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

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, установить 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 опередлений 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 предположим, 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 -> 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 -> 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.

Revision #3
Created 2022-03-11 05:13:05 UTC by gasick
Updated 2022-07-12 06:37:40 UTC by gasick