим значением. В результате в теле функции к ней привязывается значение. При этом есть ограничение: каждая переменная может входить в образец один раз. Если предполагается, что компоненты образца совпадают, то это условие необходимо добавить в тело функции.

Если какое-либо значение не используется в правой части, то в образце можно указать _. Например, `False && _ = False, True || _ = True`. Поэтому наши определения можно упростить:

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

(||) :: Bool -&gt; Bool -&gt; Bool
False || x  =  x
True  || _  =  True
```

Такие записи лучше, потому что при сопоставлении с _ в образце значение не играет роли и не вычисляется. Для **&&** второй аргумент будет вычислен только в том случае, если первый аргумент – `True`; для **||** – только если первый аргумент – `False`.

Как мы уже говорили ранее, ошибочное или не останавливающееся вычисление можно связать со специальным элементом **⊥**. Например, ему будет соответствовать

```
bot :: a
bot = error "Bottom!"
```

Для первого варианта определений без _:

```
True  && ⊥  =  ⊥
False && ⊥  =  ⊥

True  || ⊥  =  ⊥
False || ⊥  =  ⊥
Для упрощенного варианта:
True  && ⊥  =  ⊥
False && ⊥  =  False

True  || ⊥  =  True
False || ⊥  =  ⊥
```

Таким образом, вид образцов и порядок следования уравнений влияет на вычисления. В Prelude `(||)` и `(&&)` определены в соответствии со вторым вариантом.

Образец может представлять собой список или кортеж. Исходя из этого, можно записать такие определения:

```
1 fst :: (a,b) -&gt; a
2 fst (x,_)  =  x
3
4 snd :: (a,b) -&gt; b
5 snd (_,y)  =  y
```

Чтобы разобраться с определениями `head` и `tail`, достаточно вспомнить, что список конструируется при помощи `:`. Скобки вокруг образцов вроде `(x:xs)` обязательно должны указываться, так как применение функции имеет самый высокий приоритет. В противном случае `f x:xs` будет интерпретироваться как `(f x):xs`.

Что произойдет, если с приведенными определениями попробовать вычислить `head []` или `tail []`?

Мы получим ошибку:

```
1ghci&gt; head []
2*** Exception: Non-exhaustive patterns in function head
```

`[]` не соответствует образцу `x:_`, а других уравнений для head нет.

Эта ситуация нежелательна; если для некоторых аргументов функция не вычисляется, то лучше явно добавить значение `undefined`, либо понятные сообщения об ошибках:

```
head []  =  error "empty list"

tail []  =  error "empty list"
```

В этом случае мы получим более ясное сообщение «Exception: empty list».

Иногда удобно дать имя целому образцу, чтобы использовать в правой части уравнения. Например, рассмотрим определение функции `dropWhile` из Prelude, которая принимает некоторый **предикат** (т. е. функцию типа `a -> Bool`) и список и вычисляет список, который получится, если отбросить начальные элементы, которые удовлетворяют предикату. Например,

```
ghci&gt; dropWhile (&gt;0) [1,2,-23,4,5]
[-23,4,5]
```

Определение может быть таким:

```
dropWhile :: (a -&gt; Bool) -&gt; [a] -&gt; [a]
dropWhile _ [] =  []
dropWhile p (x:xs') | p x       = dropWhile p xs'
                    | otherwise = (x:xs')
```

Образцу (x:xs') можно присвоить свое имя и переписать второе уравнение:

```
dropWhile p xs@(x:xs') | p x       = dropWhile p xs'
                       | otherwise = xs
```

`xs@(x:xs')` – это **@-образец** (по-английски **@** читается как «as»). Он всегда успешно привязывает имя `xs` к выражению, сопоставляемому с `(x:xs')`.

## case

Для сопоставления с образцом можно использовать case-конструкции.

```
1s x = case x of
2        1 -> 4
3        2 -> 3
4        3 -> 2
5        4 -> 1
6        _ -> 0
```

После `case x of` следует цепочка образцов-посылок и следствий, отделяемых знаком **`->`**. `x` будет последовательно сопоставляться с образцами, пока не будет найден подходящий.

Следствия должны иметь одинаковый тип.

Последним случаем по умолчанию может быть `_ ->` ..., так как заполнитель `_` успешно сопоставляется с любым значением.

Остальные определения можно переписать с использованием case. В этом смысле case можно считать базовой конструкцией.

```
-- Запись нескольких уравнений через case
x || y = case (x,y) of
           (False,x) -> x
           (True, _) -> True
-- Запись if-then-else через case
abs x = case x >= 0 of
          True  ->  x
          False -> -x
```

# let и where

Если требуется создать локальные привязки, то используется let:

```
roots a b c =
  let d  = b^2 - 4*a*c
      sd = sqrt d
      x1 = (-b + sd)/(2*a)
      x2 = (-b - sd)/(2*a)
  in (x1, x2)
```

После `let` следует блок с объявлениями, затем `in` и выражение, использующее объявления.

Можно также использовать образцы и объявления функций. Объявления вычисляются рекурсивно, поэтому одно может выражаться через другое.

Конструкции let могут быть вложенными.

Кроме того, есть конструкция where, которая работает схожим образом:

```
roots a b c = (x1, x2) where
  d  = b^2 - 4*a*c
  sd = sqrt d
  x1 = (-b + sd)/(2*a)
  x2 = (-b - sd)/(2*a)
```

Но `where` относится ко всему определению, а не к отдельному выражению. Поэтому `where` часто используется в уравнениях с условиями:

```
f x y z  |  p1 x y z  =  …
         |  p2 x y z  =  …
         |   . . . . . . .
         |  pn x y z  =  …
       where
         …
```

# Выравнивание кода

Мы встретились с элементами языка, предполагающими группировку определений. В Haskell для этого используется **выравнивание**: определения из одной группы должны располагаться одно под другим, а сама группа выделяется отступом.

Группировка определений требуется после ключевых слов `do, let, of, where`. (`do` в действии мы увидим позже; остальные конструкции нам уже знакомы.) Мы уже применяли правильное выравнивание – посмотрите приведенные выше определения с case-of, where и let.

Длинные выражения можно переносить; главное чтобы перенесенная часть начиналась после той колонки, в которой начато определение:

```
1where
2  foo = a + b + c + d
3          + e + f + g + h
4              + i + j + k + l
```

При необходимости блоки можно выделить явным образом. Группа заключается в фигурные скобки, а определения разделяются точкой с запятой:

```
s x = case x of {1 -> 4; 2 -> 3; 3 -> 2; 4 -> 1; _ -> 0}

roots a b c = (x1, x2) where {d = b^2-4*a*c; sd = sqrt d; x1
        = (-b+sd)/(2*a); x2 = (-b-sd)/(2*a)}
```

Однако на практике достаточно использовать выравнивание, и код с правильным выравниванием легче читать.

Если что-то не так, то ghci выдаст сообщение: «Parse error (possibly incorrect indentation)».

# Специальные конструкции для списков

## Диапазоны

Чтобы можно было удобно записывать выражения списков, в Haskell имеются **диапазоны**. Их элементы должны принадлежать классу `Enum`.

Используются следующие методы (в предыдущей статье мы их опустили, чтобы упростить изложение):

```
enumFrom :: a -> [a]
enumFromThen :: a -> a -> [a]
enumFromTo :: a -> a -> [a]
enumFromThenTo :: a -> a -> a -> [a]
```

Базовый синтаксис таков: `[x..y]` вычисляется в список

```
[x, succ x, (succ.succ) x, (succ.succ.succ) x, …, y]
```

Пример:

```
ghci> ['a'..'c']
"abc"
ghci> [1..23]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23]
```

(Значение `['a'..'c']` записано в виде `"abc"`, но это сокращение для `['a','b','c']`.)

Чтобы перечисление происходило с другим шагом или в обратном порядке, используется запись `[x,x'..y]`, где `x'` – элемент, который должен следовать за первым элементом `x`, а `y` – последний элемент.

```
ghci> [1,3..23]
[1,3,5,7,9,11,13,15,17,19,21,23]
ghci> ['z','y'..'a']
"zyxwvutsrqponmlkjihgfedcba"
```

Диапазоны могут соответствовать бесконечным спискам. В ленивом языке это не создает проблем.

```
ghci> head [0..]
0
ghci> take 5 [0..]
[0,1,2,3,4]
ghci> [0..] !! 23
23
```

## Выделение

В аксиоматике теории множеств Цермело–Френкеля имеется аксиома выделения, позволяющая построить множество на основе имеющегося, выбрав элементы, которые соответствуют определенному предикату.

По аналогии имеются специальные выражения для выделения списков. В простейшем виде это записывается так:

```
[e1 | n <- e2]
```

Здесь *e1* и *e2* – произвольные выражения, где *e2* вычисляется в список. Выражение *n <- e2* называется **генератором**. ***n*** будет пробегать все значения из списка *e2* и использоваться в *e1*.

```
ghci> [n^2 | n <- [1..10]]
[1,4,9,16,25,36,49,64,81,100]
```

Генераторов может быть несколько, они перечисляются через запятую:

```
ghci> [(m,n) | m <- [1..3], n <- [1..3]]
[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)]
```

Перестановка генераторов дает тот же список, но с другим порядком элементов:

```
ghci> [(m,n) | n <- [1..3], m <- [1..3]]
[(1,1),(2,1),(3,1),(1,2),(2,2),(3,2),(1,3),(2,3),(3,3)]
```

Это можно сравнить со вложенными циклами. Первый генератор – внешний цикл, а второй – внутренний. В соответствии с этим последующие генераторы могут зависеть от предыдущих. Например, рассмотрим функцию

```
concat' xs = [y | ys <- xs, y <- ys]
```

Если вычислить `concat' [[1,2,3],[4,5,6],[7,8,9]]`, то ys будет пробегать значения `[1,2,3], [4,5,6], [7,8,9]`, а y будет пробегать значения каждого из этих списков. Таким образом, `concat'` работает как уже известная нам функция `concat`.

Собственно **выделение** задействуется, когда мы добавляем **условия**. Например, рассмотрим список троек (x,y,z), таких, что x2 + y2 = z2 и 1 ≤ x,y,z ≤ 15. Все возможные тройки мы получим при помощи трех генераторов `x <- [1..15], y <- [1..15], z <- [1..15]`. Условие x^2+y^2==z^2 добавляется после них:

```
 ghci> [(x,y,z) | x <- [1..15], y <- [1..15], z <- [1..15],
x^2+y^2==z^2]
[(3,4,5),(4,3,5),(5,12,13),(6,8,10),(8,6,10),(9,12,15),(12,5,13),
 (12,9,15)]
```

Следующая функция будет вычислять список делителей числа (включая 1, но исключая само число):

```
factor n = [m | m <- [1..n `div` 2], mod n m == 0]
ghci> factor 100
[1,2,4,5,10,20,25,50]
ghci> factor 23
[1]
```

Теперь можно составить при помощи `factor` список чисел, равных сумме своих множителей (такие числа называются совершенными):

```
ghci> [n | n <- [1..], n == (sum.factor) n]
[6,28,496,8128,…
```

Если число делится только на 1 и на себя, то оно называется *простым*. Можно написать соответствующий предикат:

```
isPrime n = factor n == [1]
ghci> isPrime 100
False
ghci> isPrime 23
True
```

При этом не забывайте, что вычисления ленивые и, если хвост списка `factor n` – не `[]`, то сравнение с `[1]` даст `False`. Полное вычисление `factor n` не требуется.

Отсюда получим список всех простых чисел:

```
ghci> [n | n <- [1..], isPrime n]
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,…
```

(Более красивое решение мы скоро получим при помощи *решета Эратосфена*.)

# Первые примеры

## Отображение списков

Теперь мы покажем несколько популярных примеров функциональных программ.

У нас пока нет полной выразительной мощи языка и его библиотек, но мы описали многие конструкции и надеемся, что читатель уловит общие идеи функционального подхода.

Рассмотрим простую практическую задачу. Пусть у нас имеется список элементов `[a, b, c, …]` и требуется заменить каждый элемент на `[f a, f b, f c, …]`.

Эта операция называется **отображением** (`map`).

```
map f [a, b, c, …] = [f a, f b, f c, …]
```

Для начала можно записать тип необходимой функции:

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

В функциональном языке принято мыслить рекурсивно, и списки сами по себе рекурсивны. Желаемые манипуляции со списком требуется описать относительно головы и хвоста:

- Головной элемент `x` должен быть заменен на `f x`.
- Все элементы в хвосте `xs` должны быть отображены, т. е. `xs` заменяется на `map f xs`.

Базой рекурсии служит пустой список. Здесь, очевидно, действует тривиальное правило:

```
map _ [] = []
```

Шаг рекурсии будет выражать уравнение

```
map f (x:xs) = f x : map f xs
```

Пример вычисления:

```
map (*2) [1,2,3] = map (*2) (1:2:3:[])
                 = (*2) 1 : map (*2) (2:3:[])
                 = 2 : map (*2) (2:3:[])
                 = 2 : (*2) 2 : map (*2) (3:[])
                 = 2 : 4 : map (*2) (3:[])
                 = 2 : 4 : (*2) 3 : map []
                 = 2 : 4 : 6 : map []
                 = 2 : 4 : 6 : []
                 = [2,4,6]
```

Как только мы фиксируем `f`, `map` превращается в конкретное отображение. Например, можно определить

```
doubleElements = map (*2)
negateElements = map negate
```

(Вспомните о каррировании и частичном применении.)

Другие примеры использования:

```
ghci> map (*2) [1,2,3]
[2,4,6]
ghci> map (\ x -> 3*x+2) [1,2,3]
[5,8,11]
ghci> map (>0) [1,2,3]
[True,True,True]
```

Написанную нами функцию можно найти в Prelude.

Это типичная для функциональных языков абстрактная функция высокого порядка. Вместо того чтобы записывать конкретное отображение, мы используем произвольную функцию `f`, которая становится параметром `map`.

Можно пойти еще дальше по пути абстракции. У нас осталась другая конкретная функция – `(:)`. Следующим пунктом был бы вопрос о том, что особенного в `(:)` – возможно, ее стоит заменить произвольной функцией. Если зайти еще дальше, то встанет вопрос, что особенного в списках.

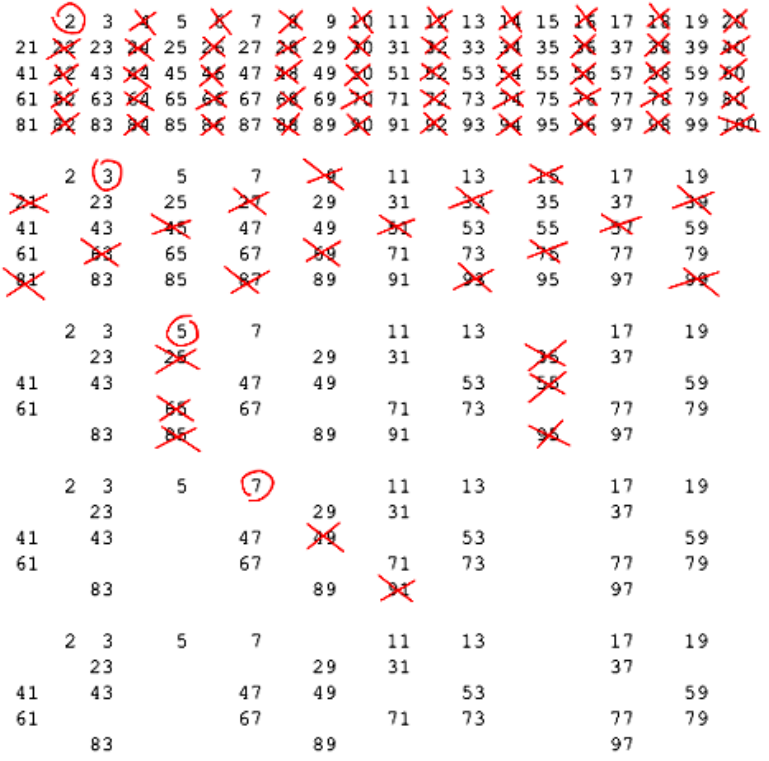
## Решето Эратосфена

Древнегреческому математику Эратосфену Киренскому приписывается следующий алгоритм нахождения простых чисел:

1. Сначала выписываются все целые числа больше 2 (1 простым числом не считается).
2. Из списка выбирается первое простое число $p$ (на первом шаге $p = 2$), и вычеркиваются числа $2p$, $3p$, $4p$, …
3. Далее берется следующее невычеркнутое число $p$, и процесс повторяется.
4. Все оставшиеся в списке числа будут простыми.

Пример просеивания `[2..100]` через решето Эратосфена приведен на рисунке 1.

**Рисунок 1. Пример просеивания [2..100] через решето Эратосфена**



Список всех целых чисел больше 2 можно записать в виде `[2..]`.

Вычеркивание чисел, кратных p, можно записать в виде

```
elim :: Integer -> [Integer] -> [Integer]
elim p xs = [x | x <- xs, mod x p /= 0]
```

Запишем рекурсивную функцию `sieve` («решето»). Она будет принимать список `(p:xs)`, первый элемент (голова) которого **p** должен быть простым. Далее она вычеркивает **p** из хвоста `xs` и снова просеивает результат через решето:

```
sieve :: [Integer] -> [Integer]
sieve (p:xs) = p : sieve (elim p xs)
```

Теперь всё готово. Наш бесконечный список `[2..]` можно просеять через решето, и получится бесконечный список простых чисел:

```
primeNumbers :: [Integer]
primeNumbers = sieve [2..]
```

Например, вычислим первые 23 простых числа:

```
ghci> take 23 primeNumbers
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83
]
```

Если вы попробуете вычислить `primeNumbers`, то список будет печататься бесконечно. Элементы будут выводиться по мере вычисления.

# Быстрая сортировка

Очень часто возникает задача **сортировки**. Исходные данные – некоторый список [x1, …, xn]. Требуется найти список [x'1, …, x'n], состоящий из тех же элементов, такой, что x'1 ≤ … ≤ x'n.

Начнем с типа функции сортировки. Во-первых, это будет функция, переводящая один список в другой, отсортированный, с теми же элементами того же типа. Во-вторых, сортировка подразумевает сравнение элементов, поэтому они должны иметь тип класса `Ord`.

```
quickSort :: (Ord a) => [a] -> [a]
```

Алгоритм быстрой сортировки определяется рекурсивно. Каждый шаг выглядит так:

- Выбрать *ведущий элемент* списка *y*.
- Разбить оставшиеся элементы на *xs*, которые меньше ведущего или равны ему, и на *zs*, которые больше.
- Вставить перед *y* отсортированные *xs*, а после – отсортированные *zs*.

Рекурсия заключается в том, что *xs* и *zs* сортируются тем же алгоритмом.

Как всегда, рекурсивное определение сводит задачу к более простой, а именно сортировке меньших списков. Базовым случаем можно считать сортировку пустого списка:

```
quickSort [] = []
```

Если список не пустой, то в качестве элемента *y* удобно взять его голову. Тогда *xs* и *zs* набираются из хвоста. Через выделение списков это запишется так:

```
xs = [x | x <- ys, x<=y]
zs = [z | z <- ys, z>y]
```

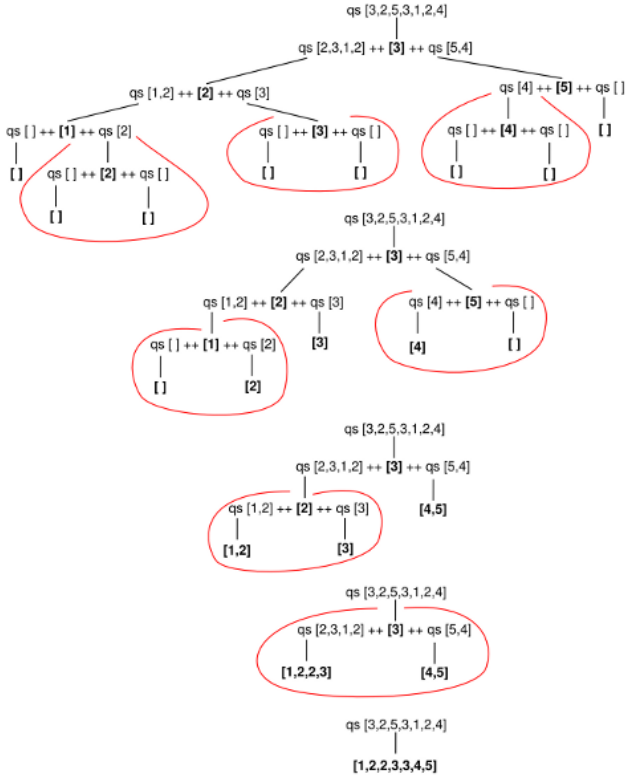
Соединение списков запишется в виде *xs* ++ **[y]** ++ *zs*.

Итак, основным уравнением будет

```
quickSort (y:ys) = xs ++ [y] ++ zs where
  xs = quickSort [x | x<-ys, x<=y]
  zs = quickSort [z | z<-ys, z>y]
```

Всё готово. На рисунке 2 приведен пример вычисления `quickSort [3,2,5,3,1,2,4]`. Рекурсивные вызовы показаны в виде дерева, а конкатенация списков – в виде пошаговой свертки ветвей.

**Рисунок 2. Пример вычисления quickSort [3,2,5,3,1,2,4]**



# Заключение

Мы рассмотрели важные элементы Haskell, которые используются при определении функций, а также специальные выражения для построения списков.

Приведенные примеры функционального кода позволяют проследить использование рекурсии, функций высокого порядка и бесконечных структур данных.

В общем случае, при функциональном подходе:

- Сложное вычисление должно выражаться через композицию простых функций.
- Каждая функция отвечает за свой небольшой элемент вычислений.
- Параметры вычислений выносятся в аргументы функций.
- В соответствии с частичным применением, сначала должны следовать более общие параметры.
- Функции должны быть как можно более абстрактными и общими, пригодными для повторного использования. Часто используемые шаблоны можно оформлять в виде функций высокого порядка.

В качестве упражнения по определению функций и применению рекурсии вы можете записать свои аналоги для `drop, length, null, take, zip` (их описание содержится в предыдущей статье).

По аналогии с описанной в тексте `dropWhile` попробуйте определить `takeWhile`, которая будет вычислять начальный список элементов, удовлетворяющих данному предикату.

Запишите другие полезные функции для работы со списками:

```
 iterate f x = [x, f x, (f.f) x, (f.f.f) x, …]  (бесконечный
1список)
2repeat x = [x,x,x,x,...]  (бесконечный список)
3cycle [x1,…,xn] = [x1,…,xn, x1,…,xn, x1,…,xn, x1,…,xn, …]  (n >=
41)
 replicate n x = [x,x,x,x,…]  (n элементов)
```

**На наступній лекції ми підведемо підсумки розвитку функціональних мов програмування.**