Custom Solitaire

This problem involves a solitaire card game invented just for this exercise.[1] You will write a program that tracks the progress of a game.

A game is played with a *card-list* and a *goal*. The player has a list of *held-cards*, initially empty. The player makes a move by either *drawing*, which means removing the first card in the card-list from the card-list and adding it to the held-cards, or *discarding*, which means choosing one of the held-cards to remove. The game ends either when the player chooses to make no more moves or when the sum of the values of the held-cards is greater than the goal.

The objective is to end the game with a low score (0 is best). Scoring works as follows: Let *sum* be the sum of the values of the held-cards. If *sum* is greater than *goal*, the *preliminary score* is three times *sum* − *goal*, else the preliminary score is *goal* − *sum*. The score is the preliminary score unless all the held-cards are the same color, in which case the score is the preliminary score divided by 2 (and rounded down as usual with integer division).

The types are defined as follows (you will probably want to derive from `Eq` and `Show`):

```
data Color = Red | Black

data Suit = Clubs | Diamonds | Hearts | Spades

data Rank = Num Int | Jack | Queen | King | Ace

data Card = Card { suit :: Suit, rank :: Rank }

data Move = Draw | Discard Card
```

1. Write a function `cardColor`, which takes a card and returns its color (spades and clubs are black, diamonds and hearts are red).
2. Write a function `cardValue`, which takes a card and returns its value (numbered cards have their number as the value, aces are 11, everything else is 10).
3. Write a function `removeCard`, which takes a list of cards `cs`, and a card *c*. It returns a list that has all the elements of `cs` except `c`. If `c` is in the list more than once, remove only the first one. If `c` is not in the list, raise an error.
4. Write a function `allSameColor`, which takes a list of cards and returns `True` if all the cards in the list are the same color and `False` otherwise.
5. Write a function `sumCards`, which takes a list of cards and returns the sum of their values. *Use a locally defined helper function that is tail recursive.*
6. Write a function `score`, which takes a card list (the held-cards) and an int (the goal) and computes the score as described above.
7. Define a type for representing the state of the game. (What items make up the state of the game?)
8. Write a function `runGame` that takes a *card list* (the card-list) a *move list* (what the player "does" at each point), and an *int* (the goal) and returns the score at the end of the game after processing (some or all of) the moves in the move list in order. Use a locally defined recursive helper function that takes the current state of the game as a parameter. As described above:
    - The game starts with the held-cards being the empty list.
    - The game ends if there are no more moves. (The player chose to stop since the *move list* is empty.)
    - If the player discards some card `c`, play continues (i.e., make a recursive call) with the held-cards not having `c` and the card-list unchanged. If `c` is not in the held-cards, raise an error.
    - If the player draws and the card-list is empty, the game is over. Else if drawing causes the sum of the held-cards to exceed the goal, the game is over. Else play continues with a larger held-cards and a smaller card-list.

1 This exercise is adapted from an online course by Prof. Dan Grossman.

In the second part, arrange the code to get the cards, the moves and the goal from the user.

9. Write a function `convertSuit` that takes a character c and returns the corresponding suit that starts with that letter. For example, if `c` is `'d'` or `'D'`, it should return `Diamonds`. If the suit is unknown, raise an error.
10. Write a function `convertRank` that takes a character c and returns the corresponding rank. For face cards, use the first letter, for "Ace" use '1' and for 10 use 't' (or 'T'). You can use the `isDigit` and `digitToInt` functions from the *Data.Char* module. If the rank is unknown, raise an error.
11. Write a function `convertCard` that takes a suit name (char) and a rank name (char), and returns a card.
12. Write a function `readCards` that will read (and return) a list of cards from the user. On each line, the user will type a card as two letters (such as "hq" for "Queen of Hearts" or "s2" for "Two of Spades"). The user will end the sequence by typing a single dot.
13. Write a function convertMove that takes a move name (char), a suit name (char), and a rank name (char) and returns a move. If the move name is `'d'` or `'D'` it should return `Draw` and ignore the other parameters. If the move is `'r'` or `'R'`, it should return `Discard c` where `c` is the card created from the suit name and the rank name.
14. Write a function `readMoves` that will read (and return) a list of moves from the user. On each line, the user will type a move as one or three letters (such as "d" for "Draw" or "rhq" for "Discard Queen of Hearts"). The user will end the sequence by typing a single dot.
15. Bring all of it together so that the following `main` function will work:

```
main = do putStrLn "Enter cards:"
          cards <- readCards
          -- putStrLn (show cards)

          putStrLn "Enter moves:"
          moves <- readMoves
          -- putStrLn (show moves)

          putStrLn "Enter goal:"
          line <- getLine

          let goal = read line :: Int

          let score = runGame cards moves goal
          putStrLn ("Score: " ++ show score)
```

A sample run of the program should look as follows:

```
Enter cards:
c1
s1
c1
s1
.
Enter moves:
d
d
d
d
d
.
Enter goal:
42
Score: 3
```

A sample run with an error:

```
Enter cards:
cj
s8
.
Enter moves:
d
```

```
rhj
.
Enter goal:
42
bonus2: card not in list
```

Name your module *Main.hs* and make sure that the following command creates an executable:

```
ghc Main.hs -o bonus2
```