CS583 Project Report - Fan Tan card games

Han Lin

This is a “Fan Tan” or “Sevens” poker card game. It will have four players in the game. Then each player will be given the same numbers of card. The player with the diamond seven will have to be first and put it in the middle. The one who first give out all of cards will be the winner.

There are other poker games in Haskell or in other languages. The user of this project will be people who want to play this kind of card game that is a text-based type of program.

The game will be a text-based type. Users will type some commands which are defined. My idea is to have them type the command one at a time until is their turn.

Some objects are presented as follow:

1. *Card Value:*
   Data type, Ranks, which include Ace, Two, Three, …, King.
   Data type, Suits, which include Hearts, Diamonds, Spades and Clubs.
   Data constructor, PlayingCards, which has two fields, Ranks and Suits.

2. *Cards on hand*
   Type synonym PlayerCardsHold: A list of PlayingCards that each player has on hand.

3. *Cards that given out*
   Type synonym CenterCards: A list of PlayingCards that the center has.

4. *Game State*
   A nested data constructor which act as a global state to keep track of the entire game. Its field types are as follow

   ```
   GameState {
     player1 :: PlayerCardsHold,
     player2 :: PlayerCardsHold,
     player3 :: PlayerCardsHold,
     player4 :: PlayerCardsHold,
     center :: CenterCards
   }
   ```

Some functions are presented as follow:

1. *allCards :: [PlayingCards]*
   A function that will display all cards in a deck

2. *showCards :: [PlayingCards] -> String*
   A pretty printer that will do the pretty printing of a list of PlayingCards. For example, it will pretty print [PlayingCards Ace Hearts, PlayingCards Seven Diamonds] as "Ace_Hearts Seven_Diamonds ".

3. *play x :: PlayingCards -> State GameState [PlayingCards]*
   A state function that will do the play card action. For instance, "play (PlayingCards Ace Hearts)" will return the center card that center = PlayingCards Ace Hearts

4. *checkCards n :: Int -> State GameState [PlayingCards]*
   A state function that will do the check card action with case n =1 check player1's card, n = 2 check player2's card, n = 3 check player3's card, n = 4 check player4's card, n = 5 check center card.

5. *start :: State GameState [PlayingCards]*
   A state function that will do the start game action and return the center card which it will play the Seven_Diamonds card on the center according to the rule.

6. *dealHand :: Int -> Int -> [PlayingCards]*
   A function that will deal out a deck to players. First it will generate a random list of PlayingCards and check the two Int variables to determine the range in the random list. For instance, dealHand 0 12 will deal out from index 0 to index 12 in that list of PlayingCards to the player. It is possible to shuffle and get a different deck of cards contents by changing mkStdGen in the function.

7. *showGameState :: GameState -> String*
   A pretty printer that will do the pretty printing of a GameState. For example, it will pretty print *GameState {player1 = PlayingCards Ace Hearts, player2 = PlayingCards Ace Diamonds, player3 = PlayingCards Two Hearts , player4 = PlayingCards Two Diamonds, center = PlayingCards Seven Diamonds }* as

   Player 1 has: Ace_Hearts
   Player 2 has: Ace_Diamonds
   Player 3 has: Two_Hearts
   Player 4 has: Two_Diamonds
   Center is: Seven_Diamonds

Decision I made during my project:

1. Remove the rolling dice. As I working on my project, I failed to see the real effect of my rolling dice implementation. Maybe I couldn't come up with a good implementation of the rolling dice action, so I decided to not to include it. Here is a snippet of it

```
-- -- | Roll the dice to determine the sequence of players
-- rollDice :: State GameState Int
-- rollDice = do
--   currentState <- get
--   let gen = generator currentState
--   let (d1, gen') = randomR (1, 4) gen
--   put (currentState { generator = gen' } )
--   return d1
```

2. Apply state monad to the game. In the beginning as I was working on my project, at that time I had not yet to start my play action and game state implementation and I learned the state monad from the class. So I decided to apply the state monad to the game, and it worked perfectly. At my project milestone 2, I misunderstood the concept of the state monad, so I give a wrong implementation of the player state and some function state that were wrong too. Then I finally implement the global state GameState and combine with local state. Below are some old implementations that I implemented at that time.

```
-- checkStart :: State PlayerCardsHold PlayingCards
-- checkStart = state $ \(x:xs) -> if isIn (PlayingCards Seven Diamonds) (x:xs) == True
--                                 -- then (PlayingCards Seven Diamonds, x:xs)
--                                 then (PlayingCards Seven Diamonds, x:delete (PlayingCards
Seven Diamonds) xs)
--                                 else (x, x:xs)

-- play :: PlayingCards -> State PlayerCardsHold PlayingCards
-- play n = state $ \(x:xs) -> if isIn n (x:xs) == True
--                             then (n, x:delete n xs)
--                             else (x, x:xs)

-- player1State :: State PlayerCardsHold PlayingCards
-- player1State = do
--   dealHand 0 12
--   checkStart
--   play $ PlayingCards Four Diamonds
```

3. Change from each player control what cards they want to play according to the proposal to give a hard code version that define which card it will have to play next. For this decision, I couldn’t come up with a implementation to implement each player control what cards they want to play, so I just change to a hard code version.