Report of Solitaire Game with Haskell

Student Name: [redacted]
Registration Number: [redacted]
Username: [redacted]
Module Code: COM2108
Module Name: Functional Programming
Assignment Description: Grading Assessment

Part 1

Step1

It's a simple but fundamental step for designing this game. Both Suit and Pip can be implemented as sum types, and both can be enumerated. Card is the tuple of Suit and Pip, and Deck is a list of Card.

```
data Suit = Spades | Clubs | Diamonds | Hearts deriving (Eq, Ord, Enum, Bounded, Show)
data Pip = Ace | Two | Three | Four | Five | Six | Seven | Eight | Nine | Ten | Jack | Queen | King deriving
(Eq, Ord, Enum, Bounded, Show)
type Card = (Pip, Suit)
type Deck = [Card]
```

Step2

It's also the simple step. Haskell unique pattern enumeration statistics can be used to achieve the goal of this step easily. Pack is the product of a Pip list that continuously grows from Ace to King, and a list from Spades to Hearts. sCard and pCard can be completed by using Enum type of succ and pred to match the next or last one of the lists respectively. isAce and isKing use pattern matching with Eq.

```
pack :: Deck
pack = [(p, s)| p <- [Ace ..], s<-[Spades ..] ]

sCard :: Card -> Card
sCard (King, s) = (Ace, s)
sCard (p, s) = (succ p, s)

pCard :: Card -> Card
pCard (Ace, s) = (King, s)
pCard (p, s) = (pred p, s)

isAce :: Card -> Bool
isAce = (==) Ace . fst

isKing :: Card -> Bool
isKing = (==) King . fst
```

Shuffle needs a seed to generate a random sequence and then zip it with Pack and do a sort.

```
shuffle :: Deck -> Int -> Deck
shuffle d seed = map fst s
  where
        r = take (length d) (randoms (mkStdGen seed)  :: [Int])
        z = zip d r
        s = sortBy (\(_, n1) (_, n2)-> compare n1 n2) z
```

Step3

The Board should be defined with data since the type of Board will have two constructors in the next step, which are EOBard, and SBoard. At the same time, the type of Board should be implemented as an instance of Show to display the Board. We only need to implement additional display of Columns, and splice the String.

```
showColumns :: Columns -> String
showColumns = unlines . map ((++) " " . show)
```

Step4

- eODeal Function
  - Shuffle, we implemented above, is used in this function. The general steps for creating a new layout are as following:
    - Receive a seed
    - Shuffle the cards with seeds
    - Distribute cards according to the rules of free Solitaire
  - Recursion is used to implement the function that divides the cards into 6 cards in each column.

```
oDealA :: Deck -> [Deck]
oDealA [] = []
oDealA d  = take 6 d : oDealA (drop 6 d)
```

- toFoundations
  - Some design is required to use in the toFoundations. Analyse the situation firstly which cards can be placed in foundations.
    - Ace is in the first card of columns.
    - Ace is in reserve.
    - The first card of columns has a successor of foundation.
    - There is a successor to foundation in the reserve.
  - Only four situations should be implemented into one function that received the board and returned the board after the corresponding operation is performed, and judges whether the board is the same before or after the operation. In the case of “same”, the situation does not hold in the current situation; and in the opposite situation means the operation is successful, then recursively call toFoundations.

```
toFoundations :: Board -> Board
toFoundations SBoard {} = undefined
toFoundations eob
  | eob /= eob' = toFoundations eob'
  | eob /= eob'' = toFoundations eob''
  | eob /= eob''' = toFoundations eob'''
  | eob /= eob'''' = toFoundations eob''''
  | otherwise = eob
  where
      eob' = moveAceFromColToFoundations eob
      eob'' = moveAceFromRsvToFoundations eob
      eob''' = moveCardFromColsToFoundations eob
      eob'''' = moveCardFromRsvToFoundations eob
```

Step5

It is better to add a new card type, to cope with the Spider without changing the original implementation of EOBoard, instead of modifying the original card type. The corresponding types of Spider games are as followings:

```
type FaceUp = Bool
type FaceCard = (Card, FaceUp)
type SpiderCols = [[FaceCard]]
data Board = EOBoard Foundation Columns Reserve | SBoard Foundation SpiderCols Stock deriving (Eq)
```

Part 2

Step1

This step is similar with eODeal. There are four possible movements for the card.

1. From the reserve to foundation
2. From column to column
3. From column to foundation
4. From column to reserve

Analyse these four situations as functions.

Step2

The most valuable among all the possible moves found above should be selected, so a function is needed to evaluate the situation. The valuation considerations are as following:

1. Cells should be used as many times as possible.
2. If the reserve is less than a last resort, do not use it. If there are too many reserves, players will be punished.
3. King as the only card that in the column is a situation that players happy to see.
4. The more card scores players can get in this step, the better.

```
boardWeighted :: Board -> Int
boardWeighted (EOBoard f c r)= notEmptyCol + cardValues + rsvPenalty + colOnlyHaveKing
    where
        notEmptyCol = length $ chooseNotNull c
        cardValues = 100 * getCardValues f
        rsvPenalty = -20 * length r
        colOnlyHaveKing = 10*length (filter isKing (map head (filter ((==1). length) c)))
boardWeighted _ = undefined
```

Step3

This step is relatively simple, haveWon only needs to determine whether the columns and reservations are all empty. playSolitaire uses the incoming seed to generate the game first, and then chooseMove until victory or nowhere to go.

Step4

Use the seed to generate random sequences, and then use these random sequences to playSolitaire and count the scores.