Card_Game.hs - Haskell Card Game

Nikola Oljaca

October 12, 2024

Introduction

`Card_Game.hs` is a Haskell implementation of basic card game logic. It provides functionalities for manipulating cards, evaluating their values, performing game moves, and scoring the game. The game allows players to draw cards, discard them, and check conditions, such as whether all cards in hand are of the same color. The repository for this project is hosted at: `https://github.com/NOljaca`.

Dependencies

To compile and run the code, the following Haskell module is required:

- **Data.Char**: This module provides character operations, including:
  - `digitToInt`: Converts a character representing a digit to its corresponding integer.
  - `isDigit`: Checks if a character is a digit.
  - `toUpper`: Converts a character to uppercase.

Data Types

1. `Color`

The `Color` data type defines the possible colors for a card.

```
data Color = Red | Black deriving (Show, Eq)
```

2. `Suit`

The `Suit` data type represents the suit of a card.

```
data Suit = Clubs | Diamonds | Hearts | Spades deriving (Show, Eq)
```

3. `Rank`

The `Rank` data type represents the rank of a card, which can either be a number (2-10) or a face card (Jack, Queen, King, Ace).

```
data Rank = Num Int | Jack | Queen | King | Ace deriving (Show, Eq)
```

4. `Card`

The `Card` data type encapsulates both the `Suit` and the `Rank` of a card.

```
data Card = Card { suit :: Suit, rank :: Rank } deriving (Show, Eq)
```

5. `Move`

The `Move` data type represents possible game actions: `Draw` or `Discard` a `Card`.

```
data Move = Draw | Discard Card deriving (Show, Eq)
```

Core Functions

`cardColor :: Card -> Color`

This function determines the color of a card based on its suit.

```
cardColor (Card suit _) =
    if suit == Spades || suit == Clubs
    then Black
    else Red
```

`cardValue :: Rank -> Int`

This function returns the value of a card based on its rank. The value of face cards (Jack, Queen, King) is 10, and the Ace has a value of 11.

```
cardValue rank =
    case rank of
        Num n  -> n
        Jack   -> 10
        Queen  -> 10
        King   -> 10
        Ace    -> 11
        _      -> error "Bitte eine gültige Zahl verwenden"
```

`convertSuit :: Char -> Suit`

Converts a character (representing a suit) into a `Suit` type. For example, 'C' becomes `Clubs`.

```
convertSuit c =
    case toUpper c of
        'C' -> Clubs
        'D' -> Diamonds
        'H' -> Hearts
        'S' -> Spades
        _   -> error "Invalid suit"
```

`convertRank :: Char -> Rank`

Converts a character (representing a rank) into a `Rank` type. For example, 'Q' becomes `Queen`.

```
convertRank c =
    if isDigit c
    then Num (digitToInt c)
    else case toUpper c of
        'J' -> Jack
        'Q' -> Queen
        'K' -> King
        'A' -> Ace
        _   -> error "Invalid rank"
```

Game Logic

`addHeldCard :: [Card] -> Card -> [Card]`

This function adds a card to the player's hand.

```
addHeldCard xs card = xs ++ [card]
```

`removeCard :: [Card] -> Card -> [Card]`

Removes a card from the player's hand.

```
removeCard [] _ = []
removeCard (x:xs) c
    | c == x    = xs
    | otherwise = x : removeCard xs c
```

Running the Game

To simulate the card game, the function `rrunGame` takes a deck of cards, a sequence of moves, and the goal score. It recursively processes each move, updating the game state.

```
rrunGame :: [Card] -> [Move] -> Int -> Int
rrunGame (y:ys) (x:xs) goal = rrunGame' (y:ys) (x:xs) goal []
```

`rrunGame'` is the helper function that handles the core logic of processing moves and calculating scores.

Example Cards and Moves

`Card` examples:

```
card1 = Card { suit = Spades, rank = Num 4 }
card2 = Card { suit = Hearts, rank = Queen }
card3 = Card { suit = Diamonds, rank = Num 7 }
card4 = Card { suit = Clubs, rank = Ace }
```

Example moves:

```
move1 = Draw
move = [move1]
deck = [card3, card4, card3]
```