ChatGPT

Методичка по языку Haskell

1. Основы синтаксиса: переменные, функции, типы данных

Haskell – **чисто функциональный, статически типизированный** язык. Это значит, что тип каждого выражения известен на этапе компиляции, и все выражения в коде не имеют побочных эффектов (функции всегда возвращают результат и не изменяют внешнее состояние) [1]. Haskell обладает **выводом типов** – компилятор сам может определить типы большинства выражений, поэтому часто нет необходимости явно указывать тип переменной или функции [2]. Тем не менее, рекомендуется прописывать **аннотации типов** для функций (особенно не тривиальных) – это улучшает читаемость и помогает отлавливать ошибки.

Переменные в Haskell по сути являются именами для значений. После определения изменить значение переменной нельзя – в языке *нет оператора присваивания*, только связывание имени с выражением [3]. Например, определение `x = 5` задаёт имя `x` со значением `5`. Переменные и функции обозначаются **с маленькой буквы**, а имена типов и конструкторов данных – с Большой [4].

Определение функций напоминает математические выражения. Используется синтаксис `имя параметры = выражение`. Например, функция сложения двух чисел:

```
add :: Int -> Int -> Int   -- Аннотация типа (не обязательна, но полезна)
add x y = x + y            -- Функция add возвращает сумму x и y
```

Здесь `add` принимает два параметра типа `Int` и возвращает `Int`. Объявление типа `Int -> Int -> Int` читается как "`add` принимает `Int`, потом еще `Int`, и возвращает `Int`" [5]. Возвращаемый тип – последний в цепочке через `->`, а все предыдущие – типы параметров. Вызов функции происходит просто передачей аргументов через пробел: например, `add 2 3` вернёт `5`. Обратите внимание: в определении функции **не используются скобки** вокруг параметров и не требуется ключевое слово `return` для возврата – функция в Haskell *всегда возвращает последнее вычисленное выражение* автоматически.

Базовые типы данных в Haskell включают: - `Int` – целое число фиксированного диапазона (обычно 32-битное, максимум около 2 млрд) [6]. - `Integer` – целое произвольной длины (не имеет фиксированного предела, может быть сколь угодно большим) [7]. - `Float` – число с плавающей запятой одинарной точности (обычно 32 бита). - `Double` – число с плавающей запятой двойной точности (64 бита, более точный аналог Float). - `Bool` – логический тип, значения `True` или `False`. - `Char` – символ в одиночных кавычках, например `'a'`. Строки в Haskell – это списки символов типа `[Char]` (есть синоним `String`).

Пример: определим пару переменных разных типов и выведем их типы в интерактивной консоли GHCi:

```haskell
a = True                -- Bool
b = "HELLO!"            -- String, то есть [Char]
c = (False, 'X')        -- кортеж (Bool, Char)

-- В GHCi:
-- > :t a        (вывод: a :: Bool)
-- > :t b        (вывод: b :: [Char])
-- > :t c        (вывод: c :: (Bool, Char))
```

Здесь `:t` – команда ghci, показывающая тип выражения. Мы видим, например, что `"HELLO!"` имеет тип `[Char]` – то есть список символов, а кортеж `(False, 'X')` имеет тип `(Bool, Char)`.

Haskell – **неизменяемый** (immutable) язык: если мы определили `x = 10`, то `x` всегда будет равен 10 в этой области видимости. Вместо изменения переменных код пишется как набор **выражений** и **функций**, где новые значения получаются из старых, но не изменяют их. Например, можно определить новое значение на основе старого:

```haskell
x = 10
y = x + 5    -- y будет 15, при этом x всё так же равен 10
```

Подобная модель упрощает понимание программы: значения не “меняются под ногами”, а каждый именованный результат раз и навсегда привязан к своему выражению.

2. Каррирование и частичное применение

Одной из фундаментальных концепций Haskell является **каррирование**. В Haskell функция *не может* официально принимать более одного аргумента – если мы пишем функцию с “несколькими” параметрами, на самом деле эта функция возвращает новую функцию для каждого аргумента кроме последнего [8] [9]. Например, объявление:

```haskell
f :: Int -> Double -> Char -> Bool
```

означает, что `f` – функция, которая принимает `Int` и возвращает *функцию*, которая принимает `Double` и возвращает *функцию*, которая принимает `Char` и в итоге возвращает `Bool` [9]. Этот процесс преобразования функции многих аргументов в набор вложенных функций называется **каррированием**. Haskell автоматически каррирует все функции с более чем одним параметром [10].

Благодаря каррированию становится возможным **частичное применение функций**. Частичное применение – это вызов функции не со всеми аргументами, а только с частью, в результате чего получается новая функция, ожидающая остальные аргументы. По сути, в Haskell любое применение функции по умолчанию может быть частичным. Например:

```haskell
add :: Int -> Int -> Int
add x y = x + y

addOne :: Int -> Int        -- новая функция, добавляющая 1 к аргументу
addOne = add 1

res = addOne 5              -- res будет 6 (т.е. 1 + 5)
```

Здесь `add` – функция от двух аргументов. Мы создали `addOne` как частично применённую `add` с первым аргументом 1. Теперь `addOne` – это функция одного аргумента, которая прибавляет 1 к своему параметру. Вызов `addOne 5` вернёт 6. Мы могли бы также напрямую использовать частичное применение при вызове: выражение `add 1 5` эквивалентно вышеописанному.

Еще пример частичного применения – **секции** (sections): это синтаксический сахар для частичного применения инфиксных операторов. Например, выражение `(2/)` – это функция, которая делит 2 на свой аргумент, а выражение `(/2)` – функция, делящая свой аргумент на 2. Пример:

```haskell
halve = (/ 2)     -- делит число на 2
res1 = halve 10   -- 5.0 (результат типа Fractional)

subtractFrom2 = (2 -)  -- вычитает из 2 переданный аргумент
res2 = subtractFrom2 5  -- -3
```

Важно отметить, что термин *“частичное применение”* часто используется для удобства, хотя технически в Haskell любая передача не всех аргументов функции – это просто **обычное применение**, возвращающее функцию. С точки зрения языка, мы всегда имеем дело с функциями одного аргумента (каррирование), поэтому выражение `f 5` для `f :: Int -> Int -> Int` можно думать как «частично применённая функция» для ясности, хоть это и не совсем формально [11].

3. Рекурсия и хвостовая рекурсия

Поскольку в Haskell отсутствуют традиционные императивные циклы (`for`, `while` и т.д.), повторение операций достигается с помощью **рекурсии** – вызова функцией самой себя [12] [13]. Рекурсия – естественный механизм для функционального языка. Простейший пример – вычисление факториала:

```haskell
factorial :: Integer -> Integer
factorial 0 = 1                  -- Базовый случай: факториал 0 равен 1
factorial n = n * factorial (n-1) -- Рекурсивный случай
```

Здесь функция `factorial` вызывает сама себя, уменьшая аргумент `n` на 1 на каждом шаге. Обязательно должен быть **базовый случай** (в данном случае `0`), на котором рекурсия останавливается, иначе она была бы бесконечной.

Рекурсия может привести к накоплению вызовов в памяти (каждый незавершённый рекурсивный вызов хранит контекст). Однако, если рекурсивный вызов является *последней* операцией в функции, такую рекурсию называют **хвостовой (tail recursion)**. Хвостовая рекурсия может быть автоматически оптимизирована компилятором до эффективного цикла, не потребляющего дополнительный стек [14] [15]. Идея в том, что результат рекурсивного вызова сразу же возвращается, не требуя сохранения контекста.

Рефакторинг факториала в хвостово-рекурсивный стиль с использованием аккумулятора:

```
factorialTR :: Integer -> Integer
factorialTR n = go n 1
  where
    go 0 acc = acc              -- когда n достиг 0, возвращаем накопленный
результат
    go k acc = go (k - 1) (acc * k)  -- рекурсивный вызов в хвостовой позиции
```

Здесь дополнительный параметр `acc` накапливает результат (аккумулятор). Вызов `factorialTR 5` вычисляется как `go 5 1`, затем `go 4 5`, `go 3 20`, ..., `go 0 120` и возвращает `120`. Последний рекурсивный вызов `go` просто передаёт результат дальше без дополнительных операций – это хвостовая рекурсия. Такой код компилятор оптимизирует в простой цикл без разрастания стека вызовов [14].

Вывод: любую прямую рекурсию в Haskell при желании можно переписать как хвостовую, добавив необходимые параметры-аккумуляторы. Если это не сильно ухудшает читабельность, стоит так делать – **GHC оптимизирует хвостовую рекурсию** до эффективного кода, эквивалентного циклу [15]. Тем не менее, благодаря ленивости и оптимизациям, даже не хвостовая рекурсия в Haskell может работать хорошо, но помнить о хвостовой рекурсии полезно для пишущего эффективный код.

4. Алгебраические типы данных

Haskell позволяет программисту определять свои **алгебраические типы данных (ADT)** с помощью ключевого слова `data`. Алгебраические типы включают в себя *суммарные типы* (перечисления альтернатив) и *произведённые типы* (структуры с несколькими полями), а также их комбинации.

Простейший пример ADT – перечисление (суммарный тип), где мы перечисляем возможные значения типа. Например, определим тип цвета:

```
data Color = Red | Green | Blue
```

Здесь `Color` – новое имя типа, а `Red`, `Green`, `Blue` – **конструкторы** этого типа (начинаются с заглавной буквы) [4]. Теперь переменная типа `Color` может принимать одно из трёх значений. Подобно этому, можно определить тип дня недели, карточной масти и т.д.

Конструкторы могут иметь параметры, что позволяет создавать более сложные типы. Например:

```haskell
data Shape = Circle Float
           | Rectangle Float Float
```

Тип `Shape` имеет два варианта (конструктора): `Circle` с параметром радиуса типа `Float` и `Rectangle` с двумя параметрами (ширина и высота). Таким образом, `Shape` – тоже суммарный тип: значение либо "круг с радиусом r", либо "прямоугольник с размерами w и h".

Алгебраические типы могут быть *параметрическими*, т.е. иметь параметры типа (аналог обобщённых типов или генериков). Например, стандартный список в Haskell в примитивном виде можно описать так:

```haskell
data List a = Nil | Cons a (List a)
```

Здесь `a` – параметр типа (произвольный тип элемента). Конструктор `Nil` представляет пустой список, а `Cons` – непустой список, содержащий элемент типа `a` (голова списка) и хвост – оставшийся список типа `List a` [16]. По сути, мы описали, что список либо пуст, либо состоит из элемента и ещё одного списка. Это и есть рекурсивное определение списка.

Haskell также поддерживает **синтаксис записей (Record Syntax)** для удобного именования полей в конструкторах. Например, можно определить `Rectangle` с именованными полями:

```haskell
data Rectangle = Rectangle { width :: Float, height :: Float }
```

Это определяет конструктор `Rectangle` с двумя полями, а также автоматически создаёт функции доступа `width` и `height`, которые принимают `Rectangle` и возвращают соответствующее поле [17]. Например, можно написать `width (Rectangle 5.0 3.0)` и получить `5.0`.

5. Сопоставление с образцом (Pattern Matching)

Сопоставление с образцом – мощный механизм Haskell для разбора данных по структуре. Он позволяет выразительно проверять форму значений алгебраических типов (и не только) и разбирать их на компоненты.

Функции в Haskell часто определяются *несколькими уравнениями* с разными образцами параметров. При вызове функции пытаются по порядку подобрать подходящий образец и выполняют соответствующее тело функции [18]. Рассмотрим на примере списка (как определено выше):

```haskell
len :: List a -> Int
len Nil = 0                    -- если список пустой, длина 0
len (Cons _ xs) = 1 + len xs   -- если список непустой: игнорируем голову (_)
и рекурсивно считаем хвост
```

При вызове `len`, сначала проверяется первый шаблон. Если аргумент совпадает с `Nil`, возвращается 0. Если нет, пытается второй шаблон: значение должно совпасть с `Cons _ xs` – то есть быть непустым списком. Символ `_` здесь означает "любое значение, которое нам не важно" [19]. Таким образом, для любого `Cons` конструктора, функция вернёт `1 + len xs` (один плюс длина хвоста). Порядок определения образцов важен: они проверяются сверху вниз.

Аналогично для стандартного спискового типа `[]` / `(x:xs)` можно определить длину:

```
length' :: [a] -> Int
length' []     = 0
length' (_:xs) = 1 + length' xs
```

Здесь `[]` – пустой список, а шаблон `(_:xs)` разбивает список на головной элемент (который обозначен `_` и не используется) и хвост `xs`.

Сопоставление с образцом можно использовать и внутри функции через выражение `case`. Например, ту же функцию `length'` можно было написать так:

```
length' lst = case lst of
                []   -> 0
                _:xs -> 1 + length' xs
```

Однако чаще встречается форма с определениями через `=` для разных образцов, как в предыдущем примере – она более декларативна.

При сопоставлении можно извлекать значения из конструкций. Например, определим функцию, вычисляющую площадь нашего типа `Shape`:

```
area :: Shape -> Float
area (Circle r)      = pi * r^2                    -- извлекаем радиус r из
конструктора Circle
area (Rectangle w h) = w * h                       -- извлекаем ширину w и
высоту h
```

При вызове, скажем, `area (Circle 10)`, будет использовано второе уравнение, и подставив `r=10` вычислится `pi * 100`. А для `area (Rectangle 5 7)` сработает третья строка и вернёт `35`.

Образцы могут быть вложенными и сложными. Можно сопоставлять сразу несколько уровней конструкции. Например, для списка можно сразу расписать шаблон из нескольких элементов: `[x, y, z]` в определении функции раскроется как `x: y: z: []`. Также есть синтаксическое удобство – **ас-образцы**: запись вида `v@pattern` позволяет назвать всю структуру целиком,

одновременно разобрав её по шаблону. Например: `list@(x:xs)` внутри функции даст имя `list` всему списку, а также `x` его голове и `xs` хвосту.

Важно следить, чтобы паттерны покрывали все возможные варианты. Если ни один образец не подойдет для переданного значения, будет ошибка времени выполнения (неуспешное сопоставление с образцом) [20]. Например, если бы мы определили `area` только для `Circle` и не указали вариант для `Rectangle`, вызов `area (Rectangle 1 2)` привёл бы к исключению. Поэтому компилятор обычно предупреждает о неполных паттерн-матчингах. Для некоторых стандартных функций, вроде `head :: [a] -> a` (возвращает первый элемент списка), невызова для пустого списка приводит к исключению – лучше избегать таких *частичных функций* или обрабатывать случай пустого списка отдельно (см. раздел об обработке ошибок).

6. Модули

Haskell-программы организованы в **модули**. Каждый файл `.hs` является модулем, и имя модуля должно совпадать с именем файла (с заглавной буквы) [21]. Например, код в файле `MyProject/Utils.hs` должен начинаться с объявления модуля:

```
module MyProject.Utils where
```

Это объявление в начале файла задаёт имя модуля и сообщает компилятору, что далее идут определения, принадлежащие этому модулю. По умолчанию, без явного списка экспортов, модуль экспортирует все определенные в нём функции и типы. Можно явно указать, что экспортировать, перечислив имена в круглых скобках после имени модуля.

Модули помогают организовать код и избежать конфликтов имён. Чтобы использовать функции или типы из другого модуля, используется **импорт**. Импорт пишется в начале файла (после объявления модуля) с ключевым словом `import`. Примеры импорта:

```
import Data.List               -- импортировать всё из Data.List
import Data.Char (toUpper, toLower)  -- импортировать только указанные функции
из Data.Char
import qualified Data.Map as M      -- импортировать Data.Map с префиксом M
(квалифицированный импорт)
```

В первом случае мы получаем в область видимости все функции, экспортируемые модулем `Data.List`. Во втором – только `toUpper` и `toLower` из `Data.Char` [22]. Третий пример иллюстрирует **квалифицированный импорт**: все имена из `Data.Map` нужно будет использовать с префиксом `M.` – например, `M.insert` – это удобно, если разные модули имеют одноимённые функции, или чтобы явно видеть, откуда та или иная функция.

По соглашению, главный модуль программы называется `Main` и должен экспортировать функцию `main`. Когда вы компилируете программу в исполняемый файл, именно `Main.main` будет точкой входа.

Простая структура проекта может быть такой: - `Main.hs`:

```
module Main where
import MyProject.Utils  -- импорт собственного модуля
main :: IO ()
main = putStrLn (myMessage "Haskell")
```

- `MyProject/Utils.hs`:

```
module MyProject.Utils (myMessage) where
myMessage :: String -> String
myMessage name = "Hello, " ++ name ++ "!"
```

Здесь мы создали модуль `MyProject.Utils` с функцией `myMessage`, экспортировали её, и импортировали в `Main`. Функция `main` вызывает `myMessage "Haskell"` и печатает результат.

7. Ленивые вычисления

Одной из «визитных карточек» Haskell являются **ленивые вычисления** (lazy evaluation). По умолчанию Haskell не вычисляет выражения сразу, а откладывает вычисление до тех пор, пока результат не понадобится [23]. В традиционных (строгих) языках сначала вычисляются все аргументы функции, затем сама функция; в Haskell же аргументы могут вовсе не вычисляться, если в них нет необходимости.

Простой пример: возьмём функцию `max` и два выражения в аргументах – `(5+3)` и `(4*4)`:

- В строгом языке (например, Python или Java) сначала будут вычислены оба аргумента: `5+3` и `4*4` (получим 8 и 16), и лишь затем `max 8 16` вернёт 16.
- В Haskell же выполнение пойдёт по-другому: `max (5+3) (4*4)` сперва создаёт “обещания” (thunks) для каждого аргумента, не вычисляя их [23]. Затем `max` сравнивает первый и второй аргументы. Чтобы выполнить сравнение, ему *приходится* вычислить аргументы – но только до той степени, которая необходима для сравнения. В итоге `5+3` вычисляется (даёт 8), затем `4*4` (даёт 16), и `max` возвращает 16. Вычисление каждого подвыражения отложилось до момента, когда без него было не обойтись.

Благодаря ленивости в Haskell возможны **бесконечные структуры данных**. Например, можно определить бесконечный список всех положительных чисел: `nums = [1..]`. Этот список сам по себе никогда полностью не вычислится (он бесконечен), но вы можете брать из него сколько нужно элементов. Например, `take 5 nums` вернёт `[1,2,3,4,5]`, а `sum (take 100 nums)` посчитает сумму первых 100 чис. Ленивость обеспечивает, что вычислятся только те части бесконечного списка, которые реально востребованы программой.

Однако у ленивости есть и минусы – например, сложно предсказать использование памяти, и иногда накапливаются "ленивые" невычисленные выражения (thunks), что может приводить к утечкам

памяти. Для управления этим в Haskell есть средства принудительного вычисления (**строгие вычисления**): например, функция `seq` или специальный оператор строгого применения `$!`. Также многие структуры данных имеют строгие версии полей (например, `Data.Map` хранит значения строго). Но для начала достаточно понимать общий принцип: *вычисляется как можно позже и только то, что действительно нужно* [24].

Важно: ленивость означает, что порядок вычислений может быть неочевидным. Тем не менее, в чисто функциональном коде (без побочных эффектов) это никак не влияет на результат – только на производительность. А вот в коде с побочными эффектами (например, внутри `IO`) Haskell **гарантирует** порядок выполнения, используя монадическую обёртку IO (подробнее в разделе про монады).

8. Списки и функции высшего порядка: map, filter, fold

Списки – одна из основных структур данных в Haskell. Это однородные последовательности элементов. В квадратных скобках можно задавать списки явно, например: `[1,2,3]` – список из трёх чисел. Синтаксис `[a..b]` создаёт список арифметической прогрессии от `a` до `b`. Например, `[1..5]` соответствует `[1,2,3,4,5]`. Можно задать шаг: `[1,3..9]` есть `[1,3,5,7,9]`. И как упоминалось, открытый диапазон `[1..]` задаёт бесконечный список всех целых начиная с 1.

Работа со списками в Haskell часто осуществляется с помощью **функций высшего порядка** – функций, принимающих другие функции в качестве аргументов. Наиболее распространённые: `map`, `filter` и семейство `fold` (свёртки).

- `map` применяет заданную функцию к каждому элементу списка, возвращая список результатов.
- `filter` отбирает из списка элементы, удовлетворяющие условию (предикату).
- `foldl` / `foldr` – сворачивают список в одно значение, применяя *аккумуляторную* функцию шаг за шагом (левая или правая свёртка).

Примеры использования:

```
nums    = [1..5]                        -- [1,2,3,4,5]
doubled = map (*2) nums                 -- удвоим каждый элемент: [2,4,6,8,10]
evens   = filter even nums              -- отфильтруем чётные: [2,4]
sumNums = foldl (+) 0 nums              -- сумма всех элементов: 15
prodNums = foldr (*) 1 nums             -- произведение всех элементов: 120
```

В примере `map (*2) nums` применяет лямбда-функцию умножения на 2 ко всем элементам списка `nums`. Результат – новый список `[2,4,6,8,10]`, исходный `nums` не изменяется (в функциональном языке структуры *не модифицируются*, а создаются новые). Функция `filter even nums` оставляет только те элементы `nums`, для которых предикат `even` (чётность) возвращает True – остаются `[2,4]`.

Свёртки (`foldl`, `foldr`) берут бинарную функцию и начальное значение аккумулятора, после чего проходят по списку, постепенно накапливая результат. В случае `foldl (+) 0 nums` происходит вычисление `(((0 + 1) + 2) + 3) + ... + 5`, возвращая 15. `foldr (*) 1 nums` вычисляет `1 * 1 * 2 * 3 * 4 * 5` рекурсивно (начиная справа), возвращая 120.

Особенность: `foldl` (левая свёртка) ассоциирует операцию влево, а `foldr` – вправо. Для ассоциативных операций (+, *, && и т.п.) это не важно для результата, но может быть важно для производительности и для бесконечных списков. Например, `foldr` способен работать на бесконечных списках в случаях, когда функция может лениво игнорировать часть входа (например, `foldr (||) False` – возвращает True, как только встречает True в списке, даже если список бесконечен). `foldl` же на бесконечном списке не завершится, потому что должен дойти до конца. Также `foldl` в нестрогом языке может накапливать большие невычисленные выражения; часто используют строгий вариант `foldl'` (из `Data.List`), который вычисляет аккумулятор по ходу. Но эти детали оптимизации выходят за рамки нашего уровня – в простых случаях можно использовать, что понятнее по смыслу.

Haskell предлагает и другие удобные функции для работы со списками: `take`, `drop`, `concat`, списковые генераторы (list comprehensions) и многое другое. Например, список comprehension: `[x^2 | x <- [1..5], even x]` означает “возьми `x` из списка 1..5, оставь только чётные (`even x`), и возведи их в квадрат” – результат будет `[4,16]`. Такие встроенные конструкты и функции высшего порядка заменяют необходимость явного написания циклов, делая код декларативным и лаконичным.

9. Монады: `Maybe` и `IO`

Монада – это особый тип (точнее, типовой класс `Monad`), экземпляры которого поддерживают определённый интерфейс (`>>=`, `return` и др.), позволяющий удобно связывать последовательности вычислений. Если не углубляться в теорию категорий, монаду можно представить как *конвейер вычислений* с контекстом.

Рассмотрим монаду `Maybe`. Этот тип определён как:

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

`Maybe` представляет "контекст возможного отсутствия значения". Значение типа `Maybe Int`, например, либо имеет вид `Just 5` (оборачивает настоящее значение 5), либо `Nothing` (означает отсутствие результата). Как монада, `Maybe` позволяет строить цепочки вычислений, где каждый шаг может вернуть `Nothing` (неудачу) и тогда вся цепочка прекратится, либо вернуть `Just x` и тогда x передастся в следующий шаг. Это удобно для обработки ошибок без исключений – *“чистый”* способ сигнализировать об ошибке, вместо бросания исключения [25].

Пример: допустим, у нас есть несколько функций, которые могут возвращать `Nothing` (например, парсинг числа, поиск элемента и т.п.). Если вызывать их последовательно, придётся после каждого шага проверять результат на `Nothing`. Монада `Maybe` вместе с синтаксическим сахаром **do-нотацией** облегчают эту задачу. Вместо громоздкого вложенного `case` можно писать так:

```haskell
safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing          -- деление на 0 не допускается
safeDiv x y = Just (x `div` y)

calc :: Int -> Int -> Maybe Int
calc a b = do
    x <- safeDiv a b      -- попытка вычислить a/b, x имеет тип Int внутри Just
    y <- safeDiv a (b-1)  -- попытка вычислить a/(b-1)
    return (x + y)
```

Здесь в блоке `do` каждая строчка вида `v <- ...` извлекает значение из контекста `Maybe` (если там `Just`). Если на любом шаге встретится `Nothing`, дальнейшие вычисления пропускаются и сразу возвращается `Nothing` – такая семантика определена для монадического оператора `>>=` (bind) у `Maybe` [26]. Наш пример `calc` возвращает `Just (a/(b) + a/(b-1))`, если оба деления успешны, или `Nothing`, если хотя бы одно оказалось невалидным (например, если `b` или `b-1` равны 0).

Монада `IO` отвечает за **операции ввода-вывода и побочные эффекты**. В Haskell любые функции, имеющие побочные эффекты, получают тип `IO a` – это значение, которое, будучи выполненным, производит эффект и возвращает результат типа `a`. Например, `getLine :: IO String` при выполнении читает строку от пользователя и выдаёт строковое значение, а `putStrLn :: String -> IO ()` при выполнении печатает строку на экран (возвращая пустой тип `()` в контексте IO).

Как и `Maybe`, монада `IO` поддерживает do-нотацию, позволяющую писать последовательность действий по шагам, как будто мы имеем императивный код – на самом деле этот код описывает *одно* совокупное действие `IO`. Пример – простейшая консольная программа:

```haskell
main :: IO ()
main = do
    putStrLn "Как тебя зовут?"
    name <- getLine
    putStrLn ("Привет, " ++ name ++ "!")
```

Здесь `name <- getLine` означает: выполнить действие `getLine` (читает ввод пользователя) и присвоить переменной `name` полученное "внутреннее" значение (тип `String`). Если попытаться использовать результат `getLine` вне контекста IO, ничего не выйдет – можно получить его только внутри очередного действия IO. Это принципиально: у типа `IO` *нет* конструкторов, и нельзя получить `String` из `IO String`, минуя механизмы, предоставленные монадоой [27]. Тем самым Haskell гарантирует, что вы не сможете спрятать побочный эффект внутри чистой функции: любое использование `IO` явно отражается в типе функции. В нашем примере `main` имеет тип `IO ()`, что и требуется для точки входа в программу (по соглашению `main` в Haskell – это действие в `IO`, которое выполняется при запуске программы).

Итак, `Maybe` и `IO` – два примера монад: - `Maybe` – монада, инкапсулирующая возможность отсутствия результата (или возникновения ошибки). Позволяет писать последовательность вычислений, автоматически прекращающуюся при первой неудаче. - `IO` – монада для эффектов: чтения/записи, работы с файлами, генерирования случайных чисел и т.д. Позволяет последовательно выполнить операции, сохраняя чистоту остальной части программы (чистые функции не могут вызвать `IO` без явного помечания в типе).

Обе они реализуют интерфейс `Monad` (и некоторые другие, например `Functor` и `Applicative`), поэтому для них доступна do-нотация и операторы вроде `>>` (последовательное выполнение без передачи значения) и `>>=` (явное последовательное связывание с функцией). **Важно:** понимание монад часто приходит не сразу – не бойтесь, если поначалу это кажется магией. Практически монады можно воспринимать как *контексты*, с которыми можно работать единообразно. `Maybe` – контекст "может не быть значения", `IO` – контекст "это действие с эффектом". Без углубления в теорию, этого достаточно, чтобы начать ими пользоваться.

10. Обработка ошибок

В языках вроде Python/JavaScript мы привыкли к механизмам исключений (try/except) для обработки ошибок. В Haskell тоже есть исключения, но они используются в основном для ошибок в `IO` или в ситуациях, когда ошибка не предотвращена на уровне типов. В идеологии Haskell предпочтительно по возможности **обрабатывать ошибки через типы**, а не через исключения.

В чистом коде (вне `IO`) типовые способы обработки ошибок – это уже знакомые нам типы: - `Maybe a` – сигнализирует о возможном отсутствии значения (ошибке). Например, безопасное преобразование строки в число: возвращает `Just n` или `Nothing`, вместо того чтобы бросать исключение при ошибке. - `Either e a` – расширение идеи Maybe: хранит либо результат `Right a`, либо информацию об ошибке `Left e` (где `e` может быть строкой сообщения или специальным типом ошибки). Например, парсер может возвращать `Either String Result`, где `Left "Parse error..."` содержит описание ошибки.

Используя `Maybe`/`Either`, мы *принуждаем* пользователя функции обработать случай ошибки, потому что он получит не “чистое” значение, а обёртку, которую надо разобрать (через сопоставление с образцом или монаду). Это ведёт к более надёжному коду: невозможно забыть про ошибку – тип заставит учесть все варианты.

Пример: напишем функцию безопасного извлечения первого элемента списка:

```
safeHead :: [a] -> Maybe a
safeHead []    = Nothing
safeHead (x:_) = Just x
```

Если список пуст, возвращается `Nothing`, иначе `Just x` (первый элемент). Пользуясь этой функцией, вы уже не сможете нечаянно получить исключение – тип `Maybe a` заставит вас рассмотреть оба случая. Например:

```haskell
case safeHead myList of
    Nothing -> putStrLn "Список пуст!"
    Just val -> putStrLn ("Первый элемент: " ++ show val)
```

В стандартной библиотеке есть много функций, возвращающих `Maybe` / `Either` для ситуаций, когда что-то может пойти не так (парсинг, поиск и пр.). Это считается хорошей практикой вместо бросания исключений. Исключения в чистом коде существуют (через функцию `error` или неопределённость `undefined`), но их использование не рекомендуется, кроме как для отладки или случаев, когда ошибка означает программную ошибку, которую нельзя обработать (например, паттерн-матчинг, не покрывший случай, приводит к `error` внутри).

В контексте IO (то есть для побочных эффектов) Haskell поддерживает исключения, подобные обычным языкам. Например, чтение из файла может бросить исключение, если файл не найден. Эти исключения обрабатываются с помощью функций в `Control.Exception` (например, `catch`, `try` и т.д.). Пример:

```haskell
import Control.Exception (try, IOException)

main = do
    result <- try (readFile "data.txt") :: IO (Either IOException String)
    case result of
      Left ex   -> putStrLn $ "Ошибка ввода-вывода: " ++ show ex
      Right contents -> putStrLn $ "Файл содержит " ++ show (length (lines
contents)) ++ " строк"
```

Здесь `try` ловит исключение чтения файла и возвращает `Either` – `Left ex` в случае ошибки или `Right contents` в случае успеха. Таким образом, даже в `IO` мы вписываем обработку ошибки в тип – работая с `Either`.

Подведём итог: **обработка ошибок в Haskell** обычно делается через типы (*«если что-то может пойти не так – отрази это в типе результата»*). Тип `Maybe` используется для простых случаев (ошибка без подробностей), `Either` – для случаев, когда нужно вернуть информацию об ошибке. Исключения всё же есть и иногда уместны (например, для непредвиденных ситуаций или в библиотечном коде), но их использование локально и управляется тоже через `IO`. Такой подход способствует написанию функций, которые либо явно учитывают возможность неуспеха, либо задекларированы как `IO` (а значит, потенциально небезопасные и требующие внимания). В итоге ваш код становится более надёжным, поскольку вы не пропустите обработку ошибки – компилятор поможет вам в этом.

11. Record Syntax и классы типов (введение)

Record Syntax (записи). Как упоминалось в разделе про ADT, Haskell позволяет задавать **имена полей** в конструкторах данных. Это нечто похожее на struct или объекты с именованными полями в других языках. Использование record syntax упрощает доступ к полям и делает код понятнее.

Пример определения записи:

```
data Person = Person { name :: String, age :: Int }
```

Здесь мы определили тип `Person` с конструктором `Person`, имеющим два поля: `name` и `age`. Автоматически созданы функции `name :: Person -> String` и `age :: Person -> Int` для доступа к полям [17]. Теперь можно создать значение: `p = Person { name = "Alice", age = 30 }`. Обращение `name p` вернёт `"Alice"`, а `age p` – `30`.

Поля записей можно использовать при сопоставлении с образцом:

```
showPerson :: Person -> String
showPerson (Person { name = n, age = a }) = n ++ " is " ++ show a ++ " years
old."
```

Здесь мы извлекли поля `n` и `a` из записи. Можно даже сопоставлять по части полей: `Person { age = 0 }` в образце, например, матчится на `Person` с `age` 0 (возраст ноль).

Record syntax – это лишь синтаксический сахар, но он очень распространён для типов с многими полями.

Классы типов. В Haskell *классы типов* (`typeclasses`) – это не ООП-классы, а скорее интерфейсы, описывающие некоторое поведение типов [28]. Если тип является экземпляром класса, это значит, что для него реализованы определённые функции (методы), заданные этим классом. Классы типов позволяют реализовать **ад-хок полиморфизм** – грубо говоря, перегрузку функций для разных типов.

Например, есть класс типов `Eq`, который определяет операцию проверки на равенство (`(==)`) и неравенство (`(/=)`). Если тип принадлежит классу `Eq`, его значения можно сравнивать на равенство. Большинство встроенных типов (числа, символы, списки и т.д.) являются экземплярами `Eq` [29]. Аналогично, класс `Ord` – для упорядочиваемых типов (имеют операции `>`, `<` и т.д.) [30]; класс `Show` – для типов, значения которых можно представить в виде строки (метод `show`) [31]; класс `Read` – обратный `Show`, умеет парсить строку в значение (`read`) [32], и так далее.

Пример: оператор `(==)` имеет тип `(Eq a) => a -> a -> Bool` [33]. Часть `(Eq a) =>` – это **контекст класса**: говорит, что для любого типа `a`, если `a` принадлежит классу `Eq`, то `(==)` может принимать два `a` и вернуть `Bool`. То есть, мы можем сравнить два значения типа `a` оператором `(==)`, только если `a` реализует `Eq`. Попытка вызвать `(==)` на типе, не являющемся экземпляром `Eq`, приведёт к ошибке компиляции.

Для пользовательских типов можно сделать их экземплярами стандартных классов. Часто это делается автоматически через `deriving`. Например:

```haskell
data Color = Red | Green | Blue
  deriving (Eq, Show)
```

Это попросит компилятор сгенерировать реализацию `Eq` и `Show` для типа `Color`. В результате мы сможем сравнивать цвета: `Red == Green` будет `False`, а также печатать их: `show Red` вернёт строку `"Red"`. `deriving` умеет автоматически выводить экземпляры для некоторых стандартных классов (`Eq`, `Ord`, `Show`, `Read` и других), если тип удовлетворяет необходимым условиям (например, все поля тоже реализуют эти классы).

Можно и вручную написать экземпляр класса для своего типа с помощью ключевого слова `instance`. Например, сделаем тип `Person` экземпляром `Show`, чтобы красиво выводить информацию о человеке:

```haskell
instance Show Person where
    show (Person { name = n, age = a }) = "Person(" ++ n ++ ", " ++ show a ++
")"
```

Теперь `show somePerson` вернёт строку, сформированную по нашему шаблону.

Класс может также подразумевать ограничение на тип-параметр (например, `class Num a where ...` требует, чтобы `a` реализовывал `Eq` и `Show` – в определении класса это можно указать). Но в базовых случаях таких сложностей немного.

Также важно: один и тот же тип может быть экземпляром нескольких классов. Например, тип `Int` является и `Eq`, и `Ord`, и `Show`, и `Num` и др. В объявлении функции мы можем потребовать, чтобы тип был экземпляром нескольких классов сразу, через синтаксис `(Eq a, Show a) => ...` – такой контекст означает: "тип `a` должен реализовывать *и* `Eq`, *и* `Show`".

Классы типов – мощнейшая способность Haskell, позволяющая, например, создавать обобщённые функции (как `(+)` или `(==)`), работающие с разными типами, но при этом сохраняющие статическую типобезопасность. В более продвинутом коде появляются собственные классы типов (например, `Monad` – класс типов для монад, `Functor` – для функторов, и т.д.), но для начала достаточно освоить базовые: `Eq`, `Ord`, `Show`, `Read`, `Enum`, `Bounded` и несколько других.

Пример использования классов: функция, которая требует, чтобы её аргумент можно было и печатать, и сравнивать:

```haskell
checkAndShow :: (Eq a, Show a) => a -> a -> String
checkAndShow x y = if x == y
                   then "Equal: " ++ show x
                   else "Not equal: " ++ show x ++ " and " ++ show y
```

Здесь контекст `(Eq a, Show a) =>` говорит, что для типа `a` должны существовать реализации `Eq` и `Show`. В теле функции мы сравниваем `x` и `y` (требуется `Eq`) и выводим их через `show` (требуется `Show`). Вызвать `checkAndShow` можно на любых типах, удовлетворяющих этим ограничениям – например, на Int, на Bool, на String, на нашем `Color` (мы сделали `Color` производным от Eq и Show) и т.д.

12. Краткий обзор библиотеки Gloss (графика)

Для создания простых графических приложений (например, игры) на Haskell часто используют библиотеку **Gloss** – она предоставляет удобный интерфейс для 2D графики, анимации и обработки ввода. Gloss абстрагирует детали работы с OpenGL, позволяя рисовать примитивы, формы и анимировать их без сложности.

Установка Gloss. Предположим, у вас уже установлен компилятор GHC и инструмент сборки Cabal или Stack. Gloss – это пакет, доступный через Hackage. Если вы используете Cabal, можно установить Gloss командой, например:

```
cabal update && cabal install gloss
```

Это загрузит и соберёт библиотеку Gloss и её зависимости [34]. В случае со Stack, нужно добавить `gloss` в раздел `dependencies` вашего проекта (например, в файл `.cabal` или `package.yaml`), и затем запустить `stack build`. После успешной установки вы сможете импортировать модуль `Graphics.Gloss` в своем коде.

Простейший пример с Gloss. Нарисуем окно с простым изображением – скажем, окружностью. Код программы:

```
import Graphics.Gloss

main :: IO ()
main =
  -- создаём окно размером 200x200, с заголовком "Circle", позиция окна
(100,100)
  display (InWindow "Circle" (200, 200) (100, 100))
          white            -- цвет фона (белый)
          (Circle 80)      -- рисунок: окружность радиуса 80
```

Давайте разберём этот код. Мы импортируем модуль `Graphics.Gloss`, который предоставляет необходимые функции. Функция `display` открывает окно и рисует в нём картинку [35]. У неё три параметра: 1. **Display** – режим окна. Здесь мы используем `InWindow "Circle" (200,200) (100,100)`, что означает окно размером 200x200 пикселей, заголовок "Circle", расположить окно на экране по координатам (100,100) от левого верхнего угла экрана. Можно вместо этого использовать `FullScreen` для полноэкранного режима. 2. **Color** – цвет фона. Мы указали `white` (белый) фон. В Gloss есть заранее заданные цвета (`black`, `white`, `red`, `blue`, ...). 3. **Picture** – картинка, которую

нужно нарисовать. Gloss определяет тип `Picture` и ряд функций/конструкторов для него. Мы используем конструктор `Circle 80` – окружность радиуса 80 (в тех же условных единицах, что и пиксели, хотя по умолчанию 1 единица = 1 пиксель)[36]. Конструктор `Circle` рисует незакрашенную окружность (контур). Есть также `circleSolid 80` для заполненного круга.

Когда вы скомпилируете и запустите эту программу, появится окно с белым фоном и чёрным кругом радиуса 80, вписанным в центр окна[37]. Закрыть окно можно, нажав Esc или закрыв окно средствами ОС.

Gloss позволяет намного больше: можно комбинировать примитивы (функция `pictures` для списка картинок), трансформировать их (функции `translate`, `rotate`, `scale` для смещения, вращения и масштабирования фигур) и даже делать анимацию и игры через функции `animate` и `play`. Но для начала важно, что мы буквально несколькими строками смогли открыть графическое окно и что-то нарисовать.

Обработка ввода. Gloss имеет упрощённый механизм событий. В примере выше мы используем `display`, которая просто рисует статическую картинку. Для интерактивности есть функция `play`, у которой можно указать обработчики событий (клавиатура, мышь) и шагов времени. Это более продвинуто, но документация Gloss и примеры на Hackage могут помочь, когда вы захотите сделать игру. Главное – начать с маленького: нарисовать фигуру, попробовать сдвинуть её, затем реагировать на нажатия клавиш. Gloss хорошо подходит для учебных проектов, позволяя быстро увидеть результат и не отвлекаться на низкоуровневое программирование графики.

Установка и запуск резюме: Убедитесь, что Gloss установлена (через cabal или stack), импортируйте `Graphics.Gloss`, затем используйте `display` / `animate` / `play` в вашем `main`. Например, наш код с `display` выше – минимальный самодостаточный пример. Компилируем (`ghc Main.hs -o circle`) и запускаем – видим окно с нарисованным кругом.

Эта методичка покрывает базовые и промежуточные концепции Haskell, помогая сделать путь от вывода "Hello, world!" до написания простых приложений и игр. Материал получился сжатым, но постарайтесь усвоить каждую часть пошагово. Haskell – мощный язык, и хотя поначалу многие концепции (ленивость, монады, рекурсия) могут быть непривычными, с практикой вы оцените лаконичность и выразительность кода на Haskell. Удачи в изучении!

---

1 2 5 6 7 28 29 30 31 32 33 Типы и классы - Выучи Haskell во имя добра!
https://learnhaskellforgood.narod.ru/learnyouahaskell.com/types-and-typeclasses.html

3 Ленивые вычисления — Википедия
https://ru.wikipedia.org/wiki/%D0%9B%D0%B5%D0%BD%D0%B8%D0%B2%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8

4 16 17 18 19 20 Haskell. Типы данных, паттерн матчинг и функции
https://www.lovemedia.net/en/articles/haskell-pattern-matching/

8 9 10 11 Немного о каррировании в Haskell / Хабр
https://habr.com/ru/articles/249241/

12 13 Рекурсия <- О Haskell по-человечески
https://www.ohaskell.guide/recursion.html

14 15 Хвостовая рекурсия: ankalagonblack — LiveJournal
https://ankalagonblack.livejournal.com/14631.html

21 22 Haskell/Modules — Викиучебник
https://ru.wikibooks.org/wiki/Haskell/Modules

23 Ленивые вычисления / Хабр
https://habr.com/ru/articles/131910/

24 9. Редукция выражений
https://anton-k.github.io/ru-haskell-book/book/9.html

25 26 27 Haskell. Классы типов, монады
https://www.lovemedia.net/en/articles/haskell-monads/

34 35 36 37 Your First Haskell Application (with Gloss) - Andrew Gibiansky
https://andrew.gibiansky.com/blog/haskell/haskell-gloss/