Основы языка Haskell

А. В. Дубанов

Кафедра ИУ-9 МГТУ им. Н. Э. Баумана

2014 г.

Haskell

Haskell — чистый функциональный язык программирования общего назначения.

Функциональное программирование

Функциональное программирование (ФП) — раздел дискретной математики и парадигма программирования, в которой процесс вычислений рассматривается как вычисление значений математических функций.

Функция

В программировании

Функция — процедура (подпрограмма), (возможно) принимающая некоторые аргументы и (обязательно) возвращающая значение.

В математике

Функция — это закон или правило, согласно которому каждому элементу из областью определения ставится в соответствие определенный элемент из области значений функции.

Чистая функция

- Детерминирована (значение функции однозначно определяется значениями ее аргументов).
- Не имеет побочных эффектов (не взаимодействует со средой выполнения: не осуществляет ввода-вывода, не вызывает исключений и т.п.).

Следствия

- Слабая связность компонентов программы (повторное использование кода, юнит-тестирование).
- Мемоизация результатов вычислений.
- Декларативность программы: не требуется явно указывать последовательность вычислений.
- Возможность организации параллельных вычислений.

Особенности Haskell

- Использование чистых функций.
- Результаты недетерминированных функций и функций с побочными эффектами инкапсулируются в монады.
- Имеется подмножество языка для императивного программирования.

Монада

Монада (в программировании):

- Абстракция линейной цепочки связанных вычислений.
- Контейнерный тип для возвращаемых значений недетерминированных функций и функций с побочными эффектами.

Особенности Haskell

- Мемоизация результатов вычилений.
- Ленивые и нестрогие вычисления.
- Многопоточность.
- Порядок вычислений определяется компилятором.

Особенности Haskell

- Широкое использование математической нотации, обилие синтаксического сахара.

Особенности Haskell

- Развитая система типов: полиморфные типы, алгебраические типы (параметризуемые, рекурсивные и т.д.).
- Строгая статическая типизация = надёжность кода.
- Автоматический вывод типов (модель типизации Хиндли-Мильнера).

Особенности Haskell

- Функции высшего порядка. Операции над функциями.
- Все сущности программы могут рассматриваться как функции. Переменных нет!
- Язык стандартизован (Haskell-98, Haskell-2010).
- ...

Средства разработки

Haskell Platform

- GHC — Glasgo Haskell Compiler
- GHCi (WinGHCi) — GHC interactive (REPL)
- runghc (runhaskell) — non-interactive interpreter
- cabal
- haddock
- Libraries

haskell.org

- Haskage
- Hoogle

Литература

1. haskell.org
2. M. Lipovača. Learn You a Haskell for a Great Good! learnyouahaskell.com
3. B. O'Sullivan, D. Stewart, J. Goerzen. Real World Haskell. book. realworldhaskell.org/read/
4. Р. Душкин. Функциональное программирование на языке Haskell.
5. Статьи на www.rsdn.ru
6. А. Бешенов. Функциональное программирование на Haskell. Статьи на www.ibm.com/developerworks/ru/library

Выражения

Выражения записываются в математической нотации. Операторы применяются в инфиксной форме. Порядок вычислений задается приоритетом операторов и круглыми скобками.

```
> 2 + 2 * 2
6
> (2 + 2) * 2
8
```

Применение функции к аргументам

```
Prelude> mod 10 3
1
```

Применение функции к результату вычисления другой функции:

```
Prelude> log (exp 1)
1.0
```

Значение функции без аргументов:

```
Prelude> pi
3.141592653589793
```

Применение функции к аргументам

Применение функции к аргументам имеет наивысший приоритет:

```
Prelude> div 4 2 + div 5 2
4
```

Применение функции к отрицательному числу или аргументом с унарным минусом:

```
Prelude> abs (-1)
1
Prelude> sin (-pi)
-1.2246467991473532e-16
```

Standard Prelude

Модуль `Prelude` по умолчанию загружается при запуске GHCi и доступен GHC.

Тип выражения

В Haskell каждое выражение имеет тип. Если тип не указан в программе, компилятор пытается его вывести (обычно успешно).

`:t` *выражение*, `:type` *выражение* — команда GHCi для получения типа выражения.

```
> :t 'c'
'c' :: Char

> :t id
id :: a -> a

> :t (2+2)
(2+2) :: Num a => a
```

Команды GHCi

`:info` *имя_функции*	Информация о функции
`:i` *имя_функции*	
`:quit`	Выход (Ctrl+D)
`:q`	
`:help`	Помощь по командам GHCi
`:h`	
`:?`	

Сигнатура

Сигнатура — обозначение типа.

выражение `::` *тип*

`::` — «имеет тип»

Примеры сигнатур

Сигнатуры значений

```
'c'   :: Char
"abc" :: [Char]
1     :: Num a => a
1.0   :: Fractional a => a
```

Сигнатуры функций

```
not  :: Bool -> Bool
abs  :: Num a => a -> a
mod  :: Integral a => a -> a -> a
sqrt :: Floating a => a -> a
map  :: (a -> b) -> [a] -> [b]
```

a — параметр типа

Примеры сигнатур

Сигнатуры операторов

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

> :t (&&)
(&&) :: Bool -> Bool -> Bool
```

Оператор — функция двух аргументов, применяемая в инфиксной форме.

Как определены операторы

Обычно для операторов определены ассоциативность и приоритет.

```
> :i (+)
class Num a where
  (+) :: a -> a -> a
  ...
    -- Defined in ‘GHC.Num’
infixl 6 +

> :i (*)
class Num a where
  ...
  (*) :: a -> a -> a
  ...
    -- Defined in ‘GHC.Num’
infixl 7 *
```

Как определены операторы

```
> :i (++)
(++) :: [a] -> [a] -> [a]    -- Defined in ‘GHC.Base’
infixr 5 ++

> :i (==)
class Eq a where
  (==) :: a -> a -> Bool
  ...
    -- Defined in ‘GHC.Classes’
infix 4 ==
```

Соглашения

- Имена функций пишутся со строчной буквы. При необходимости используется camelCase, символ _ использовать не рекомендуется.
- Имена типов, классов типов, конструкторов типов и модулей пишутся с заглавной буквы.

Встроенные типы

- Пустой (единичный) тип.
- Логический тип.
- Числовые типы.
- Кортежи.
- Списки (включая строки).

Пустой тип

```
() :: ()
```

Как правило, его возвращают функции, применяющиеся только ради своих побочных эффектов. Например:

```
putStr :: String -> IO ()
```

Логический (булевый) тип

```
True  :: Bool
False :: Bool

(&&) :: Bool -> Bool -> Bool
(||) :: Bool -> Bool -> Bool

infixr 3 &&
infixr 2 ||

not :: Bool -> Bool
```

Логические значения возвращают операторы проверки на равенство, операторы сравнения и предикаты. Возвращаемые значения используются в т.ч. управляющими конструкциями.

Операторы проверки на равенство

```
(==) :: Eq a => a -> a -> Bool
(/=) :: Eq a => a -> a -> Bool
```

Оба значения должны быть одного типа.

Операторы сравнения

```
(>)  :: Ord a => a -> a -> Bool
(<)  :: Ord a => a -> a -> Bool
(>=) :: Ord a => a -> a -> Bool
(<=) :: Ord a => a -> a -> Bool
```

Оба значения должны быть одного типа.

Символы

`:: Char`

Одиночные символы Unicode

«Пробельные символы»: `' '`, `'\t'`, `'\n'`, `'\r'`, `'\f'`, `'\v'`.

Экранирование \: `'\\'`

Классификация символов: модуль `Data.Char`

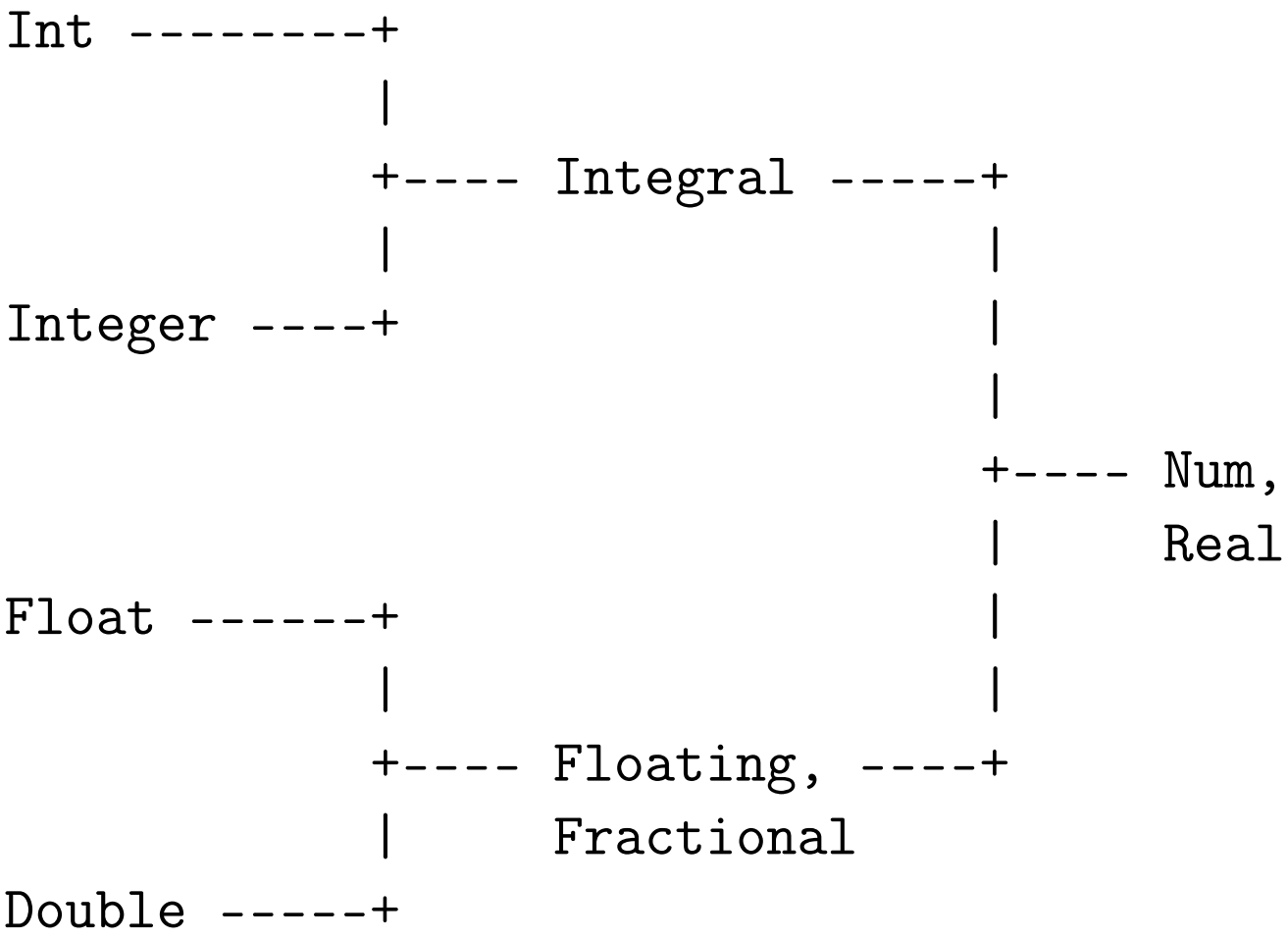
Простые числовые типы

`Int`	«Аппаратное» целое*
`Integer`	Целое произвольной разрядности
`Double`	С плавающей запятой двойной точности (!)
`Float`	С плавающей запятой**

* Обычно используется для индексов. Не менее 30 бит. См: `minBound :: Int`, `maxBound :: Int`.

** Для совместимости с C.

Классы числовых типов



Преобразования числовых типов

```
fromIntegral :: (Integral a, Num b) => a          -> b
fromRational ::  Fractional a       => Rational -> a

toInteger  :: Integral a => a -> Integer
toRational :: Real     a => a -> Rational
```

Арифметические операторы

Левоассоциативные

```
(+) :: Num a => a -> a -> a
(-) :: Num a => a -> a -> a  -- невозможно получить через :t
(*) :: Num a => a -> a -> a

subtract :: Num a => a -> a -> a  -- замена бинарного (-)
negate   :: Num a => a -> a       -- замена  унарного (-)

(/) :: Fractional a => a -> a -> a
```

Арифметические операторы

Левоассоциативные

```
div :: Integral a => a -> a -> a
mod :: Integral a => a -> a -> a
```

Функции `div` и `mod` могут использоваться в инфиксной форме. Для них определен приоритет такой же, как и для `/`. Например:

```
5 `div` 2
```

Операторы возведения в степень

Правоассоциативные

```
(^)  :: (Integral b, Num a) => a -> b -> a
(^^) :: (Fractional a, Integral b) => a -> b -> a
(**) :: Floating a => a -> a -> a
```

Числовые функции

См. документацию по `Prelude` в Hoogle.

Кортежи

Кортежи (tuples) — последовательности элементов (возможно, *разных* типов), фиксированной длины. Например:

```
('c', 1.0, 1, True)
```

Доступ к отдельным элементам кортежа: см. «Сопоставление с образцом»

Кортежи

Кортежи считаются принадлежащими одному типу, если у них совпадают число и типы элементов по порядку. Такие кортежи можно сравнивать поэлементно:

```
> (2, 2, 3) < (2, 3, 2)
True
> (3, 2, 2) < (2, 3, 2)
False
```

Вопрос: что есть пустой кортеж?

Списки

Списки (lists) — последовательности произвольной длины из элементов *одинакового* типа. Например:

```
[]                 :: [a]                  -- Пустой список
['a', 'b', 'c']    ::           [Char]
[1, 2, 3]          :: Num t => [t]
[[1,2], [3,4,5]] :: Num t => [[t]]    -- Вложенные списки

[[1,2],3]  -- ОШИБКА!
```

Операции над списками

```
(++)   :: [a] -> [a] -> [a]    -- Конкатенация
(!!)   :: [a] -> Int -> a      -- Получение эл-та по индексу
length :: [a] -> Int           -- Число элементов

[1,2] ++ [3,4]  -- [1,2,3,4]
[1,2,3] !! 1    -- 2
length [1,2,3]  -- 3
```

Сложность: $O(n)$

Операции над списками

Если [1,2,3,4] — список, то:

- 1 — его голова (head),
- [2,3,4] — его хвост (tail).

```
head [1,2,3,4]  -- 1
tail [1,2,3,4]  -- [2,3,4]
last [1,2,3,4]  -- 4
init [1,2,3,4]  -- [1,2,3]
```

Сравнение списков

```
[1,2,3] == [1,2,3]  -- True
[1,2,3] == [1,2,1]  -- False
[1,2,3] <= [2,2,3]  -- True
[1,2]   <= [1,2,3]  -- True
```

Строки

Строки — списки символов.

```
x :: String  {- эквивалентно -}  x :: [Char]
```

В GHCi:

```
> ['a', 'b', 'c']
"abc"
> "abc" ++ "def"
"abcdef"
> "abc" !! 1
'b'
> length "abcd"
4
```

Функции

В программе:

```
area r = 4 * pi * r ^ 2
```

В GHCi:

```
let area r = 4 * pi * r ^ 2
```

Замечания

Знак = означает связывание выражения с именем.

Запись

```
area r = 4 * pi * r ^ 2
```

является сокращенной формой записи

```
area = \r -> 4 * pi * r ^ 2
```

`\r -> {- ... -}` — анонимная функция, `\` соответствует λ в записи

$$\lambda r.4\pi r^2$$

Константы

Константу в программе можно определить как функцию без аргументов:

```
unit = 1
```

Сигнатура функции

В программе тип функции может быть указан явно — с помощью сигнатуры:

```
area :: Double -> Double
area r = 4 * pi * r ^ 2
```

Когда писать сигнатуры?

- Когда компилятор не может определить тип.
- Когда надо ограничить диапазон значений.
- Когда не требуется полиморфная функция*.
- Для лучшей документированности кода.

* Например, `mySort :: [Int] -> [Int]` вместо `mySort :: [a] -> [a]`.

Загрузка определений в GHCi

`:load` *имя_файла*	Загрузить файл
`:load "`*путь_к_файлу*`"`	
`:l ...`	
`:r`, `:reload`	Перезагрузить последний загруженный файл

Сопоставление с образцом

Многие функции естественно определяются через разбор различных вариантов или могут быть заданы таблицей.

С вариантами функции:

```
numeral :: Integer -> String
numeral 0 = "zero"
numeral 1 = "one"
numeral 2 = "two"
numeral _ = "any number"
```

Сопоставление с образцом

Правила

- Сигнатура — одна для всех вариантов.
- Перечисление образцов — от частного к общему.
- Области определения вариантов не должны пересекаться.
- Должны быть предусмотрены все возможные варианты.

Сопоставление с образцом

Внутри функции:

```
numeral :: Integer -> String
numeral n = case n of
    0 -> "zero"
    1 -> "one"
    2 -> "two"
    _ -> "any number"
```

As-pattern (@)

As-pattern `@` обеспечивает доступ к элементам составных типов при сопоставлении с образцом.

Программа:

```
test t@(x, y) =  print x
              >> print y
              >> print t
```

GHCi:

```
> test (1,2)
1
2
(1,2)
```

Как написать функцию

Пример

Написать функцию для вычисления $n!$.

1. Запишем определение факториала в математической нотации:

$$n! = \begin{cases} 1 & n = 0, \\ n \cdot (n-1)! & n > 0. \end{cases}$$

2. Просто перепишем это определение на языке Haskell:

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Условные конструкции

- `if-then-else`
- Охранные условия (guards)

if–then–else

$$|x| = \begin{cases} -x & x < 0, \\ x & x \geqslant 0. \end{cases}$$

```
abs' x = if x < 0 then -x else x
```

Альтернатива if–then–else

```
if' :: Bool -> a -> a -> a
if'    True    x    _  = x
if'    False   x    _  = x

abs' x = if' (x < 0) (-x) x
```

Охранные условия

$$\operatorname{signum}\, x = \begin{cases} -1 & x < 0, \\ 0 & x = 0, \\ 1 & x > 0. \end{cases}$$

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

или

```
signum' x | x <  0      = -1
          | x == 0      =  0
          | otherwise   =  1
```

Переопределение функции

Функция может быть переопределена, если новое определение не вызывает конфликтов типов.

Локальные определения

Например, необходимо найти действительные корни квадратного уравнения вида $ax^2 + bx + c = 0$. Будем считать, что эти корни существуют. На Haskell можем записать:

```
quadratic 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)
```

(предпочтительно) или

```
quadratic 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)
```

Оформление кода

Отступы существенны и контролируются компилятором, т.к. отступы являются синтаксическим сахаром (. . .).

Руководства по стилю кодирования:

- www.haskell.org/haskellwiki/Programming_guidelines
- stackoverflow.com/questions/1983047/good-haskell-coding-standards
- **github.com/tibbe/haskell-style-guide/blob/master/haskell-style.md**

Оператор применения функции к аргументам ($)

Пусть `f` — функция. Тогда запись `f $ x` эквивалентна `f x` с тем отличием, что оператор $ имеет самый низкий приоритет среди всех операторов. Используется для сокращения числа круглых скобок при записи выражений:

```
f $ g $ h x  {- экв. -}  f (g (h x))
f $ x + g x  {- экв. -}  f (x + g x)
```

Первая программа

```haskell
-- hello.hs

module Main where

main :: IO ()
main = putStrLn "Hello, World!"
```

или, если вся программа помещается в одном файле:

```haskell
-- hello.hs

main = putStrLn "Hello, World!"
```

Сигнатура не обязательна.

Выполнение в GHCi

```
> :l "hello.hs"
> :main
Hello, World!
```

Компиляция и выполнение

```
$ ghc --make -o hello hello.hs
$ ./hello
Hello, World!
```

Выполнение как скрипта

Файл с программой:

```
#!/usr/bin/runhaskell

main = putStrLn "Hello, World!"
```

Bash:

```
$ chmod +x hello.hs
$ ./hello.hs
Hello, World!
```

Задание

Напишите и отладьте в среде GHCi:

1. Функцию для нахождения наибольшего общего делителя двух произвольных целых чисел (например, по алгоритму Евклида). Сравните результаты работы своей функции с результатами встроенной функции `gcd`.
2. Функцию, проверяющую целое число на простоту методом перебора делителей. По-возможности постарайтесь избежать вычисления квадратного корня.
3. Функцию наименьшего общего кратного двух целых чисел. Сравните результаты работы своей функции с результатами встроенной функции `lcm`.

Решения

НОД по Евклиду

```
gcd' :: Integer -> Integer -> Integer
gcd' 0 0 = error "gcd' 0 0 is undefined"
gcd' a b | b == 0     = a
         | otherwise  = gcd' b (mod a b)
```

Чего здесь не хватает?

Решения

Проверка числа на простоту

```
-- | Проверка на простоту методом поиском делителя

prime :: Integral a => a -> Bool
prime n = n == divisor n 2

-- | Поиск делителя

divisor :: Integral a => a -> a -> a
divisor n m | mod n m == 0  = m  --  *
            | m * m   == n  = n  --  <==> m >= sqrt n
            | otherwise     = divisor n (m+1)
```

* Это условие должно быть первым

Решения

НОК

```
lcm' :: Integer -> Integer -> Integer
lcm' a b = div (abs (a * b)) (gcd' a b)
```

Функции высшего порядка (higher-order functions)

. . . или функции как объекты 1-го класса.

Операции над функциями:

- Частичное применение;
- Карринг;
- Композиция функций.

Частичное применение

Частичное применение функции к аргументам — способ «зафиксировать» часть аргументов функции.

Пусть определена функция:

```
f x y = x * x + y * y
```

Предположим, в программе необходимо вычислять эту функцию для `x = 2` и разных `y`. Для этого можем определить функцию:

```
g = f 2
```

Частичное применение

Имеем:

```
f x y = x * x + y * y
g = f 2
```

Функция g будет функцией одного аргумента:

```
> :t g
g :: Integer -> Integer
> g 2
8
```

т.к. `g 2` эквивалентно `f 2 2`.

Рекомендация

Тот аргумент, который будет меняться чаще, записывайте последним.

Функция `flip`

Если в ранее определенной функции порядок аргументов не удобен для частичного применения, то их можно поменять местами с помощью функции `flip` из `Prelude`:

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

f x y = x * x - y * y
g = (flip f) 2  -- \y -> f 2 y
```

Карринг

Каррирование, карринг (currying) — преобразование функции нескольких аргументов в функцию, принимающую свои аргументы по одному. В Haskell все функции по умолчанию каррированы.

Пусть определена функция:

```
f x y z = x * x + y * y + z * z
```

Без синтакического сахара это определение может быть записано так:

```
f = \x -> \y -> \z -> x * x + y * y + z * z
```

Отсюда, корректны будут следующие выражения:

```
((f 10) 20) 30
g = f 10 20
```

Функции curry/uncurry

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

Пример:

```
> uncurry max (2, 3)
3
```

Композиция функций

$(F \circ G)(x) = F(G(x))$

`(f . g)(x) ≡ f (g x) ≡ f $ g x`

Пример:

$f(x) = |\sin x|$

```
f = abs . sin
```

При определении функции через композицию аргумент писать **не нужно**!

Операторы

Операторы — функции двух аргументов. Применение оператора как функции:

```
> (&&) True True
True
```

Применение функции как опреатора:

```
> 10 `min` 5
5
```

Замечание

Важно! Использование функции как оператора без определения приоритета может привести к ошибкам:

```
> let plus x y = x + y
> 2 `plus` 2 * 2
8
```

Приоритет и ассоциативность операторов

```
infixl 6 -   -- Приоритет 6, левоассоциативный
infixr 8 ^   -- Приоритет 8, правоассоциативный
infix  4 ==  -- Приоритет 4, без указания ассоциативности
```

Приоритет = 0 .. 9, наивысший приоритет 10 имеет применение функции к аргументам.

Приоритет операторов в Prelude

Приоритет	Операторы			
10	Применение ф-ции к арг-там			
9	(.)	(!!)		
8	(**)	(^)	(^^)	
7	(*)	(/)	`div`	`mod`
6	(+)	(-)		
5	(++)	(:)		
4	`elem`	`notElem`		
3	(&&)			
2	(||)			
1	(>>=)	(=<<)	(>>)	
0	($)	($!)		

Пример

Определение оператора `($)` в `Prelude`:

```
($) :: (a -> b) -> a -> b
f $ x =  f x
infixr 0  $
```

Пример

Определим функцию для вычисления среднего двух чисел и определим для нее приоритет как для оператора:

```
avg x y = (x + y) / 2
infix 7 `avg`
```

Вариант:

```
x `avg` y = (x + y) / 2
infix 7 `avg`

> 1 + 3 `avg` 5
5.0
```

Частичное применение операторов

Сечение (section) — частичное применение операторов.

```
inc  = (+1)  -- inc  x = x + 1
sqr  = (^2)  -- sqr  x = x ^ 2
rec  = (1/)  -- rec  x = 1 / x
half = (/2)  -- half x = x / 2
```

Особенности оператора (-)

```
f = (1-)  -- f x = 1 - x
f = (-1)  -- не является сечением, (-1) :: Integer

-- Определение унарного декремента
-- через частичное применение:

dec = flip subtract 1  -- dec x = 1 - x
```

Оператор `(:)`

Если оператор `(:)`:

- стоит в уравнении *справа* от =, то он является конструктором списка из одиночного элемента (головы) и списка (хвоста),
- стоит в уравнении *слева* от =, то он является частью образца списка, где отделяет голову от хвоста.

Конструктор списка

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

Т.е. конструктор списка `(:)` в Haskell аналогичен `cons` в языках семейства Lisp.

Образцы списков

```
> let test lst@(x:xs) = print lst >> print x >> print xs
> test [1,2,3]
[1,2,3]
1
[2,3]
> test [1]
[1]
1
[]
> let ys = [1,2,3]
> let (x:xs) = ys
> x
1
> xs
[2,3]
```

Рекурсивная обработка списков

Функция `map` в `Prelude`:

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs

> map (+1) [1,2,3]
[2,3,4]
```

Пример

Функция для вычисления среднего арифметического чисел в списке:

```
sum' []     = 0
sum' (x:xs) = x + sum' xs

average xs = sum' xs / n
    where n = fromIntegral $ length xs
```

Примеры образцов списков

```
g []        = "Пустой список"
g [_]       = "Список строго из одного элемента"
g [_, _]    = "Список строго из двух элементов"
g [_, _, _] = "Список строго из трех элементов"
g (_:_:_:_) = "Список из трех или более элементов"
g (_:_:_)   = "Список из двух или более элементов"
g (_:_)     = "Список из одного или более элементов"
g  _        = "Любой другой список"
```

Вместо заполнителей могут быть формальные аргументы.

Какую сигнатуру будет иметь эта функция?

Некоторые функции для обработки списков

```
reverse :: [a] -> [a]
take    :: Int -> [a] -> [a]
drop    :: Int -> [a] -> [a]

reverse [1,2,3]  -- [3,2,1]
take 3 "abcdef"  -- "abc"
drop 2 "abcdef"  -- "cdef"
```

Некоторые функции для обработки списков

```
elem    :: Eq a => a -> [a] -> Bool
notElem :: Eq a => a -> [a] -> Bool

infix 4 `elem`
infix 4 `notElem`

2 `elem`    [1,2,3]  -- True
0 `notElem` [1,2,3]  -- True
```

Некоторые функции для обработки списков

```
splitAt :: Int -> [a] -> ([a], [a])

concat [[1,2], [3,4]]  -- [1,2,3,4]
splitAt 3 "abcdef"     -- ("abc","def")


map    :: (a -> b)    -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]

map    abs [-1, -2, -3]      -- [1,2,3]
filter odd [0,1,2,3,4,5,6]   -- [1,3,5]
```

Функции lines/unlines и words/unwords

```
lines :: String -> [String]
words :: String -> [String]

unlines :: [String] -> String
unwords :: [String] -> String

> lines "line 1\n line 2\n\n line 4"
["line 1"," line 2","","," line 4"]
> unlines ["line 1","line 2", "line 3"]
"line 1\nline 2\nline 3\n"

> words "word1 word2  word3\tword4\nword5"
["word1","word2","word3","word4","word5"]
> unwords ["word1","word2","word3"]
"word1 word2 word3"
```

Модуль Data.List

Импорт модуля: - В программе: `import Data.List` - В GHCi:`:m + Data.List`

```
isPrefixOf :: Eq a => [a] -> [a] -> Bool
isSuffixOf :: Eq a => [a] -> [a] -> Bool
isInfixOf  :: Eq a => [a] -> [a] -> Bool
```

```
> isInfixOf "def" "abcdefgh"
True
```

Свертка списков

Свёртка списка (folding, reduce, accumulate) — преобразование списка к единственному значению путем применения к элементам списка функции или оператора.

Левоассоцитивная свертка (для операций с нейтральным элементом слева):

$$(\cdots(((x_0 \bigstar x_1) \bigstar x_2) \bigstar x_3) \bigstar x_4 \ldots x_{n-1} \bigstar x_n)$$

Правоассоциативная свертка (для операций с нейтральным элементом справа):

$$x_1 \bigstar (x_2 \bigstar (x_3 \bigstar x_4 \ldots x_{n-1} \bigstar (x_n \bigstar x_0) \cdots))$$

$\bigstar$ — бинарная операция, x_0 — стартовое значение (нейтральный элемент), $x_1 \ \ldots \ x_n$ — элементы списка.

Свертка списков

Левоассоцитивная свертка:

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

foldl f x0 [x1, x2, ..., xn]
    == (...((x0 `f` x1) `f` x2) `f`...) `f` xn
```

Правоассоциативная свертка:

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

foldr f x0 [x1, x2, ..., xn]
    == x1 `f` (x2 `f` ... (xn `f` x0)...)
```

Свертка списков

Свертка без стартового значения (только для непустых списков):

```
foldl1 :: (a -> a -> a) -> [a] -> a
foldr1 :: (a -> a -> a) -> [a] -> a

> foldl1 (+) [1,1,1]
3
> foldr1 (^) [2,2,2]
16
```

Специальные виды свертки

```
and [True, True , True]   -- True
or  [True, False, False]  -- True

sum     [1,2,3,4]  -- 10
product [1,2,3,4]  -- 24

minimum [1,2,3,4]  -- 1
maximum [1,2,3,4]  -- 4

any even [1,2,3]  -- True
all even [1,2,3]  -- False

concat [[1,2], [3,4]]  -- [1,2,3,4]
```

Специальные виды свертки

При применении к пустому списку эти функции свертки возвращают нейтральный элемент:

```
and     []  ==  True
or      []  ==  False
sum     []  ==  0
product []  ==  1
concat  []  ==  []
```

Ассоциативные списки

Ассоциативные списки — списки пар «ключ–значение».

```
hosts =
    [ ("yandex.ru"    , "213.180.204.11")
    , ("google.ru"    , "173.194.32.183")
    , ("mail.ru"      , "94.100.180.199")
    , ("linux.org.ru", "217.76.32.61")
    , ("habrahabr.ru", "212.24.43.44")
    ]
```

Собрать/разобрать:

```
> zip [1,2,3] ["one","two","three"]
[(1,"one"),(2,"two"),(3,"three")]
> unzip [(1, "one"), (2, "two"), (3, "three")]
([1,2,3],["one","two","three"])
```

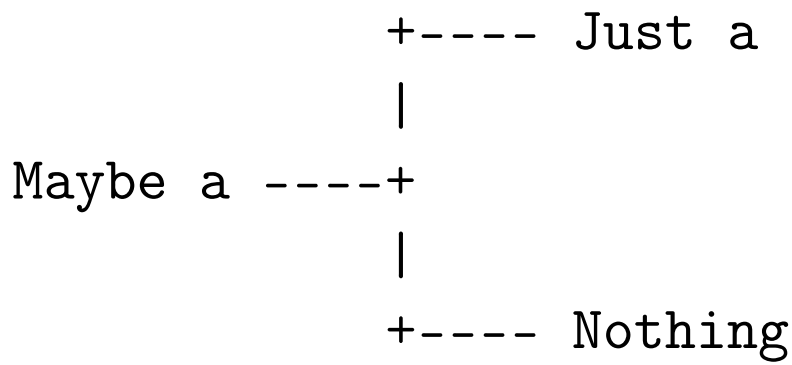
Требование к ключу: быть воплощением класса `Eq`, чтобы по нему можно было осуществлять поиск.

Поиск по ключу

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b

> lookup 1 [(1,"one"),(2,"two"),(3,"three")]
Just "one"
> lookup 0 [(1,"one"),(2,"two"),(3,"three")]
Nothing
```

Монада Maybe



Поиск и Maybe

```
hosts =
    [ ("yandex.ru"    , "213.180.204.11")
    , ("google.ru"    , "173.194.32.183")
    , ("mail.ru"      , "94.100.180.199")
    , ("linux.org.ru", "217.76.32.61")
    , ("habrahabr.ru", "212.24.43.44")
    ]

ip :: String -> String
ip host = case lookup host hosts of
    Just address -> address
    Nothing      -> "Not found!"

> ip "mail.ru"
"94.100.180.199"
> ip "yahoo.com"
"Not found!"
```

Арифметические последовательности

Если тип принадлежит классу `Enum`, то:

```
> [1 .. 10]
[1,2,3,4,5,6,7,8,9,10]

> ['a' .. 'z']
"abcdefghijklmnopqrstuvwxyz"

> [10 .. ]   -- бесконечный список целых чисел, начиная с 10
```

Пример

Факториал с использованием генератора арифметической последовательности и свёртки:

```
factorial n = product [1 .. n]
```

List Comprehension

List Comprehension — получение списка элементов, удовлетворяющих условию.

Список квадратов всех четных x, таких, что $x \in \mathbb{Z}$ и $x \in [0; 10]$:

```
> [x ^ 2 | x <- [0..10], even x]
[0,4,16,36,64,100]
```

www.haskell.org/onlinereport/haskell2010/haskellch3.html#x8-400003.10

Пример

Быстрая сортировка:

```
qsort []     = []
qsort (x:xs) = smaller ++ [x] ++ bigger
  where
    smaller = qsort [a | a <- xs, a <= x]
    bigger  = qsort [a | a <- xs, a >  x]
```

Является ли такая реализация на самом деле быстрой?

Бесконечные списки

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

Программа не будет зацикливаться, если она использует конечное число элементов бесконечного списка (в силу «ленивой» модели вычислений).

```
> take 5 $ repeat 0
[0,0,0,0,0]
> drop 5 $ take 10 $ cycle [1,2,3]
[3,1,2,3,1]
```

См. также: `iterate`, `replicate`.

Ввод-вывод

Высокоуровневые функции чтения из `stdin`:

```
getChar     :: IO Char
getLine     :: IO String
getContents :: IO String  -- lazy
```

Высокоуровневые функции записи в `stdout`:

```
putChar  :: Char   -> IO ()
putStr   :: String -> IO ()
putStrLn :: String -> IO ()
```

Модуль `System.IO` реализует ввод-вывод в стиле C, требует использования подмножества языка для императивного программирования.

Операторы связывания и функция return

Минимальное полное определение монады включает в себя:

```
return :: Monad m =>    a -> m a
(>>=)  :: Monad m => m a -> (a -> m b) -> m b  -- bind
```

```
> return "a string" >>= putStrLn
a string
> :t return "a string" >>= putStrLn
return "a string" >>= putStrLn :: IO ()
```

Операторы связывания и функция return

Часто используется еще один оператор связывания — оператор следования:

```
(>>) :: Monad m => m a -> m b -> m b  -- then

> putStrLn "one" >> putStrLn "two"
one
two
> :t (putStrLn "one" >> putStrLn "two")
(putStrLn "one" >> putStrLn "two") :: IO ()
```

Функции show и read

```
show :: Show a => a -> String
read :: Read a => String -> a
```

```
> show "abc"
"\"abc\""
> show [1,2,3]
"[1,2,3]"
```

Следствие. Если не определен тип результата read, возникает ошибка:

```
> read "123"
-- (Сообщение об ошибке) --

> (read "123") :: Integer
123
```

См. также: др. методы класса Read, в т.ч. lex.

Функция `print`

Функция `print` может быть определена так:

```
print x = putStrLn $ show x
```

Пример

Вывод списка «в столбик»:

```
printList :: Show a => [a] -> IO ()
printList []     = return ()
printList (x:xs) = print x >> printList xs

> printList [1,2,3]
1
2
3
```

Пример

Напишем программу, которая вычисляет и выводит среднее арифметическое чисел, введенных с консоли.

Главная функция:

```
main = getLine >>= {- вычисления -} >>= putStrLn
```

Чистая функция для вычисления среднего арифметического:

```
average xs = sum xs / n
    where n = fromIntegral $ length xs
```

Нужна функция-адаптер.

Пример

Адаптер между вводом-выводом и вычислениями:

```
main = getLine >>= wrapper >>= putStrLn

wrapper = return . show . average . map read . words
--        5        4       3          2          1
```

Композицию читаем «с конца»:

1. Разбивка строки на подстроки по пробелам.
2. Преобразование списка строк в список чисел.
3. Вычисление среднего арифметического.
4. Преобразование числа в строку.
5. Упаковка в монаду.

Пример

Среднее арифметическое чисел, введенных с консоли, код целиком:

```
main = getLine >>= wrapper >>= putStrLn

average xs = sum xs / n
    where n = fromIntegral $ length xs

wrapper = return . show . average . map read . words
```

Работа:

```
> :main
1 2 3
2.0
```

Высокоуровневые запись в файл и чтение из файла

```
readFile   :: FilePath -> IO String

writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
```

Пример: запись и чтение таблицы

Таблица в программе — список списков:

```
table =
    [ [  0,   1,   2,   3]
    , [100, 101, 102, 103]
    , [ -1,  -2,  -3,  -4]
    ]
```

Таблица в текстовом файле:

```
0 1 2 3
100 101 102 103
-1 -2 -3 -4
```

Пример: запись и чтение таблицы

Запись таблицы:

```
saveTable :: Show a => FilePath -> [[a]] -> IO ()
saveTable fp xss = writeFile fp $
    unlines $ map unwords $ map (map show) xss
```

Пример: запись и чтение таблицы

Чтение таблицы:

```
loadTable :: Read a => FilePath -> IO [[a]]
loadTable fp = readFile fp >>=
    return . map (map read) . map words . lines
```

Пример: запись и чтение таблицы

Выражение

```
> loadTable "./table.tab" >>= print
```

вычисляться не будет, т.к. в этом случае нельзя вывести тип, который должна вернуть функция `read`. Поэтому:

```
loadTableOfIntegers :: FilePath -> IO [[Integer]]
loadTableOfIntegers = loadTable
```

Тогда:

```
> loadTableOfIntegers "./table.tab" >>= print
[[0,1,2,3],[100,101,102,103],[-1,-2,-3,-4]]
```

Аргументы командной строки

```
getArgs :: IO [String]
```

Демонстрация:

```
import System.Environment (getArgs)

main = getArgs >>= mapM_ print
```

```
> :main one two three
"one"
"two"
"three"
> :main "first arg" "second arg"
"first arg"
"second arg"
```

В том же модуле — всё о среде исполнения: имя файла программы, переменные окружения и т.д.

Пример: аналог утилиты grep

Утилита grep читает из стандартного потока ввода строки символов и выводит в стандартный поток вывода те строки, в которых встречается подстрока, переданная как аргумент командной строки:

```
$ grep Haskell quick-haskell.md
% Основы языка Haskell
Haskell
Haskell --- чистый функциональный язык программирования
Особенности Haskell
...
```

Пример: аналог утилиты grep

Нам потребуются:

```
import System.Environment (getArgs)
    -- для получения аргументов командной строки

import Data.List (isInfixOf)
    -- для проверки, входит ли подстрока в строку
```

Пример: аналог утилиты grep

```
main :: IO ()
main = getArgs >>= getWhat >>= \substr ->
    getContents >>= grep substr >>= putStr
```

В главной функции:

- Получаем аргументы командной строки,
- Из них первый — подстрока, которую надо искать, к которой прменяем функцию, в которой:
  - читаем `stdin` до признака конца файла, откуда
  - выбираем строки, содержащие заданную подстроку,
  - выводим результат в `stdout`.

Пример: аналог утилиты grep

- Если аргументов командной строки нет, то: строка для поиска — пустая,
- Иначе: строка для поиска — 1-й аргумент командной строки.

```
getWhat :: [String] -> IO String
getWhat    []         = return ""
getWhat    (what:_)   = return what
```

Пример: аналог утилиты grep

Собственно grep:

```
--       Что искать
--       |          Где искать
--       |          |
grep :: String -> String -> IO String
grep substr = return . unlines . filter pred . lines
    where pred = isInfixOf substr
```

- Разбить вход на строки,
- Отфильтровать строки,
- Полученные строки соединить в одну через '\n',
- Упаковать в монаду.

Пример: аналог утилиты grep

Код целиком:

```
#!/usr/bin/runhaskell

import System.Environment (getArgs)
import Data.List (isInfixOf)

main :: IO ()
main = getArgs >>= getWhat >>= \substr ->
    getContents >>= grep substr >>= putStr

getWhat :: [String] -> IO String
getWhat []       = return ""
getWhat (what:_) = return what

grep :: String -> String -> IO String
grep substr = return . unlines . filter pred . lines
    where pred = isInfixOf substr
```

Трассировка

...без нарушения чистоты функции:

```
import Debug.Trace

factorial 0 = trace ("0  -- end"    ) 1
factorial n = trace (show n ++ " " ) $ n * factorial (n - 1)

> factorial 4
4
3
2
1
0  -- end
120
```

Непременно удалять трассировку из завершенной программы!

Полезное

- `error :: [Char] -> a`
- `f = undefined`
- `it`
- `:main` *args*

Типы данных — Data Types

Типы данных, определяемые программистом

- Синонимы типов
- Алгебраические типы
  - Рекурсивные типы
  - Записи
  - Перечисления
- Введение нового типа
- Монады

Синонимы типов — Type Synonyms

```
type AddressBook = [(Name, Address)]
type Name        =   String
type Address     =   String
type TableT a    = [[a]]
```

Так определены типы `String` и `FilePath`.

Алгебраические типы данных — Algebraic Data Types

Алгебраический тип данных — составной тип данных с конструктором типа.

Конструктор типа — функция, принимающая значения типов, из которых составлен алгебраический тип, и возвращающая значение алгебраического типа.

Конструктор может иметь одинаковое или разные имена с типом, может входить в состав определения более сложных типов.

Пример

```
data Figure = Square    Double
            | Rectangle Double Double
            | Circle    Double
            deriving (Show, Read)

area :: Figure           -> Double
area   (Square a)        =  a ^ 2
area   (Rectangle a b)   =  a * b
area   (Circle r)        =  pi * r ^ 2

> area $ Square 2
4.0
> area $ Rectangle 2 3
6.0
```

TODO

Example: Rational algebraic type from the Standrd Prelude.

Перечислимый тип

```
data Color = Red     --
           | Orange  -- Все конструкторы
           | Yellow  -- перечислимого типа
           | Green   -- должны
           | Blue    -- быть
           | Indigo  -- без аргументов
           | Violet  --
           deriving (Show, Read, Eq, Enum, Ord)

{-  Yellow < Green      == True
    Green  > Blue       == False
    [Orange .. Blue]    == [Orange, Yellow, Green, Blue]
    maxBound :: Color   == Violet
    minBound :: Color   == Red
-}
```

Рекурсивный тип

```
data Tree a = Empty                          -- Пустой узел
            | Node a (Tree a) (Tree a)       -- Поддерево
            deriving (Show, Read, Eq)

ins :: Ord a => a -> Tree a -> Tree a
ins x Empty          = Node x Empty Empty
ins x (Node a left right)
    | x == a         = Node x  left          right
    | x <  a         = Node a (ins x left)   right
    | otherwise      = Node a  left         (ins x right)

> ins 0 Empty
Node 0 Empty Empty
> foldr ins (Node 0 Empty Empty) [-2, -3, 8, 7]
Node 0 ( Node (-3) Empty (Node (-2) Empty Empty) )
       ( Node   7  Empty (Node   8  Empty Empty) )
```

Почему функция `foldl` здесь не применима?

Записи — Records

```
data Figure = Square    Double
            | Rectangle Double Double
            | Round     Double
            deriving (Show, Read)
```

Доступ к типу и его полям осуществляется путем сопоставления с образцом с @ (as-pattern):

```
printFigure fig@(Rectangle a b) =
    print fig >> print a >> print b
printFigure fig@(Round r) = print fig >> print r
```

```
> printFigure $ Rectangle 10 20
Rectangle 10.0 20.0
10.0
20.0
> printFigure $ Round 2
Round 2.0
2.0
```

Записи с именованными полями

```
data Sphere = Sphere
    { x      :: Double
    , y      :: Double
    , z      :: Double
    , radius :: Double
    } deriving (Show, Read)

sph = Sphere
    { x      = 0.0
    , y      = 0.0
    , z      = 0.0
    , radius = 1.0  -- Если имя r уже есть в этом модуле
    }
```

Записи с именованными полями

При масштабировании сферы нужно изменить только `r`, остальные значения полей сохраняются:

```
scale :: Sphere -> Double -> Sphere
scale s m = s { radius = radius s * m }
```

Масштабируем сферу:

```
> scale sph 2
Sphere {x = 0.0, y = 0.0, z = 0.0, radius = 2.0}
```

Типы функций доступа к полю и конструктора типа:

```
> :t radius
radius :: Sphere -> Double
> :t Sphere
Sphere :: Double -> Double -> Double -> Double -> Sphere
```

Новый тип

Определим тип натуральных чисел (с нулем) так, чтобы его можно было использовать так же, как и примитивные типы.

Натуральный тип будет основан на целом типе:

```
newtype Natural = MakeNatural Integer
```

Введем ограничения на преобразование типа `Natural` к `Integer` и обратно.

Новый тип

```haskell
-- Преобразование типа Natural

fromNatural :: Natural -> Integer
fromNatural (MakeNatural n)
    | n < 0       = error "Negative can't be natural"
    | otherwise   = n

-- Преобразование к типу Natural

toNatural :: Integer -> Natural
toNatural n
    | n < 0       = error "Negative can't be natural"
    | otherwise   = MakeNatural n
```

Новый тип

Объявим тип `Natural` воплощением по меньшей мере следующих классов:

- `Num` — для выполнения арифметических операций.
- `Eq` — для проверки на равенство.
- `Show` — для представления значений в виде строк.

Новый тип

Тип Natural как воплощение класса Num

```
instance Num Natural where
    fromInteger = toNatural
    x + y       = toNatural $ fromNatural x + fromNatural y
    x - y       | z < 0       = error "Non-natural subtraction"
                | otherwise   = toNatural z
                where z = fromNatural x - fromNatural y
    x * y       = toNatural $ fromNatural x * fromNatural y
    abs x       = x                    -- required, non-negative
    signum x    | x == 0      = 0      -- required
                | otherwise   = 1      -- required
```

Новый тип

Тип `Natural` как воплощение класса `Eq`

```
instance Eq Natural where
    x == y  = fromNatural x == fromNatural y
    x /= y  = fromNatural x /= fromNatural y
```

Новый тип

Тип `Natural` как воплощение класса `Show`

```
instance Show Natural where
    show n = show $ fromNatural n
```

Новый тип

```
> MakeNatural (-1)
*** Exception: Negative number can't be natural
> MakeNatural 1
1
```

Это работает!

Новый тип

Пример использования типа Natural

```
factorial :: Natural -> Natural
factorial 0 = 1
factorial n = n * factorial (n-1)

> factorial 0
1
> factorial 1
1
> factorial 2
2
> factorial 3
6
> factorial 4
24
> factorial (-4)
*** Exception: Non-natural subtraction
```

Новый тип

Чтобы можно было использовать генераторы, тип должен быть воплощением класса `Enum`.

Тип `Natural` — воплощение класса `Enum`

```
instance Enum Natural where
    -- succ x = x + 1  -- not obligate
    -- pred x = x - 1  -- not obligate
    fromEnum = fromIntegral . fromNatural
        -- may cause overflow as...
    toEnum   = toNatural . fromIntegral
        -- ...uses Int instead of Integer!

-- Т.е. нужно описать, как получить
-- порядковый номер значения в перечислении
-- и значение по порядковому номеру
```

Новый тип

Теперь можно определить вычисление факториала с помощью генератора и свертки:

```
factorial' :: Natural -> Natural
factorial' n = product [1 .. n]

> factorial' 4
24
```

Новый тип

Итоги

(—) Многословно.

(+) Определены правила работы с типом, которые будут проверяться компилятором.

Далее

- Функтор
- Аппликативный функтор
- Моноид
- Монада

Зачем?

Для гибкости, но не в ущерб формальной корректности.

Традиционный пример: Maybe

В Prelude определен алгебраический тип:

```
data  Maybe a  =  Nothing | Just a
```

Пример

Этот тип можно использовать, например, так:

```
a `safeDiv` b
    | b == 0    = Nothing
    | otherwise = Just $ a `div` b

> 10 `safeDiv` 2
Just 5
> 10 `safeDiv` 0
Nothing
```

Как использовать результат в дальнейших вычислениях?

Функция fmap

```
> fmap (+1) (Just 2)
Just 3

> fmap (+1) Nothing
Nothing

> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f b
```

Класс Functor

Класс Functor объединяет типы, для которых существует отображение (a -> b) множества значений типа a на множества значений типа b (f — конструктор типа). Определение в стандартной библиотеке Haskell Platform:

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

Воплощения класса Functor

Функция должна быть определена для воплощений класса `Functor`. Для типа `Maybe` в стандартной библиотеке Haskell Platform имеем:

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

Отсюда:

```
fmap (+1) (Just 2)  ==  Just 3
fmap (+1) Nothing   ==  Nothing
```

Воплощения класса `Functor`

Воплощениями класса также являются типы `IO` и `[]`. В стандартной библиотеке Haskell Platform имеем:

```
> :i IO
newtype IO a
instance Functor IO
...

> :i []
data [] a = [] | a : [a]
instance Functor []
...
```

Воплощения класса Functor

```
instance Functor [] where
    fmap = map
    {- ... -}
```

Отсюда:

```
fmap (+1) [1,2,3] == map (+1) [1,2,3] == [2,3,4]
```

Воплощения класса Functor

```
instance Functor IO where
    fmap f action = do
        result <- action
        return (f result)
```

Действительно, в IO инкапсулируется код, используемый ради своих побочных эффектов, т.е. *действия* (а не *вычисления*). Применение fmap позволит применить чистую функцию к значению в монаде, например:

```
> :m + Data.List
> :m + System.Environment
> fmap (isSuffixOf "bash") $ getEnv "SHELL"
True
...
fmap (isSuffixOf "bash") $ getEnv "SHELL" :: IO Bool
```

Воплощения класса `Functor`

Воплощением класса `Functor` может быть любой тип вида (kind) `* -> *` с конструктором типа с одним параметром типа.

Законы класса `Functor`

```
fmap id        ==  id
fmap (f . g)   ==  fmap f . fmap g
```

Этим законам удовлетворяют воплощения класса `Functor` — типы `Maybe`, `IO` и `[]`.

Примеры

```
-- fmap id  ==  id

fmap id [1,2,3]   ==  id [1,2,3]   ==  [1,2,3]
fmap id (Just 1)  ==  id (Just 1)  ==  Just 1

-- fmap (f . g)  ==  fmap f . fmap g

fmap (abs . (+1))       [-1, -2, -3]  ==
(fmap abs . fmap (+1)) [-1, -2, -3]  ==
[0, 1, 2]

fmap (abs . (+1)) (Just (-5))       ==
(fmap abs . fmap (+1)) (Just (-5)) ==
Just 4
```

Инфиксная запись

В модулях `Data.Functor` и `Control.Applicative` имеется определение:

```
infixl 4 <$>

(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

Т.е. вместо

```
fmap (+1) (Just 1)
```

можно записать:

```
(+1) <$> Just 1 -- (<$>) -- apply
```

Замечание

В стандартной библиотеке имеются определения:

```
data (->) a b

instance Functor ((->) r) where  -- частичное применение ф-ции к
    fmap f g = f . g
```

Отсюда, **функции являются функторами**, а применение `fmap` к двум функциям эквивалентно композиции этих функций.

Аппликативный функтор

Аппликативный функтор расширяет понятие функтора на функции многих аргументов и функции, упакованные в контекст (тип с конструктором типа).

Примеры функций, упакованной в контекст:

```
(Just (+1)) :: Num a => Maybe (a -> a)
(Just (++)) :: Maybe ([a] -> [a] -> [a])
```

Аппликативный функтор является чем-то средним между обычными функторами и монадами. Необходимые определения даны в модуле `Control.Applicative`.

См. также: *Аппликативное программирование*.

Аппликативный функтор

В модуле `Control.Applicative`:

```
class (Functor f) => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

- `pure` — чистое значение $\rightarrow$ значение в контексте.
- `<*>` — (map) последовательные вычисления и комбинация их результатов.

Воплощения: `[]`, `Maybe`, `IO`, `((->) a)` ...

Пример: Maybe как аппликативный функтор

В Control.Applicative имеем определение:

```
instance Applicative Maybe where
    pure = Just
    Nothing  <*> _           = Nothing
    (Just f) <*> something = fmap f something
```

Отсюда:

```
> pure (Just 1)
Just 1
> pure "abc" :: Maybe String
Just "abc"
> (Just (+1)) <*> (Just 1)
Just 2
```

Законы аппликативных функторов

Минимальное определение аппликативного функтора должно удовлетворять следующим законам:

```
-- Идентичность (identity):
pure id <*> v = v

-- Композиция (composition):
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)

-- Гомоморфизм (homomorphism):
-- (отображение, сохраняющее основные операции
-- и основные соотношения):
pure f <*> pure x = pure (f x)

-- Перестановка (interchange):
u <*> pure y = pure ($ y) <*> u
```

Применение

Построение цепочек последовательных вычислений. Например:

```
> pure (+) <*> Just 3 <*> Just 5
Just 8
> pure (+) <*> Just 3 <*> Nothing
Nothing
> pure (+) <*> Nothing <*> Just 5
Nothing
```

Применение к (IO type)

Сумма чисел, введенных с клавиатуры:

```
> pure sum <*> (pure (map read) <*> (pure words <*> getLine))
10 20 30 40 50
150
```

Или так:

```
> sum <$> map read <$> words <$> getLine
10 20 30 40 50
150
```

Применение к спискам

```
> (++) <$> ["m_", "g_"] <*> ["method", "var"]
["m_method","m_var","g_method","g_var"]

> [(+), (*)] <*> [1, 2] <*> [3, 4]
[4,5,5,6,3,4,6,8]
```

Порядок вычислений во 2-м примере:

```
[(+), (*)] <*> [1, 2] == [(1+), (2+), (1*), (2*)]

[(1+), (2+), (1*), (2*)] <*> [3, 4] ==
[(1+3), (2+3), (1+4), (2+4), (1*3), (1*4), (2*3), (2*4)] ==
[   4 ,    5 ,    5 ,    6 ,    3 ,    4 ,    6 ,     8]
```

Класс Alternative (модуль Control.Applicative)

```haskell
class Applicative f => Alternative f where
    -- | The identity of '<|>'
    empty :: f a
    -- | An associative binary operation
    (<|>) :: f a -> f a -> f a
    -- | One or more.
    some :: f a -> f [a]
    some v = some_v
      where
        many_v = some_v <|> pure []
        some_v = (:) <$> v <*> many_v
    -- | Zero or more.
    many :: f a -> f [a]
    many v = many_v
      where
        many_v = some_v <|> pure []
        some_v = (:) <$> v <*> many_v
```

Воплощения класса `Alternative`

Воплощениями класса `Alternative` являются `[]` и `Maybe`:

```
empty :: [Int]     == []
empty :: Maybe Int == Nothing

Nothing <|> Just 2 == Just 2
[]      <|> [1, 2] == [1,2]
```

<|> — альтернатива

`f <|> g` означает: если вычисление `f` не успешно (`Nothing` или `[]`), то вычислить `g`.

```
Just 1  <|> Just 2   ==  Just 1
Nothing <|> Just 2   ==  Just 2
Just 1  <|> Nothing  ==  Just 1
Nothing <|> Nothing  ==  Nothing

[] <|> [1]                      == [1]
[] <|> [1] <|> [2,3]            == [1,2,3]
[] <|> [1] <|> [2,3] <|> [4] == [1,2,3,4]
```

Применение

Часто альтернативные функторы применяются в монадических парсерах в качестве комбинаторов. Там они позволяют записывать функции разбора в нотации, близкой к РБНФ.

TODO: пример парсера.

Иллюстрация: разбор программы на языке Scheme

Исходный текст на языке Scheme:

```
; test.scm

(define (sum list-of-values) (apply + list-of-values))
(define (average list-of-values)
    (/ (sum list-of-values) (length list-of-values)))
```

Программа на Haskell:

```
import Text.ParserCombinators.Parsec

-- | Структура данных для промежуточного представления
-- Программа ::= {Токен}.
-- Токен ::= Список|Атом, {Список|Атом}.

data Token = Atom String | List [Token] deriving (Show)
```

Иллюстрация: разбор программы на языке Scheme

```
-- Программа ::= {Токен}.
-- Токен ::= Список|Атом, {Список|Атом}.

program :: Parser [Token]
program = do
    spaces
    ls <- many1 ( list <|> atom )
    spaces
    return ls
```

Иллюстрация: разбор программы на языке Scheme

Списки атомов:

```
-- Непустой список:
-- Список ::= '(', Список|Атом, {Список|Атом}, ')'.

list :: Parser Token
list = do
    spaces
    char '('
    ls <- many1 ( list <|> atom )
    char ')'
    spaces
    return $ List ls
```

Иллюстрация: разбор программы на языке Scheme

Атомы:

```
-- Атом ::= Литера, {Литера}.
-- Литера ::= !(пробел|'('|')').

atom :: Parser Token
atom = do
    spaces
    a <- many1 (noneOf " \n\r\t\"()")
    spaces
    return $ Atom a
```

Иллюстрация: разбор программы на языке Scheme

Применение:

```
test = do
    src <- readFile "test.scm"
    case (parse program "" src) of
        Left  err -> print err
        Right ws  -> print ws
```

Иллюстрация: разбор программы на языке Scheme

Результат разбора:

```
[ List [ Atom "define"
       , List [ Atom "sum"
              , Atom "list-of-values"
              ]
       , List [ Atom "apply"
              , Atom "+"
              , Atom "list-of-values"
              ]
       ]
...
]
```

Парсер `Parsec`

http://legacy.cs.uu.nl/daan/parsec.html

Моноид

Моноид — множество, на котором определены:

- ассоциативная бинарная операция, которая паре элементов множества ставит в соответствие новый элемент этого же множества,
- нейтральный элемент (единица этой операции).

Например, множество списков:

- Бинарная ассоциативная операция — конкатенация списков,
- Нейтральный элемент — пустой список.

Примеры моноидов

Моноид, операция, нейтральный элемент:

- Числа, $+$, 0.
- Числа, $\times$, 1.
- Числа, min, $+\infty$.
- Числа, max, $-\infty$.
- Целые числа, НОК, 1.
- Булевы значения, И, «истина».
- Булевы значения, ИЛИ, «ложь».
- Списки, конкатенация, пустой список.
- Строки, конкатенация, пустая строка.
- Множества, объединение множеств, пустое множество.
- Словари, их объединение, пустой словарь.
- ...

Класс Monoid

В модуле `Data.Monoid` имеется определение класса `Monoid`.
Минимальное полное определение должно включать методы:

```
mempty  :: a             -- Нейтральный элемент
mappend :: a -> a -> a   -- Ассоциативная операция
```

Дополнительно, могут быть определен метод:

```
mconcat :: [a] -> a
```

Который по умолчанию определен так:

```
mconcat = foldr mappend mempty
```

Например:

```
mconcat [[1,2],[3,4],[5,6]]
[1,2,3,4,5,6]
```

Воплощения класса `Monoid`

Тип, операция, нейтральный элемент:

- `[a]`, `(++)`, `[]`.
- `()`, `(\_ _ -> ())`, `()`.
- ...

См.: `Data.Monoid`

Монада

Класс определен в модулях `Prelude` и `Control.Monad`.

```
class Monad m where  -- m -- конструктор типа
    {- ... -}
```

Минимальное определение воплощение класса должно содержать определения методов:

```
(>>=)  :: m a -> (a -> m b) -> m b  -- bind
return :: a -> m a
```

Дополнительно, могут быть определены:

```
(>>) :: forall a b. m a -> m b -> m b  -- then
fail :: String -> m a  -- выполняется при ошибке сопоставления
                       -- с образцом внутри конструкции do
```

`forall a. a -> a` $\equiv \forall\alpha.\alpha \to \alpha$ — любой подходящий тип.

Законы монад

Left identity:

```
return a >>= f  ==  f a
```

Right identity:

```
m >>= return  ==  m
```

Associativity:

```
(m >>= f) >>= g  ==  m >>= (\x -> f x >>= g)
```

MonadPlus

В модуле `Control.Monad` определен класс, сочетающий свойства монады и моноида:

```
class (Monad m) => MonadPlus m where
   mzero :: m a                    -- Нейтральный элемент
   mplus :: m a -> m a -> m a      -- Ассоц. операция...
```

Воплощениями этого класса являются `[]` и `Maybe`. Например, для списка:

```
instance MonadPlus [] where
  mzero = []
  mplus = (++)
```

Важнейшие монады

Монада(ы)	**Семантика**
`Maybe`	Исключение
`Either`	Исключение (с диагностикой)
`Error`	Ошибка выполнения (с диагностикой)
`State`, `ST`	Вычисления с состояниями
`IO`	Ввод/вывод
`Writer`	Запись логов
`[]`	Индетерминизм?

TODO

- Императивное программирование на Haskell
- Неизменяемые массивы
- Изменяемые массивы
- Пример: сортировка пузырьком
  - В монаде `IO`
  - В монаде `ST`
- Пример: псевдослучайные числа
- Do notation considered harmful
- Монадические функции (`Control.Monad`)
- Трансформеры монад
- Композиция монад (монады Клейсли)

TODO

- Предупреждение и обработка ошибок и исключений
- Foreign Function Interface (FFI)
- Программирование клиент-серверных приложений
- Событийно-ориентированное программирование
- Программирование с помощью стрелок
- Реактивное программирование