程式語言設計期末專題報告

組員：**林柏廷、蔡淵丞**

1. 程式碼概述

1.1 如何運作

- 本次專題要求實做一個共享工作表，並使用命令列介面操作。
- 程式使用Scala撰寫
- 使用Functional Programming
- 在5, 6的功能中使用**策略模式**

因為使用functional porgramming設計，所以主程式是一個`mainMenu()`函式。我們主要透過讀取使用者輸入更新`Users`跟`Sheets`，從而遞迴呼叫`mainMenu()`來達成程式的運作。

以下為一些實作上的細節限制：

- 工作表由一個`Map[String, Sheet]`統一存放，因此同個名稱的工作表只會有一個。本來嘗試讓不同用戶可以擁有同名稱的工作表，然而這樣的實作方式不利於函數式程式設計以及策略模式的呈現。
- 工作表的權限對所有能存取到它的用戶都是相同的，並且只有擁有者能設定。（此為我對作業要求的理解）
- 更改數值時只支援二元運算。（因為scala沒有eval）

1.2 定義的資料結構

- `User`：表示用戶，包含用戶名。在此實作中管理資訊統一由`Sheet`存放。

```
case class User(name: String)
```

- `Sheet`：表示工作表，包含名稱、數據、訪問權限、擁有者及協作者。

```
case class Sheet(name: String, data: Array[Array[Double]], accessRight:
String, owner: String, collaborators: Set[String])
```

- `Users`：用戶的映射，鍵為用戶名，值為`User`物件。

```
type Users = Map[String, User]
```

- `Sheets`：工作表的映射，鍵為工作表名，值為`Sheet`物件。

```
type Sheets = Map[String, Sheet]
```

1.3 功能開關

策略模式讓共享及權限的功能可以被輕易地開關。實作方式為讓各個函數接受`AccessControl`及`SharingControl`，根據具體實現的不同會使用不同的策略。

舉例：若是要開啟權限功能則將`accessControl`設為`EnableAccessControl`策略，關閉則設為`NoAccessControl`策略。

- 權限控制：`changeValue`會使用`accessControl.editable(sheet)`來檢查工作表是否可編輯。而在執行`changeAccessRight`時會使用`accessControl.accessEnabled`來確認功能是否開啟。

```
trait AccessControl {
  def accessEnabled: Boolean
  def editable(sheet: Sheet): Boolean
}

object NoAccessControl extends AccessControl {
  override def accessEnabled: Boolean = false
  override def editable(sheet: Sheet): Boolean = true
}

object EnableAccessControl extends AccessControl {
  override def accessEnabled: Boolean = true
  override def editable(sheet: Sheet): Boolean =
    sheet.accessRight == "editable"
}
```

- 共享控制：在確認用戶是否能存取某個工作表時會使用`sharingControl.hasAccess(sheet, userName)`。若有開啟共享功能則會檢`collaborators`，若無則不會。而在執行`shareSheet`時會使用`sharingControl.sharingEnabled`來確認功能是否開啟。

```
trait SharingControl {
  def sharingEnabled: Boolean
  def hasAccess(sheet: Sheet, userName: String): Boolean
}

object NoSharingControl extends SharingControl {
  override def sharingEnabled: Boolean = false
  override def hasAccess(sheet: Sheet, userName: String): Boolean =
sheet.owner == userName
}

object EnableSharingControl extends SharingControl {
  override def sharingEnabled: Boolean = true
  override def hasAccess(sheet: Sheet, userName: String): Boolean =
    sheet.owner == userName || sheet.collaborators.contains(userName)
}
```

2. 使用的範式

2.1 函數式程式設計

函數式程式設計強調使用純函數和不可變資料結構來避免副作用，使程式碼更具可預測性和可測試性。主要特點包括：

- **不變性**：不可變資料結構可以避免資料在多執行緒環境下被意外修改，確保資料的一致性和安全性。
- **純函數**：所有函數均為純函數，即不修改輸入參數，返回新值。純函數保證相同的輸入會產生相同的輸出，無副作用，使程式碼更容易理解和測試。
- **高階函數**：函數可以作為變數傳遞和回傳，使程式碼更靈活和可重用。
- **模式匹配**：模式匹配用於處理不同的分支情況，提高程式碼的可讀性和可維護性。

2.2 程式碼中的應用

- **不變性**：在程式碼中，所有資料結構均為不可變。例如，`Users`和`Sheets`均使用不可變的`Map`。

```
type Users = Map[String, User]
type Sheets = Map[String, Sheet]
```

- **純函數**：所有函數均為純函數。例如，`mainMenu`使用遞迴並且每次都創建新的`users`及`sheets`映射，`createUser`函數接收當前的用戶映射和一個用戶名，返回一個新的用戶映射或錯誤訊息。

```
def createUser(users: Users, name: String): Either[String, Users] = {
  if (users.contains(name)) {
    Left(s"User '$name' already exists.")
  } else {
    Right(users + (name -> User(name)))
  }
}
```

- **模式匹配**：在處理`Option`和`Either`等封裝類型時，使用模式匹配來簡化分支邏輯。例如，在`checkSheet`函數中使用模式匹配來處理各種錯誤。

```
def checkSheet(users: Users, sheets: Sheets, sheetName: String, userName:
String, accessControl: AccessControl, sharingControl: SharingControl):
Either[String, Array[Array[Double]]] = {
  if (!users.contains(userName)) {
    return Left(s"User '$userName' does not exist.")
  }
  sheets.get(sheetName).map { sheet =>
    if (sharingControl.hasAccess(sheet, userName)) {
      Right(sheet.data)
    } else {
      Left("This sheet is not accessible.")
    }
```

```
    }.getOrElse(Left("Sheet not found."))
  }
```

3. 未使用的範式：物件導向

3.1 特點

- **封裝**：將數據和操作這些數據的方法封裝在一個類中，從而保護數據的完整性，防止外部程式碼直接訪問和修改內部狀態。
- **繼承**：通過從現有的類創建新類來共享和擴展行為，這樣可以避免重複代碼並促進代碼的重用。
- **多型**：允許不同的類型通過相同的接口進行互動，實現了基於對象的動態行為處理。

3.2 使用上的區別

如果我們使用物件導向（OOP）：

- **封裝：**`Sheet` 和 `User` 的狀態和行為將被封裝在各自的類中。創建用戶或工作表、修改值和更改權限的方法會是 `Sheet` 或 `User` 類的一部分，而不是作為獨立的函數。這種封裝有助於保護數據完整性，並提高程式碼的可讀性和維護性。
- **多型**：定義公共接口或抽象類別，並通過這些接口來操作不同類型的工作表或用戶。多型允許不同類別的物件通過相同的接口進行互動，讓系統在運行時能夠靈活地處理不同的對象。像是`Sheet`可以有不同子類別`AccessControlSheet`或`CollaborativceSheet`。

在純函數式程式設計的系統中，我們使用不可變資料結構和純函數來管理狀態和行為。這樣的設計避免了副作用，並提高了系統的可預測性和可測試性。 相較之下，物件導向的優勢在於：

- 更模組化：OOP通過類別和物件封裝數據和行為，使系統結構更加清晰。
- 程式碼的重複使用：繼承和多型許我們在不同類之間共享和重用代碼，減少重複代碼的出現。
- 動態行為：多型讓我們可以在運行時靈活地處理不同物件，增加了系統的靈活性。然而，OOP也可能帶來一些缺點，例如物件狀態的變化可能難以追蹤，繼承層次過深可能讓程式碼難以理解和維護。相對來說，純函數式程式設計則更強調數據的不可變性和函數的純粹性，使得系統更加穩定和可預測。

4. 使用的慣用法或設計模式

4.1 策略模式

策略模式定義一系列演算法，並將每個演算法封裝起來，使它們可以互相替換，策略模式讓演算法可以獨立於使用它的客戶而變化。

4.2 程式碼中的應用

- 訪問控制和共享控制：
    - `AccessControl`特徵及其具體實現`NoAccessControl`和`EnableAccessControl`。
    - `SharingControl`特徵及其具體實現`NoSharingControl`和`EnableSharingControl`。

  通過在不同情況下選擇不同的策略來控制工作表的訪問權限和共享權限。例如，在`checkSheet`和`changeValue`函數中，根據不同的訪問控制和共享控制策略來決定用戶是否可以查看或編輯工作表。詳情與 **1.3** 相同。

4.3 使用`Either`進行錯誤處理

在本程式中廣泛使用了 `Either` 來進行錯誤處理。這是一種常見的函數式程式設計慣用法，用於取代例外處理，使得錯誤處理更加顯式和安全。

`Either` 用於表示兩種可能的結果：成功或失敗。`Either` 有兩個子類型：

- `Right(value)`：表示成功並包含結果值。
- `Left(error)`：表示失敗並包含錯誤訊息。

程式碼中的應用請參考**2.2**

5. 未使用的慣用法或設計模式

5.1 裝飾者模式

裝飾者模式（**Decorator Pattern**）是一種結構型設計模式，它允許向一個物件動態添加方法，而不改變其原有的結構。這種模式通過創建一個裝飾器類來包裝原始物件，並在包裝類中添加新的功能。這樣，可以在運行時根據需要擴展物件的行為，而不會影響其他物件。

我們原本想要用這個設計模式，因為需要動態添加減少權限跟共享的功能很符合裝飾者的感覺。然而因為他們不會只有共用的接口，會有像是更改權限或分享等行為，經過測試後發現不太可行。

5.2 在程式碼中的區別

若使用裝飾者模式`Sheet`類別會有`AccessControlDecorator`及`ShareControlDecorator`，兩者在實現`checkValue`及`changeValue`等方法的時候會有不同實作方式，例如被`AccessControlDecorator`裝飾的話`changeValue`會先檢查權限。不過我認為這個比較適合物件導向的做法。

6. 編程過程中遇到的錯誤

6.1 發生了什麼

在測試不同用戶跟工作表的互動時遇到了各種不如預期的結果。

6.2 如何使用除錯工具解決

我們沒有使用到除錯工具，而是使用單元測試來確認各個函數有辦法正確的執行任務。