SCHOOL OF ENGINEERING AND TECHNOLOGY

FINAL ASSESSMENT FOR THE BSC (HONS) COMPUTER SCIENCE; YEAR 2

ACADEMIC SESSION : SEPTEMBER 2023

PRG2214: Functional Programming Principles

Duration: 1 Week

Project **DEADLINE: Week 14**

INSTRUCTIONS TO CANDIDATES

- This assignment will contribute 50% to your final grade.
- This is an individual assignment.

IMPORTANT

The University requires students to adhere to submission deadlines for any form of assessment. Penalties are applied in relation to unauthorized late submission of work.

- Late submissions will be awarded 0%

Lecturer's Remark (Use additional sheet if required)

I.....Thien Tze Yea..... (Name) .....22022123....std. ID received the assignment and read the comments.....Thien 22/12/2023..... (Signature/date)

Academic Honesty Acknowledgement

"I .....Thien Tze Yea ….....(student name). verify that this paper contains entirely my own work. I have not consulted with any outside person or materials other than what was specified (an interviewee, for example) in the assignment or the syllabus requirements. Further, I have not copied or inadvertently copied ideas, sentences, or paragraphs from another student. I realize the penalties *(refer to page 16, 5.5, Appendix 2, page 44 of the student handbook diploma and undergraduate programme)* for any kind of copying or collaboration on any assignment."

….....Thien 22/12/2023..... (Student's signature / Date)

Table of Contents

No	Content	Pages
1	Introduction	1
2	Data Types	1
3	Functions	2-14
4	Personal Reflection	15-16
5	References	17

Introduction

In the fast-paced modern era, maintaining a balanced life requires effective organization. RoutineSync emerges with the purpose of providing individuals a comprehensive personal management tool. Designed with the concept of functional programming in Haskell Language, my aim is to empower users to effortlessly manage tasks, finances, and personal reflections in one centralized platform.

Data Types

```
data Entry = JournalEntry { journalDate :: String, journalText :: String } | FinancialTransaction { transactionType :: TransactionType, transactionDescription :: String, transanctionAmount :: Double } deriving (Eq, Show)
```

Above data type represents an entry in RoutineSync. It has two constructors – 'JournalEntry' and 'FinancialTransaction'. 'JournalEntry' takes in journalDate and journalText. 'FinancialTransaction' takes in transactionType that has a data type of TransactionType, transactionDescription and transactionAmount.

```
data TransactionType = Income | Expense deriving (Eq, Show)
```

TransactionType is a data type that takes in the type of transaction input by user. It can only be either 'Income' or 'Expense'. User must define the type of transaction before they can input the description and the amount.

```
data Todo = TodoTask { todoTitle :: String, todoDescription :: String, todoStatus :: String } deriving (Eq, Show)
```

Data Type Todo is a data type that takes in user input for todoTitle (the title for the to-do task), todoDescription (the description for the to-do task), and todoStatus (the status of the to-do task).

```
data User = User { name :: String, entries :: [Entry], todos :: [Todo] } deriving (Eq, Show)
```

User datatype has a User constructor, which takes in user's name, a list of 'Entry' representing the user's journal entries and financial transactions, and a list of 'Todo' representing the user's to-do tasks.

Functions

In this part, functions will be explained according to modules:

Main Module (Main.hs):

```haskell
main :: IO ()
main = do
  putStrLn "Please enter your name: "
  name <- getLine
  let initialUser = User name [] []
  _ <- mainMenu initialUser
  return ()
```

The main function initiates the program, prompting the user for their name, creating an initial user, and invoking RoutineSync. It operates in the IO monad, managing input/output interactions and program flow.

Types Module (Types.hs):

```haskell
addEntry :: User -> Entry -> User
addEntry user entry = user { entries = entry : entries user }
```

This function is used to update the user's record by adding new journal entries or financial transactions. It is a function that is used across multiple modules and functionalities.

MainMenu Module (MainMenu.hs):

```
mainMenu :: User -> IO User
mainMenu user = do
  putStrLn ""
  putStrLn "------------------------------------------------"
  putStrLn "===== RoutineSync - Personal Management Tool ====="
  putStrLn ""
  putStrLn "1. To-Do List"
  putStrLn "2. Financial"
  putStrLn "3. Journal"
  putStrLn "4. Exit"
  putStrLn ""
  putStrLn "================================================"
  putStrLn "------------------------------------------------"
  putStrLn ""
  putStr "What would you like to do?: "
  putStrLn ""
  choice <- getLine
```

mainMenu function is a function that serves as the main menu of RoutineSync. This is the first thing that user would see when they run the program. It displays options for To-Do List, Financial, Journal and Exit. It interacts with user to navigate between different functionalities and returns an updated ‘User’ after each operation.

```haskell
case choice of
  "1" -> do
    updatedUser <- displayTodoListManagement user
    mainMenu updatedUser

  "2" -> do
    updatedUser <- displayFinancialManagement user mainMenu
    mainMenu updatedUser

  "3" -> do
    updatedUser <- displayJournalManagement user mainMenu
    mainMenu updatedUser

  "4" -> do
    putStrLn "See you again!"
    exitSuccess

  _ -> do
    putStrLn "Invalid choice. Try again."
    mainMenu user
```

mainMenu function uses a case statement to handle the user's choice. It calls the corresponding functions based on user's choice.

To-do List Module (TodoList.hs):

```
addTodoTask :: User -> IO User
addTodoTask user = do
  putStrLn "Please enter the title of the new to-do task: "
  todoTitle <- getLine
  putStrLn "Please enter the description of the new to-do task: "
  todoDescription <- getLine
  let newTodo = TodoTask todoTitle todoDescription ""
  putStrLn "To-do Task added successfully!"
  return user { todos = newTodo : todos user }
```

addTodoTask function takes a 'User', prompts the user for the title and description for the new to-do task, creates a new 'Todo' object, and returns with the new added to-do task.

```
updateTodoStatus :: User -> String -> IO User
updateTodoStatus user title = do
  case find (\todo -> todoTitle todo == title) (todos user) of
    Just _ -> do
      putStrLn "Please enter the status of the task (eg: Completed): "
      newStatus <- getLine
      let updatedUser = updateUserStatus newStatus
      return updatedUser

    Nothing -> do
      putStrLn "To-do task not found. Please enter a valid task title: "
      reEnterTitle <- getLine
      updateTodoStatus user reEnterTitle

  where
    updateUserStatus newStatus = user { todos = updatedTodos }
      where
        updatedTodos = map updateStatus (todos user)
        updateStatus todo
          | todoTitle todo == title = todo { todoStatus = newStatus }
          | otherwise = todo
```

updateTodoStatus is a function that allows user to input the status of the to-do task. The user can freely edit the task status (not limiting to complete or incomplete) by entering the title of the desired task to select it. If the specified task is not found, it prompts the user to enter a valid task title.

```
displayTodoListManagement :: User -> IO User
displayTodoListManagement user = do
  putStrLn ""
  putStrLn "------------------------------------------------"
  putStrLn "============= To-Do List Management =============="
  putStrLn ""
  mapM_ (\todo -> putStrLn $ "[" ++ todoStatus todo ++ "] " ++ "" ++ todoTitle todo ++
" | " ++ todoDescription todo) (todos user)

  putStrLn ""
  putStrLn "1. Add new to-do task"
  putStrLn "2. Edit to-do task status"
  putStrLn "3. Back to Main Menu"
  putStrLn ""
  putStrLn "================================================="
  putStrLn "-------------------------------------------------"
  putStrLn ""
  putStr "Please enter your choice: "
  todoChoice <- getLine
```

displayTodoList is a function to display the current to-do list along with the options to add a new to-do task, edit to-do task status, and go back to main menu. It interacts with the user to perform operations related to to-do list management.

```
case todoChoice of
  "1" -> do
    updatedUser <- addTodoTask user
    displayTodoListManagement updatedUser

  "2" -> do
    putStrLn "Please enter the title of the to-do task that you want to edit: "
    todoTitleToEdit <- getLine
    updatedUser <- updateTodoStatus user todoTitleToEdit
    putStrLn "To-do task status updated successfully!"
    displayTodoListManagement updatedUser

  "3" -> return user

  _ -> do
    putStrLn "Invalid choice. Try again."
    displayTodoListManagement user
```

displayTodoList function uses a case statement to handle the user's choice. It calls the corresponding functions based on user's choice.

Financial Module (Financial.hs):

```
getTransactionType :: IO TransactionType
getTransactionType = do
  putStrLn "Please enter the transaction type (Income/Expense): "
  transactionTypeStr <- getLine
  let lowerTransactionType = map toLower transactionTypeStr
  case lowerTransactionType of
    "income" -> return Income
    "expense" -> return Expense
    _ -> do
      putStrLn "Invalid transaction type. Try again."
      getTransactionType
```

getTransactionType is a function to prompt user to enter the type of transaction. It ensures that the user enters a valid transaction type, which is either income or expense. ‘toLower’ is used for case-insensitive comparison. If the user enters any value other than ‘income’ or ‘expense’, it prompts the user to enter again.

```
getAmount :: IO Double
getAmount = do
  putStrLn "Please enter the amount in numeric: "
  amountStr <- getLine
  case reads amountStr of
    [(amount, "")] -> return amount
    _ -> do
      putStrLn "Invalid amount. Try again."
      getAmount
```

getAmount is designed to retrieve a valid numeric amount from the user. ‘reads’ has been utilized here to parse the input as a Double. The function repeats until a valid amount is provided.

```
recordFinancialTransaction :: User -> (User -> IO User) -> IO User
recordFinancialTransaction user menuFunction= do
  transactionType <- getTransactionType
  putStrLn "Please enter the description of the transaction: "
  description <- getLine
  amount <- getAmount
  let updatedUser = addEntry user (FinancialTransaction transactionType
description amount)
  putStrLn ""
  putStrLn "Financial transaction recorded successfully!"
  displayFinancialManagement updatedUser menuFunction
```

recordFinancialTransaction function allows the user to record a new financial transaction. It first calls 'getTransactionType' to obtain the transaction type and calls 'getAmount' to get the amount of the transaction. The financial transaction is added to the user's entries using 'addEntry' function. It also calls 'displayFinancialManagement' to provide further options.

```
displayFinancialSummary :: User -> (User -> IO User) -> IO User
displayFinancialSummary user menuFunction = do
  let allTransactions = entries user
      income = sum [ amount | FinancialTransaction Income _ amount <- allTransactions ]
      expense = sum [ amount | FinancialTransaction Expense _ amount <- allTransactions ]
      remainingBudget = income - expense

  putStrLn ""
  putStrLn "------------------------------------------------"
  putStrLn "=============== Financial Summary ================"
  putStrLn ""
  mapM_ displayTransactionDetails allTransactions
  putStrLn ""
  putStrLn "------------------------------------------------"
  putStrLn ""
  putStrLn $ "Total Incomes: " ++ show income
  putStrLn $ "Total Expenses: " ++ show expense
  putStrLn $ "Remaining Budget: " ++ show remainingBudget
  putStrLn ""
  putStrLn "------------------------------------------------"
  putStrLn ""
  putStrLn "1. Back to Financial Management"
  putStrLn "2. Back to Main Menu"
  putStrLn ""
  putStrLn "================================================"
  putStrLn "------------------------------------------------"
  putStrLn ""
  putStrLn "Please enter your choice: "
  financialGoBackChoice <- getLine
```

displayFinancialSummary is designed to display a summary of the user's transactions. It calculates the total incomes, total expenses, and remaining budget by filtering and summing the amounts. This helps user to have a better understanding of their financial state. Besides that, it also allows users to navigate back to financial management or main menu.

```
case financialGoBackChoice of
  "1" -> displayFinancialManagement user menuFunction

  "2" -> menuFunction user

  _ -> do
    putStrLn "Invalid choice. Try again."
    displayFinancialSummary user menuFunction
```

displayTodoList function uses a case statement to handle the user's choice. It calls the corresponding functions based on user's choice.

```
displayFinancialManagement :: User -> (User -> IO User) -> IO User
displayFinancialManagement user menuFunction = do
  putStrLn ""
  putStrLn "------------------------------------------------"
  putStrLn "============== Financial Management =============="
  putStrLn ""
  putStrLn "1. Record Financial Transaction"
  putStrLn "2. View Financial Summary"
  putStrLn "3. Back to Main Menu"
  putStrLn ""
  putStrLn "================================================"
  putStrLn "------------------------------------------------"
  putStrLn ""
  putStrLn "Please enter your choice: "
  financialChoice <- getLine
```

Above function displays the menu of the financial management. It prompts the user with options. User may choose to record a financial transaction, view financial summary, or navigate back to the main menu.

```haskell
case financialChoice of
  "1" -> do
    updatedUser <- recordFinancialTransaction user menuFunction
    displayFinancialManagement updatedUser menuFunction

  "2" -> do
    displayFinancialSummary user menuFunction

  "3" -> menuFunction user

  _ -> do
    putStrLn "Invalid choice. Please try again. "
    displayFinancialManagement user menuFunction
```

displayFinancialManagement function uses a case statement to handle the user's choice. It calls the corresponding functions based on user's choice.

Journal Module (Journal.hs):

```
recordJournalEntry :: User -> (User -> IO User) -> IO User
recordJournalEntry user actionFunction = do
  putStrLn "Please enter the date of the journal entry: "
  journalDate <- getLine
  putStrLn "Please enter your journal:"
  journalText <- getLine
  let updatedUser = addEntry user (JournalEntry journalDate journalText)
  putStrLn ""
  putStrLn "Journal added successfully!"
  displayAllJournals updatedUser actionFunction
```

recordJournalEntry functions allows the user to record a new journal entry by inputting the date and journal text. 'addEntry' function is called to add the journal entry to the user's entries. After journal is added successfully, it calls 'displayAllJournals' to provide further options.

```
displayJournalEntryDetails :: User -> IO ()
displayJournalEntryDetails user = do
  let journalEntries = [entry | entry@(JournalEntry _ _) <- entries user]
  mapM_ printEntry journalEntries
  where
    printEntry :: Entry -> IO ()
    printEntry entry =
      case entry of
        JournalEntry date text -> printf "%s: %s\n" date text
        _ -> return ()
```

The above function displays the details of journal entries. It extracts and prints details of each journal entry in the user's entries.

```haskell
deleteJournalEntry :: User -> IO User
deleteJournalEntry user = do
  putStrLn ""
  putStrLn "-------------------------------------------------"
  putStrLn "================ Delete a Journal ================"
  putStrLn ""
  displayJournalEntryDetails user
  putStrLn ""
  putStrLn "================================================="
  putStrLn "-------------------------------------------------"
  putStrLn ""
  putStrLn "Please enter the date of the journal you want to delete: "
  journalDateToDelete <- getLine

  let updatedEntries = filter (\case
                                JournalEntry date _ -> date /= journalDateToDelete
                                _ -> True
                              ) (entries user)

  case find (\case
              JournalEntry date _ -> date == journalDateToDelete
              _ -> False
            ) (entries user) of
    Just _ -> do
      putStrLn "Journal has been deleted successfully!"
      putStrLn ""
      return user { entries = updatedEntries }
    Nothing -> do
      putStrLn "Journal not found. Try again. "
      deleteJournalEntry user
```

deleteJournalEntry allows user to delete a journal entry. It will first call ‘displayJournalEntryDetails’ to display the existing journal entries for the user. Then, it prompts the user to enter the date of the journal entry to delete. It uses ‘filter’ to remove the specified task.

```haskell
displayAllJournals :: User -> (User -> IO User) -> IO User
displayAllJournals user actionFunction = do
  putStrLn ""
  putStrLn "------------------------------------------------"
  putStrLn "================= View Journal ================="
  putStrLn ""
  displayJournalEntryDetails user
  putStrLn ""
  putStrLn "================================================"
  putStrLn ""
  putStrLn "1. Delete a journal"
  putStrLn "2. Back to Journal Management"
  putStrLn "3. Back to Main Menu"
  putStrLn ""
  putStrLn "================================================"
  putStrLn "------------------------------------------------"
  putStrLn ""
  putStrLn "Please enter your choice: "
  journalChoice <- getLine
```

displayAllJournals is a function that displays all journal entries and provides option to delete the entries. First, it will call 'displayJournalEntryDetails' to show details of all journal entries. Then, it prompts the user to choose between deleting a journal entry, returning to journal management, or returning to the main menu.

```haskell
case journalChoice of
  "1" -> do
    updatedUser <- deleteJournalEntry user
    displayAllJournals updatedUser actionFunction

  "2" -> displayJournalManagement user actionFunction

  "3" -> actionFunction user

  _ -> do
    putStrLn "Invalid choice. Try again."
    displayAllJournals user actionFunction
```

displayAllJournals function uses a case statement to handle the user's choice. It calls the corresponding functions based on user's choice.

```haskell
displayJournalManagement :: User -> (User -> IO User) -> IO User
displayJournalManagement user actionFunction= do
  putStrLn ""
  putStrLn "------------------------------------------------"
  putStrLn "=============== Journal Management ==============="
  putStrLn ""
  putStrLn "1. Write a journal"
  putStrLn "2. View Journal"
  putStrLn "3. Back to Main Menu"
  putStrLn ""
  putStrLn "================================================"
  putStrLn "------------------------------------------------"
  putStrLn ""
  putStrLn "Please enter your choice: "
  journalChoice <- getLine
```

displayJournalManagement has same concept as 'displayTodoListManagement' and 'displayFinancialManagement'. It displays the menu of journal management, prompts user to write a journal, view journal or return to main menu.

```haskell
case journalChoice of
  "1" -> do
    updatedUser <- recordJournalEntry user actionFunction
    displayJournalManagement updatedUser actionFunction

  "2" -> do
    updatedUser <- displayAllJournals user actionFunction
    displayJournalManagement updatedUser actionFunction

  "3" -> actionFunction user

  _ -> do
    putStrLn "Invalid Choice. Try again."
    return user
```

displayJournalManagement function uses a case statement to handle the user's choice. It calls the corresponding functions based on user's choice.

Personal Reflection

In designing RoutineSync, I have applied some functional concepts to it. I have used a lot of ‘do’ blocks and ‘let’ statements. For example, in ‘mainMenu’ function, I've used 'do' blocks to organize the steps of the program. Each line in the 'do' block represents a different action. Besides that, ‘let’ statement is used in ‘updateTodoStatus’ function. The let statement is implicit in the assignment let updatedUser = updateUserStatus newStatus. It creates a local binding (updatedUser) for the result of the updateUserStatus function. I have demonstrated the use of parametric polymorphism in the ‘addEntry’ function. It works for any type that is an instance of the ‘User’ and ‘Entry’ type, which allows me to add both ‘JournalEntry’ and ‘FinancialTransaction’ types to the user’s entries. Last but not least, I have organized the entire code by grouping related functions together into modules. This helps in increasing my code’s readability, modularity and so on.

The biggest challenge that I have met doing this final project was the lack of modularization at first. Initially, I did not implement many functions because I wrote the entire code in few functions. I realized that this was not a good practice of functional programming when I almost finished with the code. Then, I spent half day on breaking them into smaller functions. However, I encountered with another challenge when I proceeded to organize them into modules. This is because the incorrect import of modules caused cyclic module dependencies. I have spent nearly one day to figure out the solution for it because I could not find many examples on the Internet that I can refer to. After that, I found out that someone used higher order function to resolve similar problems which gives me an idea on how to solve it. However, since there is not much information, I did a lot of trial-and-error to finally resolve it.

Besides using effective functional programming concepts in my program, user-friendly design is another strength of it. In RoutineSync, clear instructions are given to user at different stages. Additionally, I also focused on the sequential flow of the program. For example, after recording a journal entry, the user is directed to view all journals. All possible actions that will took by user has been considered. When the user deleted a journal successfully, he or she can choose whether to go back to the journal management interface or main menu which allows he or she to manage their journal without having to navigate through unrelated sections and cater to their preferences.

A weakness in my program is that it lacks enough data validation for user inputs. Although the program has implemented some basic data validation, there is still room for improvement in ensuring the reliability of user input such as consider strengthening the validity validation of date formats for the 'recordJournalEntry' function. In addition, for the 'displayJournalSummary' function, introducing a more attractive interface design such as using tables to display the data will help ensure that information is presented in a more concise and clear way. Finally, considering RoutineSync as a personal management tool, I think adding more features to it (recording workout routine etc.) can improve the program's comprehensiveness and usefulness.

References

Noonan, M. (2018, January 19). *Blog*. Storm-Country.com. https://country.com/blog/LambdaCase

6.2.13. Lambda-case — Glasgow Haskell Compiler 9.8.1 User's Guide. (n.d.). Downloads.haskell.org. Retrieved December 23, 2023, from https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/lambda_case.html

Replit Link: https://replit.com/@prg2214fp082023/Final-Assignment-tzeyea21#Main.hs