1 UNO

During the lockdown, we had to endure way too many online game nights with friends and family. The frustration of loosing every single game of UNO sits deep in our bones. We thus decided to build our own UNO AI that wins every game without us even participating. We already started implementing this genius idea, but then got sidetracked by yet another glorious project, so we kindly ask you to help us finishing our UNO plans.

We will follow the rules as described on Wikipedia[1].

1. We already have a data type `Game` describing the state of a game (you can find it in `Type.hs`). This data type uses Haskell's record syntax about which you can learn more here. Most importantly, given a `game :: Game`, you can obtain an attribute `attr` of `game` using `attr game`.

   A game state is modeled using the following attributes:

   - `drawPile :: Pile` is a list of `Card`s represents the current draw pile.
   - `discardPile :: Pile` is a list of `Card`s represents the current discard pile.
   - `hands :: Hands` is a list of `Hand`s (that is a list of lists of `Card`s) representing the hands of all players from the first player (player `0`) to the last one.
   - `currentPlayer :: Int` is the index of the current player (i.e. the current player's hand is `hands !! currentPlayer`).
   - `direction :: Direction` is the direction of the game, either `Left` or `Right`.
   - `draw2stack :: Int` represents the number of successive `Draw2` action cards on top of the discard pile.

   We are looking for a way of printing the game state to the console. Your task is to implement a function `prettyShowGame :: Game -> String` that creates a pretty string of a given game state as exemplified by the following:

[1] https://en.wikipedia.org/wiki/Uno_(card_game)#Official_rules

```
>>> putStrLn $ prettyShowGame game
Last Played Card: G 4
Direction: ->
Draw2Stack: 0
---
> Player 0: [ G 1, B 5, Y 6, Y 5, Draw4, Y 7, G 1 ]
  Player 1: [ Y 8, G 9, R 4, G 8, B 7, R 8, Y 7 ]
  Player 2: [ R 9, B Draw2, R Skip, R 7, Y 4, G 5, R Skip ]
  Player 3: [ B 5, R 7, B Reverse, Wild, Y 2, B 7, G 4 ]
```

- `->` and `<-` are used to describe the directions `Right` and `Left`.
- `> Player <number>` marks the current player.

Hint: You can use `game = startWithSeed seed numberOfPlayers` with `seed=1` and `numberOfPlayers=4` to produce a game represented by above output. Use `show card` to stringify a card.

2. Now write a function `isGameOver :: Game -> Bool` that checks whether the game of UNO is over (i.e. the hand of at least one player is empty).

3. When simulating a game of UNO, we have to decide the next game direction after a player put down a card. Write a function `nextDirection :: Direction -> Card -> Direction` that given the current direction and played card returns the new direction. Use pattern matching on cards for this.

4. Write a function `isValidSuccessor :: Card -> Card -> Bool` that checks if the card passed as a second argument is a valid successor of the card passed as a first argument.

5. Write a function `isValidMove :: Game -> Move -> Bool` that, given a game state and a move, decides whether the move is valid. Use the function `isValidSuccessor` you just implemented. In particular, you should check that

   - the player chose a card on her hand,
   - the player is not obliged to play another card on her hand instead of the chosen card, and
   - the player does not draw from an empty pile.

6. We use the `Strategy` type to represent the playing strategies of players. Given the list of numbers of cards on each opponent's hand, the player's hand, the last played card, and the direction of play, a strategy decides the next move.

   When a player decides to draw a card, she provides another strategy – a draw strategy – that decides what to do with the drawn card. This

`DrawStrategy` returns a `DrawMove` – a restricted move that either puts the drawn card on the pile or keeps it on the player's hand.

Given the drawn card and the game state, we want to decide whether a given `DrawMove` is valid. To that end, implement the function `isValidDrawMove :: Card -> Game -> DrawMove -> Bool`.

7. The function `playMove :: Game -> Move -> Game` returns the next game state given a move. However, the case for `Draw2` cards is still missing. Implement this missing case.

8. Well done! We now have everything in place to play a first game of UNO! We already provide you with a simple strategy called `simpleStrategy` and an interactive strategy to play on your own called `interactiveStrategy` (you can find them in `InternalUtil.hs`). You can use the function `playAndPrint :: Seed -> [Strategy]` expecting a seed and a list of strategies – one for each player – to simulate a game while printing each move to the console. For example, to simulate a game with three players, run

   ```
   >>> playAndPrint 0 (replicate 3 simpleStrategy)
   ```

   We also provide you a function `playStrategies :: [Strategy] -> Int` expecting a list of strategies and an integer referring to the number of games that should be simulated. `playStrategies` will then print a summary of each players performance in all simulated games.

   Now here is your final exercise: Write your own strategy for UNO by implementing the functions `unoAI :: Strategy` and `unoDrawAI :: DrawStrategy`. You will have to play against a variety of strategies to pass this exercise so make sure your AI is playing smart.