Если вы видите что-то необычное, просто сообщите мне.

Отрыв

Если вы всегда мечтали начать изучать Haskell и не знаете откуда начать, не ищите дальше! Наш серия "Отрыв" это руководство разработано для того, чтобы провести вас от базовых знаний о язык до написания полноценного кода. Вы начнете с получения всех необходимых инструментов на компьютер. Затем вы изучите базовые механизмы языка и синтаксис. А закончите написанием своего собственного типа данных.

- Haskell 101: Установка, Выражения, Типы
- Модули и синтаксис функций
- Делая свой тип.

Haskell 101: Установка, Выражения, Типы

Добро пожаловать в первую часть серии Отрыва Понедельнечного Хаскельного Утра! Если вы мечтали попробовать изучить Haskell,но никогда не могли найти хорошее руководство для этого, вы в правильном месте! У вас может не быть знания об этом прекрасном языке. Но после прочтения трех статей, вы должны будете знать базовые идеи достаточно, чтобы начать программировать самостоятельно.

Эта статья покрывает несколько различных тем. Первая, мы скачаем всё необходимое и установи. Затем мы начнем писать наше первое выражение и изучим немного про систему типов в Haskell. Дальше, мы поместим "функцию" в функциональное программирование и изучик что Haskell функции являются объектом первого класса. Наконец, мы затронем тему более сложных типов таких как списки и кортежи.

Если вы уже читали эту стать или знакомы со всеми этими концептами, вы можете перепрыгнуть ко второй части. В ней мы поговорим о написании своих фалов с кодаи и написании более сложных функций с некоторым дополнительным синтаксом. Обязательно загляните в главу 3, где мы посмотрим на то, как легко создавать свой собственный тип данных!

Эта серия, так же, с примерами в репозитории Github. Этот репозиторий позволит вам работать с некоторыми примерами кода из этих статей. В этой первой части, мы в основном будет работать с GHCI, нежели с файлами.

Наконец, как только вы закончите с этим, проверьте себя с помощью чеклиста. Это даст вам возможность проверить свои знания со всех сторон.

Установка

Если вы еще не касались Haskell совем, первый шаг - скачать платформу Haskell. Скачаем последнюю версию для вашей ОС и проследуем по подсказкам на экране.

Платфрма содержит 4 главных сущности. Первая - `GHC`, широко распространненый компилятор Haskell. Компилятор это то, что превращает код в что-то что компьютер может запустить. Второе - `GHCI`, интерпретатор для языка Haskell. Он позволяет вам вводить выражения и тестировать некоторые вычисления без того, чтоб использовать отдельный файл.

Третье - `Cabal`, менеджер зависимости для Haskell библиотек. Он позволяет вам скачивать код, который другие люди уже написали и используют в своих проектах. Наконец, инструмент `Stack`. Он добавляет еще один слой поверх Cabal и делает его проще для скачивания пакетов, с которыми не хотелось бы иметь конфликтов. Если хотите более детальное рассмотрение этой темы, можно взглянуть на `Stack Mini-Course`!

Чтобы проверить. что у вас все работает правильно, нужно запустить команду `ghci` в вашем терминале и дождаться запуска интепретатора. Мы проведем остаток этой лекции в GHCI пытая некоторые базовые свойства языка.

Выражения

У вас уже все установленно, давайте пойдем дальше! Самое фундаментальное в Haskell - всё что пишется это выражение. Все программы состоят из вычисления этих выражений. Давайте начнем с проверки некоторых, самых простых выражений, которые мы можем сделать. Веедите следующее выражение в интерпретатор. Каждый раз при нажатии `enter`, интерпретатор должен просто выводить обратно то, что вы ввели.

```
>> True
True
>> False
False
>> 5
5
>> 5.5
5.5
```

```
>> 'a'
'a'
>> "Hello"
"Hello
```

Этим набором выражений, мы покрыли большую часть базовых типов языка. Если вы делали программы ранее, эти базовые типы должны быть вам хорошо знакомы. Первыве два выражения - булевы. `True` и `False` - единственные значения этого типа. Мы так же можем делать выражения из чисел, целых и десятичных. Наконец, мы можем делать выражения отображающием отдельные символы так же как и целые слова, которые мы назовем `string`.

В интерпретаторе, мы можем назначить выражения для наименования используя `let` и знак равно. Это сохранит выражение под именем к которому мы можем ссылаться позже.

```
>> let firstString = "Hello"
>> firstString
"Hello"
```

Тип

Теперь, одно из классных вещей о Haskell это то, что любое выражение имеет тип. Давайте проверим тип базового выражения которое мы ввели вышее. Мы увидим, что идея о которой мы говорим формализованна и самом языке. Вы можете посмотреть тип любого выражения используя команду `:t commang`.

```
>> :t True
True :: Bool
>> :t False
False :: Bool
>> :t 5
5 :: Num t => t
>> :t 5.5
5.5 :: Fractional t => t
>> :t 'a'
'a' :: Char
>> :t "Hello"
```

```
"Hello" :: [Char]
```

Пара выражений проста, но другая пара кажется странной. Последнее выражение это же просто строка? Верно. Вы можете использовать понятие `String` в вашем коде. Но под капотом, Haskell думает о строках как о списке символов, о чем говорит `[Char]`. Мы вернемся к этому позже. `True` и `False` отвечает за тип `Bool`, как мы и ожидаем. Символ `a` просто единичный `Char`. Наши числа немного сложнее. Временно игнорируем слова `Num` и `Fractional`. Это то как мы можем ссылаться на различные типы. Мы будем представлять себе целые числа в качестве `Int` типа, а с плавающей запятой как `Double`. Мы можем явно назначить тип:

```
>> let a = 5 :: Int
>> :t a
a :: Int
>> let b = 5.5 :: Double
>> :t b
b :: Double
```

Мы уже можем увидеть, что-то очень интересно о Haskell. Он может взаимодействовать с информацией о типе нашего выражения просто исходя из формы. В общем, нам не нужно явно давать тип для каждого нашего выражения как мы делали в языках Java или C++.

Функции

Давайте начнем делать некоторые вычисления с нашими выражениями и увидим, что будет происходить. Мы можем начать с которых базовых математических вычислений:

```
>> 4 + 5
9
>> 10 - 6
4
>> 3 * 5
15
>> 3.3 * 4
13.2
>> (3.3 :: Double) * (4 :: Int)
```

В то время, как мы закончили с этой частью, мы поняли что здесь происходит и как мы можем это исправить. Теперь, важная заметка, всё в Haskell - выражение, и любое выржаение имеет свой тип. Логично, мы должны уметь узнавать и определять типа этих различных выражений. И мы определенно можем это делать. Нам нужно просто обернуть в скобки. чтобы убедиться, что тип команды знал, что нужно включить выражение целиком.

```
>> let a = 4 :: Int
>> let b = 5 :: Int
>> a + b
9
>> :t (a + b)
(a + b) :: Int
```

Оператор `+`, даже сам по себе без числе, всё еще выражение! Это наш первй пример функции, или выражения которое принимает аргументы. Когда мы обращаемся к нему самому то его нужно обернуть в скобки.

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

Это наш первый пример отражения типа функции. Важная часть тут - `a -> a -> a`. Это выражение говорит нам что `(+)` это функция которая принимает два аргумента, которые дожны иметь один и тот же ти. И затем выдает нам результат того же типа, что и входные данные. `Num` указывает, что нам нужно использовать числовые типы, вроде целых и с плавающей запятой. Мы не можем например сделать так:

```
>> "Hello " + "World"
```

Но есть объяснение тому, почему нельзя сложить напрмер `Int` и `Double` вместе. Функция требует использовать одинаковый тип для обоих аргументов. Чтобы это исправить, нам нужно использовать другую функци для того. чтобы изменить тип одного из аргумента, чтобы он совпадал с другим. Или мы можем позволить взаимодействию типов разрешить это самому, как мы делали это в примере выше. Но мы бежим вперед поезда. Давайте остановимся на смысле того как мы "применяем" эти функции.

В общем, мы "применяем" функции помещая аргумент после функции. Фнукция `(+)` специальная, так как мы можем использовать её между аргументами. Если мы всё таки

хотим, то можем использовать скобки вокруг нее и поставим как обычную функцию вначале. В этом случае оба аргумента будут и стоять после.

```
>> (+) 4 5
9
```

Что важно знать про функции, то что не обязательно использовать сразу все аргументы. Мы можем взять тот же оператор сложения и применит только одно число. Это называется частичное применние.

```
>> let a = 4 :: Int
>> :t (a +)
(a +) :: Int -> Int
```

Сам по себе `(+)` оператор которы принимает 2 аргумента. Сейчас мы к нему применили один аргумент, который принимает оставшийся. Дальше, так как один аргумент был `Int` второй тоже должен быть `Int`. Мы можем использовать частичное применение для выражения используя `let` и затем применить второй аргумент.

```
>> let f = (4 +)
>> f 5
9
```

Давайте немного поэкспериментируем с другими операторами, в этот раз с булевым типом. Это очень важно, потому, что они позволят создавать более сложные условия когда начнете писать функции. Это три главных оператора, которые работают таким образом, как вы ожидаете для других языков: `And`, `Or` и `Not`. Первые два принимают два булевых параметра и возвращают один, последний принимает одно значение и возвращает одно.

```
>> :t (&&)
(&&) :: Bool -> Bool -> Bool
>> :t (||)
(||) :: Bool -> Bool -> Bool
>> :t not
not :: Bool -> Bool
```

Ну и взглянем на простые примеры поведения:

```
>> True && False
False
>> True && True
True
>> False || True
True
>> not True
False
```

Последнюю функцию которую мы разберем - функция равенства. Принимает два аргумента почти любого типа и определяет равны ли они или нет.

```
>> 5 == 5
True
>> 4.3 == 4.8
False
>> True == False
False
>> "Hello" == "Hello"
True
```

Списки

Теперь мы собираемся слегка расширить наши горизонты и обсудить еще больше типов. Первая идея на которую взглянем это список. Это последовательность значений, которые имеют один тип. Определяется список с помощью квадратных скобочек. Список может не иметь элементов совсем, и такой пустой список можно вызывать.

```
>> :t [1,2,3,4,7]
[1,2,3,4,7] :: Num t -> [t]
>> :t [True, False, True]
[True, False, True] :: [Bool]
>> :t ["Hello", True]
Error! (these aren't the same type!)
>> :t []
[] :: [t]
```

Отметим ошибку в третьем примере! Списки не могут иметь различныне типы элементов. Помните, мы говорили ранее, что строка это просто список симолов. Теперь посмотрим как выглядит строка:

```
>> "Hello" == ['H', 'e', 'l', 'l', 'o']
True
```

Списки можно объединить используя оператор `(++)`. Так как строки - списки, это позволяет нам комбинировать строки как в любом другом языке.

```
>> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
>> "Hello " ++ "World"
"Hello World"
```

Списки так же имеют две функции, которые специально спроектированны, что получения определенных элементов. Мы можем использовать `head` функцияю, что получения первого элемента списки. И похожим образом, мы можемт использовать `tail` функцию для получения всех элементов, кроме первого(head).

```
>> head [1,2,3]
1
>> tail [True, False, False]
[False, False]
>> tail [3]
[]
```

Внимание! Вызов обоих функции для пустого списка приведет к ошибке!

```
>> head []
Error!
>> tail []
Error!
```

Кортежи

Теперь мы знаем о списках, вы можете гадать, если есть способ объединять элементы которые не имеют одинаковый тип. На самом деле есть! Называются они Кортежи! Можно создать кортеж, который будет иметь любое количество элементов, который со своим типом. Кортежи обозначаются с помощью круглых скобок.

```
>> :t (1 :: Int, "Hello", True)
(1 :: Int, "Hello", True) :: (Int, [Char], Bool)
>> :t (1 :: Int, 2 :: Int)
(1 :: Int, 2 :: Int) :: (Int, Int)
```

Каждый кортеж, котоый мы делаем имеет свой собственный тип основываясь на типах элементов внутри кортежа. Это значит, что следующие любые типы, даже если элементы будут иметь одинаковый тип, или иметь одинаковую длинну.

```
>> :t (1 :: Int, 2 :: Int)
(1 :: Int, 2 :: Int) :: (Int, Int)
>> :t (2 :: Int, 3 :: Int, 4 :: Int)
(2 :: Int, 3 :: Int, 4 :: Int) :: (Int, Int, Int)
>> :t ("Hi", "Bye", "Good")
([Char], [Char], [Char])
```

Так как кортежи это выражения, как и другие, мы можем его выводить! Однако, мы не можемт объединять кортежи различных типов в один список.

```
>> :t [(1 :: Int, 2 :: Int), (3 :: Int, 4 :: Int)]
[(1 :: Int, 2 :: Int), (3 :: Int, 4 :: Int)] :: [(Int, Int)]
>> :t [(True, False, True), (False, False, False)]
[(Bool, Bool, Bool)]
>> :t [(1,2), (1,2,3)]
Error
```

Заключение

Конец первой части нашей отрывной серии. Взгляните на то, что мы прошли в одной статье. Мы установили Haskell платформу и начали экспериментировать с GHCI, интерпретатором кода. Мы так же узнали о выражениях, типах, функциях которые являются строительными

элементами Haskell.

Во второй части этого набора, мы начнем писать наш код на Haskell в исходных файлах и изучим еще синтаксис языка. Проверим как мы можемт вывести что-то пользователю из нашей программы, и как можно получить что-то от пользователя на вход. Так же начнем писать наши функции и посмотрим на различные способы для указания поведения функций.

В третьей части, мы начнем создавать свой тип данных. Мы посмотрим насколько просты алгебраические типы данных Haskell, и как типы synonym и newtypes может дать нам дополнительное управление через кодовый стиль.

Модули и синтаксис функций

Вновь, добро пожаловать на серию Отрыв Понедельнечного Хаскельного Утра! Это вторая часть серии. Если вы пропустили первую часть, то вам стоит вернуться к ней, где вы сможете скачать, устаноить все необходимое. Мы так же пройдем через базовые идеи выражений, типов и функций.

Теперь вы возможно думаете: "Изучение типов с помощью интерпритатора - весело! Но я хочу писать настоящий код!" На что поход Haskell синтакс? К счастью, на этом мы и сосредоточимся.

Мы наченм писать наш модуль и функции. Посмотрим на то, как читать наш код в интерпретаторе и как запустить его через испольнительный файл. Еще изучим подробнее синтакс функйий для описания более сложных идей. В части третьей этой серии, мы узнаем, как создать свой тип данных!

Если вы хотите проследовать вместе с примерами кода в этойй части, вы можете пройти в репозиторий на Github и скачать. Ссылки будут указаны дальше в статье.

Написание файлов с исходным кодом

Теперь, вы знакомы с базовыми идеями Haskell, мы должны начать писать наш код. Для этой первой части статьи, вы скачать исходинк с Github. Или вы можете написать самостоятельно. Давайте наченм с открытия файла под названием `MyFirstModule.hs`, и объявим в нем Haskell модуль используя ключевое слово `module` в самом верху файла.

```
module MyFirstModule where
```

Выражение `where` следует за именем модуля и отражает начальную точку нашего кода. Давайте напишем очень простое выржаение, которое наш модуль будет экспортировать. На назначим выражению имя используя знак равно. В отличии от интерпретатора, нам не нужно использовать слово `let`.

```
myFirstExpression = "Hello World!"
```

Когда определяется выражение внутри модуля, распространенная практика это указать его сигнатуту в самом верхнем уровне выражения и функции. Это важно понять для любого кто собирается читать ваш код. Это так же помогает компилятору выводит типы внутри вашего подвыражений. Давайте пойдем дальше, и пометим выражение используя в качестве `String` используя оператор `::`.

```
myFirstExpression :: String
myFirstExpression = "Hello World!"
```

Так же определим нашу первую функцию. Она будет принимать `String` в качестве ввода, и складывать входную строку со строкой "Hello". Отметим, как мы определим тип функции используя стрелку от входного типа в выходной.

```
myFirstFunction :: String -> String
myFirstFunction input = "Hello " ++ input
```

Теперь имея этот код, мы можем загрузить наш модуль в GHCi. Чтобы сделать это запустим GHCi из той же директории где лежит модуль. Вы можеет использовать `:load` команду для загрузки всех опередений выражений, чтобы и меть доступ к ним. Давайте посмотрим на это в действии:

```
>> :load MyFirstModule
(loaded)
>> myFirstExpression
"Hello World!"
>> myFirstFunction "Friend"
"Hello Friend"
```

Если мы изменили наш исходный код, мы можем вернуться обратно и перезагрузить модуль в GHCi используя `:r` команду("reload"). Давайте изменим функции как показано ниже:

```
myFirstFunction :: String -> String
myFirstFunction input = "Hello " ++ input ++ "!"
```

Теперь перезагрузим и запустим еще раз!

```
>> :r
(reloaded)
>> myFirstFunction "Friend"
"Hello Friend!"
```

Ввод и вывод.

В конце, мы хотим иметь возможность запускать наш код без нужны использовать интерпретатор. Чтобы это сделать мы превратим наш модуль в бинарный файл. Делается это с помощью добавления функции под названием `main` со специальной сигнатурой.

```
main :: IO ()
```

Этот и сигнатуры может казаться странным, так как мы еще не говорили ни о каком `IO` или `()` пока. Всё что вам нужно понять, то что этот тип сигнатуры позволяет нашей `main` функции взаимодействовать с терминалом. Мы можем, например, запустить некоторые выражение вывода. Для этого воспользуемся специальным синтаксом называемым "do-syntax". Будем использовать слово `do`, и затем перечислим возможные действия вывода на каждой линии под.

```
main :: IO ()
main = do
  putStrLn "Running Main!"
  putStrLn "How Exciting!"
```

Теперь у нас есть эта главная функция, нам не нужно использовать интерпретатор. Мы можем использовать терминальную команду `runghc`.

```
> runghc ./MyFirstModule
Running Main!
How Exciting!
```

Конечно, вы так же можете хотет иметь возможность читать ввод от пользователя, и вызывать различные функции. Для этого нужно воспользоваться функцией `getLine`. Вы можете получить доступ используя специальный оператор `<-`. Затем с помощью "do-syntax", можно будет использовать `let` как делали в интерпретаторе для назначения выражению имени. В этом случае мы вызовем наше прошлое выражение.

```
main :: IO ()
main = do
  putStrLn "Enter Your Name!"
  name <- getLine
  let message = myFirstFunction name
  putStrLn message
```

Попробуем запустить.

```
> runghc ./MyFirstModule.hs
Enter Your Name!
Alex
Hello Alex!
```

Вот так, мы написали нашу первую маленькую Haskell программу.

IF и ELSE синтакс

Теперь мы собираемся немного подвинуться, и посмотреть на то как мы можем сделать нашу функцию более интересной используя Haskell синтакс конструктор. Есть две возможности для этого. Вы можете ссылаться на полный файл который имеет весь конечный код, который мы пишем в этой части. Или вы можете использовать метод "сделай сам", где придется самостоятельно заполнить определения как показано в статье.

Первая синтаксическая идея которую мы изучем будет выражение `if`. Давайте предполжим, мы хотим попросить пользовтаеля ввести число. Затем вы делаем различные действия в зависимости от того насколько большое число.

Выражение `if` немного отличается в Haskell от того, к чему мы привыкли. Для примера, следующее выражение легко понимается в Java:

```
if (a <= 2) {
  a = 4;
}
```

Такие выражения не могут существовать в Haskell! Все выражения `if` должны иметь `else` ветвление! Чтобы понять почему, нам нужно вернуться к основам из прошлой статьи. Помните, все в Haskell это выражение, и любое выражение имеет тип. Так как мы можемт назначить выражению имя, что оно будет значить для имени если выражение станет `false`? Давайте взглянем на пример правильного `if` выражения:

```
myIfStatement a = if a <= 2
  then a + 2
  else a - 2
```

Это законченное выражение. На первой лини, мы написали выражения типа `Bool`, которое может выдать `True` или `False`. На второй строке, мы написали выражение, которое будет результатом если результат будет `True`. Третья строка сработает если результат проверки будет `False`.

Помните, любое выражение имеет тип. Так каков же тип этого `if` выражения? Предпололжим наш ввод имеет тип `Int`. В этом случае, обе ветви тоже будут `Int`, значит тип нашего выражения должен быть тоже `Int`.

Remember every expression has a type. So what is the type of this if-expression? Suppose our input is an Int. In this case, both the branches (a+2 and a - 2) are also ints, so the type of our expression must be an Int itself.

```
myIfStatement :: Int -> Int
myIfStatement a = if a <= 2
  then a + 2
  else a - 2
```

Что случиться, если мы попробуем сделать, так, что строки будут иметь различный тип?

```
myIfStatement :: Int -> ???
myIfStatement a = if a <= 2
  then a + 2
  else "Hello"
```

Результатом будет ошибка, не важно какой бы тип мы не пытались указать в качестве результат. Это важный урок для выражения `if`. У вас есть две ветви, и каждая ветвь должна выдавать тот же результат. Результирующий тип это тип всего выражения.

Отступление, наш пример будет вести к тому, чтобы вы использовали определенный вид записи. Однако, вы можете собрать всё в одной строке.

```
myIfStatement :: Int -> Int
myIfStatement a = if a <= 2 then a + 2 else a - 2
```

В Haskell нет `elif` выражения как в Python. Но подобный механизм достижим. Вы можете использовать `if` выражение как целое выражение для ветви `else`.

```
myIfStatement :: Int -> Int
myIfStatement a = if a <= 2
  then a + 2
  else if a <= 6
    then a
    else a - 2
```

Охранные выражения(GUARDS)

В случаее когда мы хотите обработать различные ситуации, для читабельности кода в нем можно использовать охранные выражения. Охранные выражения позволяют вам проверять любое число различных условий. Мы можем переписать код выше используя их.

```
myGuardStatement :: Int -> Int
myGuardStatement a
  | a <= 2 = a + 2
  | a <= 6 = a
  | otherwise = a - 2
```

Есть пара тонкостей. Первая - нам не нжно использовать ключевое слово `else` с охранными выражениями, используется `otherwise`. Второе - каждый отдельный случай имеет свой собственный `=` знак, и это не `=` знак для всего выражения. Ваш код не соберется если вы попробуете написать, что-то подобное:

```
myGuardStatement :: Int -> Int
myGuardStatement a = -- BAD!
  | a <= 2 ...
  | a <= 6 ...
  | otherwise = ...
```

Сопоставление с образцом.

В отличии от других языков, Haskell имеет другой способ ветвления в коде кроме булевых типов. Вы можете так же произвести сопоставление с образом(pattern matching). Это позволит изменить поведение кода основываясь на структуре объекта. Для примера, мы можем написать множество версий функции каждя из которых работает на определенном виде аргументов. Вот пример, который ведет себя по другому основывась на типе списка, который он получает.

```
myPatternFunction :: [Int] -> Int
myPatternFunction [a] = a + 3
myPatternFunction [a,b] = a + b + 1
myPatternFunction (1 : 2 : _) = 3
myPatternFunction (3 : 4 : _) = 7
myPatternFunction xs = length xs
```

Первый пример будет совпадать с любым списком который содержит отдельный элемент. Второй пример будет совпадать с любыми примером у которого два элемента. Третий пример использует некоторый синтакс объединения с которым мы еще не знакомы. Но он совпадает с любым списком который начинается с элемента 1 или 2. Следующая строка, любой список, который начинается с 3 и 4. Последний пример будет совпадать с другими списками.

Важно отметить, каким способо шаблоны связывают значения с именами. В первом примере, один элемент списка связан с именем, так, что мы можем использовать его в выражении. В последнем примере, полный список связан с `xs`, поэтому мы можем использовать его в выражении, чтобы мы могли взять его длинну. Давайте посмотрим на эти примеры в действии.

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

> “ Порядок выражений важен! Второй пример имеет такие же шаблоны (1 : 2 : _). Но так как мы сначала указали [1,2] шаблон, он будет использовать эту версию функции. Если мы поставим универсальное значение первым, то всегда будет выполняться только этот универсальный шаблон.

```
-- BAD! Function will always return 1!
myPatternFunction :: [Int] -> Int
myPatternFunction xs = 1
myPatternFunction [a] = a + 3
myPatternFunction [a,b] = a + b + 1
myPatternFunction (1 : 2 : _) = 3
myPatternFunction (3 : 4 : _) = 7
```

К счастью, компилятор предупредит нас о том, что мы не используем какие шаблоны сопоставления с образцом.

```
>> :load MyFirstModule
MyFirstModule.hs:31:1: warning: [-Woverlapping-patterns]
Pattern match is redundant
```

```
In an equation for ‘myPatternFunction': myPatternFunction [a] = ...

MyFirstModule.hs:32:1: warning: [-Woverlapping-patterns]
Pattern match is redundant
In an equation for ‘myPatternFunction': myPatternFunction [a, b] = ...

MyFirstModule.hs:33:1: warning: [-Woverlapping-patterns]
Pattern match is redundant
In an equation for ‘myPatternFunction': myPatternFunction (1 : 2 : _) = ...

MyFirstModule.hs:34:1: warning: [-Woverlapping-patterns]
Pattern match is redundant
In an equation for ‘myPatternFunction': myPatternFunction (3 : 4 : _) = ...
```

Последним хочется отметить, нижнее подчеркивание(как показано выш) может быть использованно для любого шаблона, который мы не хотим использовать. Это универсальная функция и работает для любого значения.

```
myPatternFunction _ = 1
```

Условные выражения

Вы можете использовать сопоставление с образом в середине функции и условными выражениями. Можно переписать прошлый пример так:

```
myCaseFunction :: [Int] -> Int
myCaseFunction xs = case xs of
  [a] -> a + 3
  [a,b] -> a + b + 1
  (1 : 2 : _) -> 3
  (3 : 4 : _) -> 7
  xs -> length xs
```

Отметим, что мы используем стрелку `->` вместо знака равно для каждого случая. Условные выражения более обобщены, проще использовать внутри функции. Для примера:

```
myCaseFunction :: Bool -> [Int] -> Int
myCaseFunction usePattern xs = if not usePattern
  then length xs
  else case xs of
    [a] -> a + 3
    [a,b] -> a + b + 1
    (1 : 2 : _) -> 3
    (3 : 4 : _) -> 7
    _ -> 1
```

WHERE и LET

Если вы пришли из императивного языка, вы должно быть наблюдаете сейчас. И отметили, что похоже мы никогда не объявляем промежуточные переменные. Все выражения, что используются, получаются из шаблонов аргументов. Haskell не имет технически переменных, так как выражения не меняют их значения!Но все еще можем изменить подвыражение внутри нашей функции. Есть пара различных способов для этого. Давайте представим один приме, где мы производим несколько математических операций на входе.

```
mathFunction :: Int -> Int -> Int -> Int
mathFunction a b c = (c - a) + (b - a) + (a * b * c) + a
```

Пока мы можем поздравить друг друга с тем, что функция написана в строку, этот код не совсем читаем. Мы можем сделать его более читаемым используя промежуточные выражения. Для начала сделаем это используя `where` выражение.

```
mathFunctionWhere :: Int -> Int -> Int -> Int
mathFunctionWhere a b c = diff1 + diff2 + prod + a
  where
    diff1 = c - a
    diff2 = b - a
    prod = a * b * c
```

Часть `where` объявляет `diff1`, `diff2` и `diff3` в качестве промежуточоного значения. Потом мы можем использовать их в качестве базы функции. Мы можем использовать `where` результаты друг с другом, и не важно в каком порядке они объявленны.

```
mathFunctionWhere :: Int -> Int -> Int
mathFunctionWhere a b c = diff1 + diff2 + prod + a
  where
    prod = diff2 * b * c
    diff1 = c - a
    diff2 = b - diff1
```

Однако, нужно быть уверенным в том, что вы не делаете цикл `where`, где каждый результат завит от соседнего.

```
mathFunctionWhere :: Int -> Int -> Int
mathFunctionWhere a b c = diff1 + diff2 + prod + a
  where
    diff1 = c - diff2
    diff2 = b - diff1 -- BAD! This will cause an infinite loop!
                      --      diff1 depends on diff2!
    prod = a * b * c
```

Мы можем получить тот же результат используя `let` выражение. Синтаксически похожая формулировка, за исключением нового выражения перед. Нам потом, нужно использовать ключевое слово для указания выражеения которое будет использовать значения.

```
mathFunctionLet :: Int -> Int -> Int -> Int
mathFunctionLet a b c =
  let diff1 = c - a
      diff2 = b - a
      prod = a * b * c
  in diff1 + diff2 + prod + a
```

В ситуации с `IO` как мы писали вывод и чтения, можно использовать `let` в качестве действия без требования. Вам просто нужно сделать это без использования `where` когда ваше выражение зависит от пользовательского ввода.

```
main :: IO ()
main = do
  input <- getLine
  let repeated = replicate 3 input
  print repeated
```

Мы можем обойти эту тему. Мы можем использовать `where` для объявления функции внутри нашей функции. Пример выше можно переписать по другому:

```
main :: IO ()
main = do
  input <- getLine
  print (repeatFunction input)
  where
    repeatFunction xs = replicate 3 xs
```

В этом примере, мы объявили `repeatFunction` как функцию, котораяа принимает список(или String в нашем случае). Этаем на строке `print`, мы передаем входную строку в качестве аргумента в функциюю. Класс!

Заключение

Мы изучили очень много всего! Начали с написания нашего кода, получение ввода, выведения в терминал, и запуска нашего прилоежния в качестве исполнительного файла. Изучили расширенный синтакс функции. Изучили if-выражения, сопоставление с образом, выражения `where` и `let`.

Если вас что-то смутило, не бойетсь, вернитесь и проверьте еще раз первую статью, для того, чтобы устаканить ваши знания в типа выражений! Если вам всё понятно - двигайтесь дальше к следующей статье. В ней мы обсудим различные способы создания нашего собственного типа данных в Haskell.

Делая свой тип.

Вновь, добро пожаловать на серию Отрыв Понедельнечного Хаскельного Утра! Заключительная часть. На случай, если вы прпоустили 2 прошлые главы. В первой части мы обсудили базовую установку Haskell платформы. Затем окунулись в написание базовых выражений на Haskell в интерпретаторе. Во второй части, мы начали с написания нашей собственной функции в модуле Haskell. Так же изучили всяких синтаксических уловок для построения больших и улучшенных функций.

В третьей части мы собираемся углубиться в системы типов. Изучим как создавать свои типы данных, а так же хитрости для упрощения описания наших типов.

Созание нового типа данных

Вперед, к типам данных! Помните, что у нас есть github репозиторий где вы можете получить код для этой части. Если вы хотите реализовать его самостоятельно, вы можете перейти к модулю DataTypes. Но если вы просто хотите посмотреть на завершенный код, вы можете взглянут на DataTypesComplete.

Для этой статьи, предскавим. что мы пытаемся смоделировать некий TODO список. В этой статье создадим несколько различных `Task` типов данных для отражения отдельных задач в списке. Создадим тип данных сначала у которого будет ключевое слово и затем имя типа. Затем добавим оператор присваивания `=`.

```
module DataTypes where


data Task1 = ...
```

В отличии от выражения и функции именя которые мы использовали в ранее, наши типы начинаются с заглавной буквы. Это то что отличает типы от обычных выражений в Haskell.

Теперь собираемся создать наш первый конструктор. Это специальный тип выражения, который позволяет нам создавать объект нашего типа `Task`. Они имеют схожесть с конструкторами скажем на Java. Но они они так же очень сложны. Конструкторы имеют Заглавные буквы а так же список типов. Этот список типов содержит информацию которую хранит конструктор. В нашем случае, мы хотим, чтобы наша задача имела имя и ожидаемоее время выполнения в минутах, отражены как String, и Int соответственно.

```
data Task1 = BasicTask1 String Int
```

Вот так, теперь мы можем начать создавать Task объекты. Например, давайте определим пару простых задач как выражения в нашем модуле.

```
assignment1 :: Task1
assignment1 = BasicTask1 "Do assignment 1" 60

laundry1 :: Task1
laundry1 = BasicTask1 "Do Laundry" 45
```

Мы можем загрузить наш код в интерпретатор, чтоы проверить что он собирается и имеет смысл:

```
>> :l MyData.hs
>> :t assignment1
assignment1 :: Task1
>> :t laundry1
laundry1 :: Task
```

Отметим, что тип нашего выражения `Task1` даже не смотря, что мы собираемся объекты используя `BasicTask1Constructor`. В Java, можно иметь множество конструкторов для одного типа. Мы можем сделать так же и в Haskell, но выглядит это по сложнее. Давайте определим другой тип для различных мест, где мы можем работать над задачами. Мы можем производить работу над задачами в школе, офисе, дома. Отразим это создава конструктор для каждого из них. Разделим конструктор используя вертикальную черту `|`:

```
data Location =
  School |
  Office |
  Home
```

В этом случае, каждый из конструкторов простая отметка, которая не имеет параметров или данных хранящихся в нем. Это пример `Enum` типа. Мы можем технически сделать различные типы выражения отражающими каждый из них.

```
schoolLocation :: Location
schoolLocation = School

officeLocation :: Location
officeLocation = Office

homeLocation :: Location
homeLocation = Home
```

Но эти выражения не более полезны чем использовать сами конструкторы.

Теперь, имея пару типов, мы можем сделать так, что один из наших типов будет содержать другие! Добавим новый конструктор в наш тип задач. Это будет еще сложнее чем просто список мест.

```
data Task1 =
  BasicTask1 String Int |
  ComplexTask1 String Int Location
...

complexTask :: Task1
complexTask = ComplexTask1 "Write Memo" 30 Office
```

Это сильно отличается от конструктора в других языках. Мы можем иметь различные поля для различных отображений типов. Можно обернуть совершенно отличающийся тип зависящий от конструктора который мы используем. Это отлично, так как дает нам гибкость, которую ругие языке не могут.

Параметризированные типы

Еще использовать параметризированные типы с другими определениями типов. Это значит, что один или более полей зависят от типа, который был выбран человеком который писал код. Давайте предположим, у нас есть тип, который имеет несколько базовых конструкторов для различных видов времени. Это ограничит наше описание для простоты.

```
data TaskLength =
  QuarterHour |
  HalfHour |
  ThreeQuarterHour |
  Hour |
  HourAndHalf |
  TwoHours |
  ThreeHours
```

Теперь мы хотим описать задачу где время задачи будет выражатся в Int. Но так же хотим, чтобы была возможность описать с помощью нового типа. Давайте сделаем вторую верси нашего `Task` типа, который может использовать оба типа для времени выполнения. Мы можем сделать это с помощью параметризованного типа:

```
data Task2 a =
  BasicTask2 String a |
  ComplexTask2 String a Location
```

Тип стал мистическим, и теперь мы можем его заполнять как хотим. Но теперь при выводе `Task2` типа в сигнатуре, мы должны будет заполнить правильное определение.

```
assignment2 :: Task2 Int
assignment2 = BasicTask2 "Do assignment 2" 60

assignment2' :: Task2 TaskLength
assignment2' = BasicTask2 "Do assignment 2" Hour

laundry2 :: Task2 Int
laundry2 = BasicTask2 "Do Laundry" 45

laundry2' :: Task2 TaskLength
laundry2' = BasicTask "Do Laundry" ThreeQuarterHour
```

```
complexTask2 :: Task2 TaskLength
complexTask2 = ComplexTask2 "Write Memo" HalfHour Office
```

К этом нужно относится с осторожностью, так как это может ограничить нашу возможность делать определнные вещи. Например, мы не можем создать список, который содержит оба и `assignment2` и `complexTask2`. Это потому, что два выражения теперь различные типы.

```
-- THIS WILL CAUSE A COMPILER ERROR
badTaskList :: [Task2 a]
badTaskList = [assignment2, complexTask2]
```

Пример списка

Говоря о списках, мы можем приоткрыть завесу тайны о том, как списки реализованны.

Большое количество синтаксического сахара меняют способ написания списка на практике. Но на уровне кода, списки определяются двумя конструкторами, `Nil` и `Cons`.

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

Как мы ожидаем, тип `List` имеет один параметр. Это то что позволяет нам одновременно иметь `Int` или `String`. Конструктор `Nil` это пустой список. Не содержит объектов. Поэтому в любое время, в которое вы будете использовать выражение [], занайте вы используете `Nil`. Второй конструктор складывает один элемент с другим списком. Тип элемента и списка должны, конечно же совпадать. При использовании `:` оператора для добавления элемента в список, вы уже используете `Cons` конструктор.

```
emptyList :: [Int]
emptyList = [] -- Actually Nil

fullList :: [Int]
-- Equivalent to Cons 1 (Cons 2 (Cons 3 Nil))
-- More commonly written as [1,2,3]
fullList = 1 : 2 : 3 : []
```

Еще одна вещь, то что наша структура данных рекурсивна. Мы можем увидеть в `Cons` конструкторе как список содержит другой список с параметрами. Это Работает отлично, покоа есть какой-то базовый случай! Тогда, у нас будет `Nil`. Представьте если у нас есть один конструктор и он принимает рекурсивный параметр. У нас возникает затруднительное положение, из-за того, что мы не знаем как создать любойс писок на первом месте.

Синтаксическая записи

Давайте вернемся к основам, непараметризированному типу данных `Task`. Предположим, нас не волнует в целом объект `Task`. Скорее, мы хотим один из его кусочков, напиример имя или время. Так как наш код - единственный способ сделать это использовать сопоставление с образцом который явит нужное поле.

```
import Data.Char (toUpper)

...

twiceLength :: Task1 -> Int
twiceLength (BasicTask1 name time) = 2 * time

capitalizedName :: Task1 -> String
capitalizedName (BasicTask1 name time) = map toUpper name

tripleTaskLength :: Task1 -> Task1
tripleTaskLength (BasicTask1 name time) = BasicTask1 name (3 * time)
```

Теперь слегка упрощаем. Вы можете использовать нижнее подчеркивание вместо параметра, который вы не хотите исопльзовать. Но несмотря на это, может получится громоздко если у ваш тип имеет ножество полей. Мы можем написать нашу функцию позволяющую иметь доступ к отдельным полям. Под капотом, конечно же, будет сопоставление с образом.

```
taskName :: Task1 -> String
taskName (BasicTask1 name _) = name

taskLength :: Task1 -> Int
```

```
taskLength (BasicTask1 _ time) = time

twiceLength :: Task1 -> Int
twiceLength task = 2 * (taskLength task)

capitalizedName :: Task1 -> String
capitalizedName task = map toUpper (taskName task)

tripleTaskLength :: Task1 -> Task1
tripleTaskLength task = BasicTask1 (taskName task) (3 * (taskLength task))
```

Но это применение нельзя масштабировать, так как нам нужно писать эту функцию для каждого поля, которое мы будем создавать. Теперь представьте насколько легко, использовать метод `setter` в Java. Сравним это с `tripleTaskLength` выше. Нужно протись по всем полям, что не есть хорошо. Отличная новость, в том, что мы можем заставить Haskell написать функцию для нас использовать синтаксис записи. Для этого, всё, что нам нужно это назначить каждому полю в определении нашего типа. Давайте сделаем новую версию `Task`.

```
data Task3 = BasicTask3
  { taskName :: String
  , taskLength :: Int }
```

Теперь можно писать тот же код без `getter` функции которую мы писали выше.

```
-- These will now work WITHOUT our separate definitions for "taskName" and
-- "taskLength"
twiceLength :: Task3 -> Int
twiceLength task = 2 * (taskLength task)

capitalizedName :: Task3 -> String
capitalizedName task = map toUpper (taskName task)
```

Теперь можно создать задачу, мы всё еще можем использвать `BasicTask3` сам по себе. Но для чистоты кода, мы можем так же создать объект используя синтаксическую запись, где мы называли поле:

```
-- BasicTask3 "Do assignment 3" 60 would also work
assignment3 :: Task3
assignment3 = BasicTask3
  { taskName = "Do assignment 3"
  , taskLength = 60 }

laundry3 :: Task3
laundry3 = BasicTask3
  { taskName = "Do Laundry"
  , taskLength = 45 }
```

Мы так же можем написать `setter` еще проще используя синтаксическую запись. Вопользуемся прошлой задачей и затем списоком изменений "changes" чтобы поместить их в скобки.

```
tripleTaskLength :: Task3 -> Task3
tripleTaskLength task = task { taskLength = 3 * (taskLength task) }
```

В общем, мы используем только синтаксическую запись, когда есть один конструктор для типа данных. Мы можем использовать различные поля для различных конструкторов, но только наш код чуток безопаснее. Давайте посмотрим на еще один пример определения `Task`:

```
data Task4 =
  BasicTask4
    { taskName4 :: String,
      taskLength4 :: Int }
  |
  ComplexTask4
    { taskName4 :: String,
      taskLength4 :: Int,
      taskLocation4 :: Location }
```

Проблема текущей системы, в том. что компилятор будет создавать `taskLocation4` функцию, которая будет собираться для любой задачи. Но функция отработает правильно, только когда вызывается `ComplexTask4`. Следующий код, будет собираться даже если будет причиной падения, и чтобы этого избежать:

```
causeError :: Location
causeError = taskLocation4 (BasicTask4 "Cause error" 10)
```

В добавок, в наших различных конструкторах используются различные типы, мы не можем использовать то же имя для них. Это может выглядить странно, когда мы хотим отразить ту же идею с различными типами. Этот пример не соберется потому что GHC не может определять тип функции `taskLength4`. Она даже может иметь тип `Task -> Int` или `Task -> TaskLength`.

```
data Task4 =
  BasicTask4
    { taskName4 :: String,
      taskLength4 :: Int }
  |
  ComplexTask4
    { taskName4 :: String,
      taskLength4 :: TaskLength, -- Note we use "TaskLength" and not an Int here!
      taskLocation4 :: Location }
```

Ключевое слово типа.

Теперь, мы знаем, что большинство входных и выходных типов данных самодельные. Но бывают случаю когда вам не нужно делать этого. Мы можем создать новый тип без создавания полностью нового типа структур. Есть два способа сделать это. Первое это ключевое слово. Оно позволяет вам создавать синонимы для типов, таких как `typedef` ключевое слово в C++. Самое распространенное, как мы видели это `String` это список символов.

```
type String = [Char]
```

Распространненный способ использования для него, это когда вы объединяете множество различных типов в кортеж. Это может быть довольно нужно писать кортеж несколько раз в коде.

```
makeTupleBigger :: (Int, String, Task) -> (Int, String, Task)
makeTupleBigger (intValue, stringValue, (BasicTask name time) =
  (2 * intValue, map toUpper stringValue, (BasicTask (map toUpper name) (2 * time)))
```

Использование синонима далает запись сигнатуры гораздно чище:

```
type TaskTuple = (Int, String, Task)

makeTupleBigger :: TaskTuple -> TaskTuple
makeTupleBigger (intValue, stringValue, (BasicTask name length) =
  (2 * intValue, map toUpper stringValue, (BasicTask (map toUpper name) (2 * length))
```

Конечно, если коллекция будет большой, то стоит сделать полный тип данных для этого. Так же есть некоторые причины почему синонимы типов не всегда лучший выбор. Они могут привести к ошибкам компиляции, с которыми трудно будет работать. Вы возмжожно прошли через несколько ошибок где компилятор уже говорил, что ожидает [Char]. Это было бы понятнее если бы он говорил про String.

И межет так же вести к неинтуитивному коду. Предположим вы используете базовый кортеж вместо типа данных для отображения `Task`. Кто-то может ожидать, что тип `Task` будет иметь свой собственный тип. Затем они будут запутаны тем, что вы работаете с ним как с кортежем.

```
type Task5 = (String, Int)

twiceTaskLength :: Task5 -> Int
-- "snd task" is confusing here
twiceTaskLength task = 2 * (snd task)
```

Новые типы

Последнюю тему которую мы обсдудим будет "newtypes". Это как синоными с одной стороны и `ADT` с другой. Но они всё еще имет уникальное место в Haskell и лучше если вы привыкните пользоваться им. Предположим, мы хотим иметь новое подход для отображения `TaskLength`. Мы хотим использовать обычное число, но мы чтобы он имел свой собственный отдельный тип. Мы можем это сделать с помощью "newtype":

```
newtype TaskLength2 = TaskLength2 Int
```

Синтакс для `newtypes` выглядит похожим на ADT. Однако, `newtype` определение может только иметь один коснтруктор. И этот конструктор может только принимать отдельный тип аргументов. Большое отличие между ADT и `newtype` идет после компиляции вашего кода. В этом примере, не будет различий между `TaskLength` и `Int` типы во время выполнения. Это хорошо, так как большая часть кода для `Int` типа специализированна на быстром выполнении. Если мы сделаем настоящим ADT, это не тот случай:

```
-- Not as fast!
data TaskLength2 = TaskLength2 Int
```

Но с другой стороны, мы можем сделать гораздо больше таких трюков с `newtype`, нежели чем с ADT. Мы можем, например, использовать синтаксическую запись в конструктре для наших `newtype`. Это позволяет нам использовать имя чтобы извлечь значение изнутри без сопоставления с образцом. Часто сопоставление с образом при использовании синтаксической записи для какого-нибудь `un-TypeName` значениия в качестве имени поля. Так же отметим, что мы не можем использовать `newtype` значеение с той же фунецией как изначальный тип. Когда у анс синоним, мы должны сделать следующее:

```
data Task6 = BasicTask6 String TaskLength2

newtype TaskLength2 = TaskLength2
  { unTaskLength :: Int }

mkTask :: String -> Int -> Task6
mkTask name time = BasicTask6 name (TaskLength2 time)

twiceLength :: Task6 -> Int
twiceLength (BasicTask6 _ len) = 2 * (unTaskLength len)
-- The following would be WRONG!
-- 2 *len
```

Теперь, `TaskLength2` это эффективная обертка над `Int`. Это делает его похожим на тип синоним, за исключением того, что мы не можем просто использовать `Int` значение по себе. Как вы видите в примере выше, нам нужно пройти через процесс обертки и разворачивания значения. Это выглядит нудно. Но это очень полезно, так как решает главную проблему

использования типа синонима. Теперь если мы делаем ошибки касающиеся `TaskLength`, компилятор скажет нам о `Tasklength`. Мы не будем гадать какои из синонимов мы пропустили!

Есть другой пример. Предположим у нас есть функция с несколькими целочисленными аргументами. Если мы всгде используем `Int` типб мы легко смешаем порядок аргументов. Но если мы используем `newtype`, компилятор будет отлавливать ошибки этих типов за нас.

Заключение

Это заверешение нашего разговора по поводу создания своего типа данных и заверешение нашей Улетной серии! Если вам нужно освежить знания не забудьте проведать часть 1 и 2.