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

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.

Revision #5
Created 2022-03-11 05:07:49 UTC by gasick
Updated 2022-07-09 13:07:28 UTC by gasick