**Московский государственный технический**

**университет им. Н.Э. Баумана**

Факультет «Радиотехнический»

Кафедра «Системы обработки информации и управления»

Курс «Парадигмы и конструкции языков программирования»

Отчет по домашнему заданию

Тема «Язык программирования Haskell»

|  |  |  |
| --- | --- | --- |
| Выполнил: |  | Проверил: |
| студент группы РТ5-31Б: |  | преподаватель каф. ИУ5 |
| Карпенко Д.А. |  | Гапанюк Ю.Е. |
| Подпись и дата: |  | Подпись и дата: |

Москва, 2023 г.

Оглавление

[ВВЕДЕНИЕ. 2](#_Toc157699224)

[ФУНКЦИИ. 2](#_Toc157699225)

[HELLO WORLD! 3](#_Toc157699226)

[ВЫБОР. 3](#_Toc157699227)

[ПАТТЕРН МАТЧИНГ. 4](#_Toc157699228)

[ВЛОЖЕННЫЕ ВЫРАЖЕНИЯ. 5](#_Toc157699229)

[CПИСКИ. 6](#_Toc157699230)

[КОРТЕЖИ. 7](#_Toc157699231)

[ЛЯМБДА ФУНКЦИИ. 8](#_Toc157699232)

[СОБСТВЕННЫЕ ТИПЫ. 9](#_Toc157699233)

[АТД. 10](#_Toc157699234)

[ЗАКЛЮЧЕНИЕ. 11](#_Toc157699235)

# ВВЕДЕНИЕ.

Haskell – чисто функциональный ЯП. Обладает строгой статической типизацией, является компилируемым (но может вести себя как скриптовый).

Что означает функциональный язык? Это значит, что все вычисления в нем происходят за счет вычислений значений функций в математическом понимании. Про понятие чисто функциональные языки будет оговорено позже.

Примеры ФЯП : F#, Scala, APL и другие.

Противоположным к функциональным языкам являются императивные языки программирования, к которым относятся : C/C++, Python, Java и т.д.

Существуют также языки, к которым применимы сразу обе парадигмы, например, Rust и Ruby.

В данной работе будут показаны основы языка Haskell и его особенности, которые сильно отличают его от других.

# ФУНКЦИИ.

Раз функции являются одной из основ ФП. То считаю начать изучение именно с них. Для объявления функции необходимо задать ее название, типы входящих и выходящих аргументов и саму реализацию (тело) функции.

Пример: функция считающая произведение двух чисел.

Proizved :: Int -> Int -> Int

proizved a b = a \* b

Для применения этой функции необходимо также описать функцию main, которая применит функцию печати в консоль. Аргументом этой функции будет являться выражение типа proizved a b = a \* b.

proizved :: Int -> Int -> Int

proizved a b = a \* b

-- IO особая конструкция, позволяющая компилятору понять, что данная функция будет работать с «внешним миром»

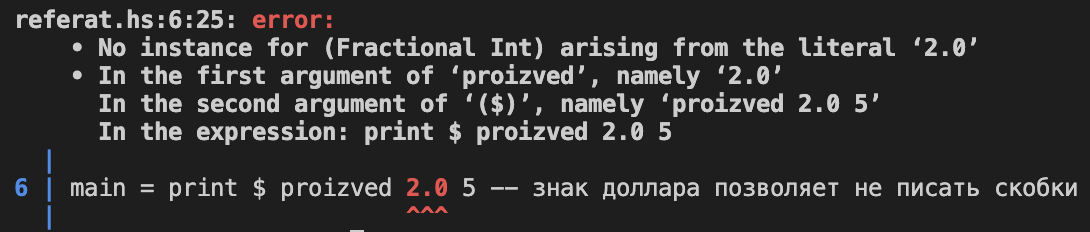
main::IO ()

main = print $ proizved 2 5 – знак доллара позволяет не писать скобки

Результат:



Но если Haskell, как было оговорено выше, обладает строгой статистической типизацией, то это означает, что типы должны совпадать абсолютно везде, так что если мы передадим нашей функции значение, не являющемся значением типа Int, то компилятор тут же выдаст ошибку:



Что и требовалось доказать.

# HELLO WORLD!

По традиции приведу пример программы, печатающий на экран “Hello world!”. Для этого необходимо объявить функцию main внутри которой применим функцию putStrLn:

main::IO ()

main = putStrLn “Hello world!”

Скомпилируем и запустим исполнимый файл программы:



# ВЫБОР.

Условия для выбора можно определять разными способами. Самый классический метод:

pass :: String -> String

pass p =

if p == “password”

then “Welcome”

else “Wrong pass!”

main::IO ()

main = print $ pass «password»

Результат:



Если же заменить аргумент функции любой другой строкой, то будет выполняться ветка “else” с информированием пользователя о том, что пароль неверный:

pass :: String -> String

pass p =

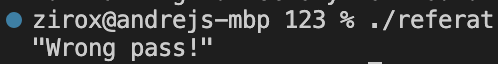
if p == “password”

then “Welcome”

else “Wrong pass!”

main::IO ()

main = print $ pass «123»



# ПАТТЕРН МАТЧИНГ.

Множественное ветвление можно осуществить как и с помощью вложенных конструкций if … else, так и с помощью оператора ИЛИ:

someFun :: Int -> String

someFun x

| x == 1 = “Один”

| x == 2 = «Два»

| x == 3 = «Три»

| otherwise = «Я не умею считать больше трех ☹«

main::IO ()

main = putStrLn $ someFun 2

В данном случае программа вернет «Два». Слово otherwise используется для определения функции в тех случаях, когда ни одно из условий выше не выполнилось.

Данный пример можно описать и другим способом:

someFun :: Int -> String

someFun 1 = “Один”

someFun 2 = «Два»

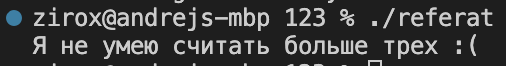
someFun 3 = «Три»

someFun \_ = «Я не умею считать больше трех ☹«

main::IO ()

main = putStrLn $ someFun 12

Вывод:



Примененная конструкция, когда аргумент сравнивается с неким образцом, и называется патерн матчингом, который в свою очередь может использоваться не только в подобных целях.

Еще вариация патерн матчинга case … of:

someFun :: Int -> String

someFun x =

case x of

1 -> “Один”

2 -> “Два”

3 -> «Три»

\_ -> «Я не умею считать больше трех ☹«

main::IO ()

main = putStrLn $ someFun 12

# ВЛОЖЕННЫЕ ВЫРАЖЕНИЯ.

Первый способ описания вложенных выражений – конструкция let … in:

calcS :: Int -> Int -> Int

calcS t v =

let time = t

velocity = v

in

time \* velocity

main::IO ()

main = print $ calcS 3 4

В данном примере мы просто заменили входные переменные синонимами, что делает код более читабельным, но никто не запрещает использовать эту конструкцию для расчета промежуточных значений:

fun :: Double -> Double

fun r =

let pi = 3.14

in

2 \* pi \* r

main::IO ()

main = print $ fun 3

Данная реализация функции, считающей длину окружности, радиуса r, позволяет избавиться от непонятных констант, помещенные в let

Аналогом let … in является ключевое слово where:

fun :: Double -> Double

fun r = 2 \* pi \* r

where

pi = 3.14

main::IO ()

main = print $ fun 3

Результат обоих применений: 18.84

# CПИСКИ.

Списки объявляются с помощью []. Они позволяют хранить множество значений одного типа. Пример:

-- стандартный модуль для работы со списками

import Data.List

main::IO ()

main = print $ head [1, 2, 3, 4, 5]

Мы создали список, состоящий из целочисленного типа значений. Функция head из модуля Data.List позволяет вернуть «голову», т.е. первый элемент списка. Поэтому результатом такой программы будет 1. Есть также функция Tail, которая позволяет вывести весь список без первого элемента

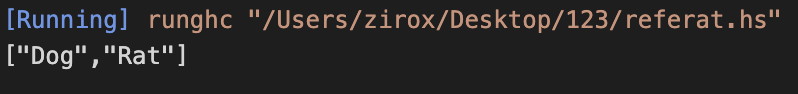
-- стандартный модуль для работы со списками

import Data.List

main::IO ()

main = print $ tail ["Cat", "Dog", "Rat"]

Результат:



Еще пример с использованием списков:

import Data.List

lenInStr :: [a] -> String

lenInStr list

| size == 1 = "Один элемент"

| size == 2 = "Два элемента"

| size == 3 = "Три элемента"

| otherwise = "какая-то строка"

where

size = length list

main::IO ()

main = putStrLn $ lenInStr ["Cat", "Dog", "Rat"]

Здесь [a] – это список произвольного типа, позволяющий работать с любыми списками любого размера.

Так как любые данные в Haskell по умолчанию нельзя изменять в ходе работы программы, то и список менять тоже нельзя. Однако можно создавать новые списки на основе исходных, имитируя таким образом «изменчивость» списков. В качестве примера напишем функцию, которая добавляет элемент в начало списка:

import Data.List

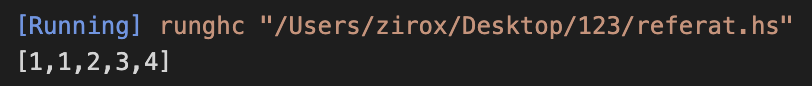
add :: a -> [a] -> [a]

add element list = element : list

main::IO ()

main = print $ add 1 [1, 2, 3, 4]

Результат:



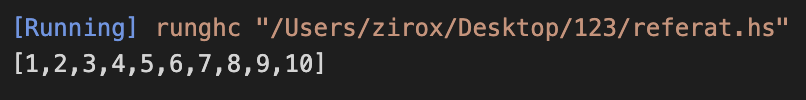
По сути всю работу на себя принимает оператор :, призванный соединять левый аргумент к списку в правом аргументе. Естественно тип присоединяемого значения обязан совпадать с типом списка. Не забываем о строгой типизации.

Также список можно создавать с помощью перечисления:

main::IO ()

main = print [1..10]

Результат:

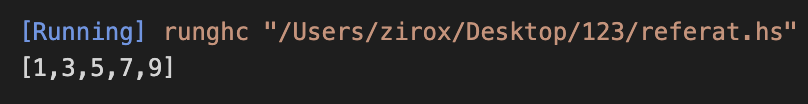


Шаг можно настроить:

main::IO ()

main = print [1, 3..10]

Результат:



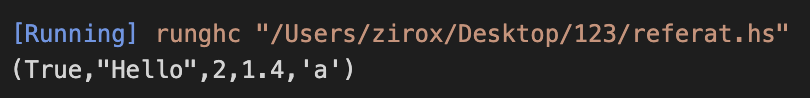
# КОРТЕЖИ.

Кортежы в Haskell объявляются с помощью (). Отличие кортежа от списка в том, что в кортеже можно хранить значение разных типов:

main::IO ()

main = print (True, "Hello", 2, 1.4, 'a')

Результат:



Попробуем сформировать кортеж из 2 значений и извлечь данные из него:

createTuple :: String -> Int -> (String, Int)

createTuple key value = (key, value)

main::IO ()

main =

let (key, value) = createTuple "Цена бутерброда" 120

in

putStrLn $ key ++ " " ++ show value ++ " Рублей"

Здесь использованая функция show, которая преобразует значение в строку для вывода в через putStrLn.

В случае кортежей формирование и чтение данных – основные действия над ними. Более того, редко используются большие кортежы, поскольку для этого существуют списки.

# ЛЯМБДА ФУНКЦИИ.

Лямбда функции определяют так называемые анонимные функции, использующиеся как вложенные. Такие функции зачастую не имеют типа (хотя его можно явно указать) и их тело весьма короткое по сравнению с обычными функциями. Резюмируя можно сказать, что лямбда функции являются частным случаем обычных функций.

main::IO ()

main = print $ (\key value -> (key, value)) "Цена бутерброда" 120

Данная программа выполняет почти то же самое что и в примере с кортежем, но в данном случае мы просто выводим этот кортеж, который сформировали с помощью лямбды.

Еще пример:

main::IO ()

main = print $ square 12 4

where

square = \x y -> x \* y

Таким образом можно сохранять функции в другие выражения, значит мы можем создать, например список функций:

main::IO ()

main = print $ head funcs 12 4

where

funcs = [\x y -> x + y,

\x y -> x - y]

В данном примере ответом будет 16. Мы применили head к funcs получив такм образом первую функцию в списке, после чего она применилась к двум аргументам 12 и 4

# СОБСТВЕННЫЕ ТИПЫ.

Для объявления собственного типа можно воспользоваться ключевым словом data. После чего следует название типа, потом ставится знак равенства, после которого идет непосредственно описание этого типа, т.е. его тело.

Пример:

data Color =

Red

| Green

| Blue

Мы создали тип цвета, в котором создали 3 конструктора: Red, Green и Blue.

В данном примере все эти конструкторы пустые, а значит сами конструкторы являются значениями. Напишем функцию, которая в зависимости от конструктора будет печатать цвет:

data Color =

Red

| Green

| Blue

printColor :: Color -> String

printColor (Red) = "Это красный"

printColor (Green) = "Это зеленый"

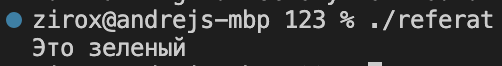
printColor (Blue) = "Это синий"

main::IO ()

main = putStrLn $ printColor Green

В данном случае с помощью паттерна матчинга мы рассматриваем все возможные конструкторы и печатаем строку.

Результат:



С новыми типами можно работать также как и с обычными, приведу пример:

data Color =

Red

| Green

| Blue

printColor :: Color -> String

printColor (Red) = "Its red"

printColor (Green) = "Its green"

printColor (Blue) = "Its blue"

rgb :: Color -> [Color]

rgb c =

case c of

Red -> [Green, Blue]

Green -> [Red, Blue]

\_ -> [Red, Green]

main::IO ()

main = print $ map printColor $ rgb Green

В данном случае я использовал функцию высшего порядка map, ее сигнатура:

Map :: (a -> b) -> [a] -> [b]. Таким образом она переводит значения констркуторов в строковый тип, после чего выполняется печать нового списка.

Сам список был составлен с помощью функции rgb, сопоставляющая конструкторы по образцу.

# АТД.

Алгебраическим типом данных называется собственный тип, в котором существую непустые конструкторы. Сразу пример:

data Person =

Worker String String Int

| Casual String Int

showPerson :: Person -> String

showPerson (Worker name workplace age) = name ++ " " ++ workplace ++ " " ++ show age

showPerson (Casual name age) = name ++ " " ++ " " ++ show age

main::IO ()

main = let human = Worker "Василий" "магазин" 23

in

putStrLn $ showPerson human

В данном примере я создал АТД персона, которая имеет свой уникальный набор типов. Если это рабочий, то используется его имя, место работы и возраст, если же это обычный человек, то у него нет места работы.

Результат:



Но так разобраться что из себя представляет String String Int не всегда легко. Для этого существуют метки:

data Person =

Worker {name :: String,

workplace :: String,

age :: Int}

| Casual {name :: String,

age :: Int}

showPerson :: Person -> String

showPerson (Worker name workplace age) = name ++ " " ++ workplace ++ " " ++ show age

showPerson (Casual name age) = name ++ " " ++ " " ++ show age

main::IO ()

main = let human = Worker "Василий" "магазин" 23

in

putStrLn $ showPerson human

теперь гораздо понятнее, что есть что. Также Haskell допускает следующее:

data Person =

Worker {name ,

workplace :: String,

age :: Int}

| Casual {name :: String,

age :: Int}

то есть можно опускать тип меток, при его повторении, и указывать его только в последней метке, тогда все метки выше автоматически будут этого же типа.

# ЗАКЛЮЧЕНИЕ.

В заключение можно сказать, что данный язык несомненно отличается от всех остальных. В этом знакомстве я определил самые основы языка, но сказанное тут, естественно, далеко не все его аспекты. Эти знания могут послужить дорогой для более глубокого изучения Haskell в дальнейшем.