HaskellReport

farouk.benarous26

June 2019

1 Introduction

Functional programming is a highly valued approach to writing code, and it's popularity is continuously increasing in commercial software applications. Functional programming is a paradigm of writing code and is eloquently put in the introduction of this Wikipedia article:

" -a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions or declarations instead of statements."

We will get into what this exactly means very soon, but first, let's get the fundamentals out of the way by providing some context. This article serves the purpose of hopefully introducing the fundamental concepts of functional programming to you and by the end of it, we'd have established a basic understanding of why we would choose to apply it into our coding styles, how it relates to JavaScript and some basic patterns that you can begin implementing into your code. Be on the lookout for a second article that will build on this one and dig even deeper into the ideas discussed here.

The roots of functional programming lie in Lambda calculus, which was a system developed in the 1930's to express computation using functions similar to what we will see very soon. For quite a while, functional programming concepts and languages were widely discussed in academia and scholarly contexts and eventually bled into the development of commercial software. What makes a programming language functional or not is its ability to facilitate the functional programming paradigm, languages like these are Ruby, Python, Julia and JavaScript, the latter of which has properties of imperative, object-oriented and functional paradigms.

2 Used Tools

Why prolog : Haskell is a statically typed, purely functional programming language with type inference and lazy evaluation. Type classes, which enable type-safe operator overloading, originated in Haskell.[30] Its main implementation is the Glasgow Haskell Compiler. It is named after logician Haskell Curry.

Haskell is based on the semantics, but not the syntax, of the Miranda programming language, which served to focus the efforts of the initial Haskell working group. Haskell is used widely in academia[32][33] and industry.[34] The latest standard of Haskell is Haskell 2010. As of May 2016, the Haskell committee is working on the next standard, Haskell 2020.

3 Development process

3.1 Goal 1 :

insert the consumption of one client in one month

```
insert_a_new_consumption :: IO()
insert_a_new_consumption = do
             print ("Set your Id")
             i<- getLine
             print ("Set your name")
             n<- getLine
             print ("Set your address")
             a<- getLine
             print ("Set your number_of_Persons")
             np<- getLine
             print ("Set your cubicmeters6_15")
             cubis6_15 <- getLine
             print ("Set your cubicmeters0_5")
             cubis0_5<- getLine
             print ("Set your month")
             m <- getLine
             print ("Set your year")
             y <- getLine
             let cubi05 = read cubis0_5 :: Float
             let cubi615= read cubis6_15 :: Float
             let pn = read np :: Float
             let w =  water_total_bill cubi05  cubi615 pn
             let w1 = show (w)
             appendFile "consumption.txt" (i ++ "\t" ++ n ++ "\t" ++ a ++ "\t" ++
             appendFile "invoice.txt" (i ++"\t" ++ w1 ++ "\t" ++ m ++ "\t" ++ y
++ "\n")
             print ("Insert another one ? y/n" )
             resp <- getLine
             if (resp=="y" || resp=="Y") then insert_a_new_consumption
else return()
```

3.2 Goal 2 :

create the invoice of a client in Latex format based on the consumption of one month

```
create_Layex_format  = do
      print ("Set your Id")
      i<- getLine
      print ("Set the month ")
      m<- getLine
      print ("Set the year ")
      y<- getLine
      print ("Set the adress    ")
      a<- getLine
      print ("Set the name  ")
      n<- getLine

      let i1 = show(i)
      let m1 = show (m)
      let y1 = show (y)
      let a1 = show (a)
      let n1 = show (n)
      list <- loadTab_invoice
      let x =search_invoice i  list


      appendFile "latex.tex" ("/documentclass{article}" ++ "\n" ++
            "/usepackage{graphicx}" ++ "\n" ++"/graphicspath{ {./images/} }" ++ "\
            "/usepackage[utf8]{inputenc}" ++ "\n" ++ "/usepackage[T1]{fontenc}" ++
             "/usepackage{hyperref}" ++ "\n" ++ "/usepackage{url}" ++ "\n"
++ "/usepackage{booktabs}" ++ "\n"
            ++ "/title{Invoice of Water Consumption}" ++"\n" ++"/date{May 2019}"+-
            ++ "\n"++ "/begin{document}" ++ "\n"++ "/maketitle" ++"\n" ++ "/begin
            ++ "This invoice is generated using the latex format in the prolog la
            ++ "/end{abstract}" ++ "\n" ++ "/keywords {Datascience /and Programmi
            ++ "\n" ++ "/section{Water Invoice}" ++ "\n" ++ "Client ID : " ++ i1
            ++ "\n" ++ "Year of the bill : "++ y1 ++"\n" ++"Adress of the client
            ++ "\n" ++ "Bills of client : " ++ x ++ "\n" ++ "/end{document}" )
      -- so here i have to create a function that takes as
an argument id client and it will return

      print("Done ")
```

3.3 Goal 3 :

calculate the average consumption for one client in one year

```
average_of_both_consumption  = do
      print("Enter Client Id : ")
      i<- getLine
      print("Enter the year you want to check average for : ")
      y<- getLine
      list <- loadTab_Consumption
      let x = search_consumption i y list
      print("Average Consumption of the year ")
      print(i)
      print(" is ")
      print(x/12)
      print("cubicmeters.")
```

3.4 Goal 4 :

calculate the total payment for one client at the end of one year

```
total_payment = do
      print("Enter Client Id : ")
      i<- getLine
      print("Enter the year you want to check average for : ")
      y<- getLine
      list <- loadTab_invoice
      let x = search_payment i y list
      print("total of payment made in the year ")
      print(y)
      print("is ")
      print(x)
```

3.5 Goal 5 :

total number of cubicmeters consummated over 5 cubicmeters of all clients in the last year

```
check_over_5 = do
      print("Enter the year ")
      y <- getLine
      list <- loadTab_Consumption
      let x = check_over_5_all y list
      print ("the result is : ")
      print(x)

check_over_5_all ye [] = 0
check_over_5_all ye ((i , n , a , np , cubis6_15 , cubis0_5 , m , y):xs) =
      if (ye==y && cubis0_5 > "5" ) then  0 + (read (cubis0_5)::Float)
+ check_over_5_all  ye xs  else check_over_5_all  ye xs
```

3.6 Goal 6 :

generate a graph connecting the monthly payment of a client during one year
and the size of each node must be proportional to that payment value.

```
appendFile "fich.dot" ("Digraph g { rankdir = LR ;node [style=filled]; "++ "Janu
                  "February [lable ='February' ,height ="++
feb  ++"]"  ++  "March [lable ='March' ,height ="++ mar ++"]"
++
                  "April [lable ='April' ,height ="++apr ++"]" ++
"May [lable ='May' ,height ="++may++"]" ++ "June [lable ='Jun' ,height ="++ juin
                  "July [lable ='July' ,height =" ++julay ++"]" ++ "August [lable
++ "September [lable ='September' ,height ="++ sep ++"]"
++
                  "October [lable ='October' ,height =" ++oct ++"]"
++"November [lable ='November' ,height =" ++oct ++"]" ++"December [lable ='Decem
)
```