Blackjack

Max Sundström, Samuel Bakken Viktorsson, William Strand Paurell

Contents

1 Introduction

For the project, the group have decided to make a blackjack game. The reasoning for choosing blackjack was first that everyone in the group wanted to make something fun and exciting, and so quickly it became clear that it had to be a game. The first idea was to make "4 in a row" but it was ultimately decided to not go with that idea, and then blackjack was suggested by one of the members and it all took off from there. We really wanted to make something that was responsive and not dull, in the sense that not every game is the same and that everything can happen. Another ambition with this idea was to make it really engaging and fun, so if you consider all those ideas and ambitions, the thing that first came to mind was a game. The other thing that made blackjack a really good choice was that when you play, you actually play against the dealer. So there is that competitive factor, that you also want to beat someone, which the group thought would be really fun.

2 Summary

When the player starts up the game the menu will pop up a text message asking the player to submit the current time, this is used as a seed to randomize the deck properly. Then the player will be given the options "Play" and "Quit". If the player types play, the game will start up. From there it's blackjack rules, the player will first get two cards and also see one of the cards that the dealer gets, this is standard blackjack rules that are used so that the player can get a better understanding of whether to be more reckless when choosing to hit or not. The player will then be asked what to do, either "Hit" or "Stand". If the player chooses to hit they will receive another card, and if the choice is to stand they won't receive another card. If the player chooses to hit and then get the sum of their cards over 21 then they bust, which means that they lose or tie and therefore won't get the choice of hitting or standing. But it's only a tie if the dealer also busts. There is one card that is a lot more interesting than the others, and that is the ace. Every card from 1-10 has the value of their number, then every dressed card is also 10 but the ace is either 1 or 11. For example, let's say that the player has an ace and a 9 on his hand (which is equal to 20 or 10) and asks to hit, then if he gets a 3 for example he shouldn't bust because then the ace gets reverted a 1 and the sum of the players cards are 13, therefore it would be wise for the player to keep hitting because he only has 13. What this means is that the ace is very forgiving, because if the dealer for example has a king showing, the player has to be bold when he chooses whether to hit or stand.So with the ace the player basically gets a second chance, because what would normally get the player to bust, magically doesn't if you have an ace in your hand.
Whenever the player feels that his cards are good enough to beat the dealer he chooses stand and lets the dealer reveal his cards. The dealer then proceeds to deal himself cards until he gets the sum of 17 because he isn't allowed to

go higher than that, which is the only advantage that the player has over the dealer, because the player has the option to keep hitting no matter what. If the player and the dealer has an equal sum it's the dealers win, but other than that its whoever has the highest sum that wins, and if there would be money involved the player doubles the amount of money he bet on that game, and if he loses he gives up all that money to the "house".[1]

3 Use Cases

3.1 Software requirements

To use the program, first cabal has to be downloaded, then GHCI and the random module needs to be installed. Then the game can be run by downloading the file and opening it in any text editor that supports Haskell. Then load the program and run it by typing "main".

3.2 How to play the game

When the player runs the file, there will pop up a text asking for the current time. Then in the terminal a menu with two options will appear. The player will get to choose whether to "play" or "quit". If the player chooses to quit, the program will terminate, but if the player chooses to play, the game will start and effectively replace the menu screen. Then the game of blackjack will start and it uses the classic blackjack rules which was described earlier. First the player gets to see their own two cards and at the same time see one of the dealers two cards. Then you will get the following options, "Hit" and "Stand". If the player chooses to stand then the the dealer will show his other card, and continue to hit until the sum of the dealers cards are equal to or more than 17. After that depending on whether the player or the dealer has the highest sum of cards on their respective hands they win. There is also a special case where the dealer and the player are in a tie, which means that the sum of their respective hands are equal. In that case the dealer wins every time, which is one of the biggest advantages that the dealer has over the player. However, if the player makes the choice to hit instead of stand, they will receive another card. The player is able to hit until the sum of cards in their hand either,goes to 21, the sum of the cards go over 21 or the player feels satisfied with the combined sum of cards in their hand. To put it simply, the player can decide whether to hit or stand and can choose freely when to stop. When the player doesn't want to hit anymore the dealer will show their second card and take another card until the sum of the dealers cards add up to 17 or more. After a game is over it will either print out "Win" or "lose" depending on whether the player wins or loses.

4 Data structures and data types

To be able to create a card game a few data structures for representing cards and a deck was implemented.

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

Figure 1: An image of the deck and card related types

To be able to identify the different playing cards in a deck the "Cardtypes" data structure was implemented. The data structure is simply made to be able to identify different cards from each other. The Suits data structure is not necessary for a blackjack game but was implemented anyway to make the rest of the implementation easier. An added bonus when the "Suits" data structure is implemented is that many functions that is implemented in our blackjack game could be used for other card games implementations. This would not be the case if only the "Cardtypes" data structure was implemented. The Card data type is also implemented to merge the "Cardtypes" and "Suits" to create a representation of a whole card. Data types for the hand and deck was also created and they consists simply by an array of "Cards".

```haskell
data GameState = GameState{
    deck :: Deck,
    playerHand :: Hand,
    dealerHand :: Hand
} deriving(Show)
```

Figure 2: An image of the gamestate data structure

As a game of blackjack unfolds the program needs to keep track which cards are in the deck and in the players hands. To be able to do this a gamestate data structure was implemented. As a game of blackjack progresses the variables in the data structure gets updated depending on what is happening in the game.

5 Control flow and functions

5.1 Menu function

```
menu :: IO()
menu = do
    putStrLn "\ESC[2JEnter Current time on format mdhms(Month, Day, Hour, Minute, Second"
    seed <- getLine
    putStrLn "\ESC[2JWelcome to our blackjack game! \n Play \n Quit"
    answer <- getLine
    let choice  | toUpper (pack answer) == pack "PLAY" = gameStart $ initState $ read seed
                | toUpper (pack answer) == pack "QUIT" = exitSuccess
                | otherwise = menu
    choice
```

Figure 3: An image that shows our main menu function

This function is the first function that is run when a user starts the game. It starts off by telling the user to write the current date, which it then uses as a seed to randomize the order of the cards in the deck. Because every time a user inputs a time it will be different, meaning that it will in fact be random cards that gets distributed every time. After the user puts in the current time the menu will pop up with two options, The user is able to either select the option to "Play" or "Quit". The user then selects one of those options which will start different functions that we will get to later. You can write "play" and "quit" with both uppercase and lowercase letters or even a blend of both and it will still work as long as you spell the word correctly.

5.2 initState function

```
initState :: Int -> GameState
initState seed = GameState {
    deck = fst $ fisherYates (mkStdGen seed) makeDeck,
    playerHand = [],
    dealerHand = []
}
```

Figure 4: An image that shows our initState function

The initState functions always runs before a new game of blackjack. The function initializes variables that will be used throughout the game. It creates a deck with a randomized order depending on what seed you had chosen previously. The function also initializes two empty arrays that will symbolize the player and the dealers hand.

5.3 fisherYates function

```
fisherYatesStep :: RandomGen g => (Map Int a, g) -> (Int, a) -> (Map Int a, g)
fisherYatesStep (m, gen) (i, x) = ((insert j x . insert i (m ! j)) m, gen')
  where
    (j, gen') = randomR (0, i) gen

fisherYates :: RandomGen g => g -> [a] -> ([a], g)
fisherYates gen [] = ([], gen)
fisherYates gen l =
  toElems $ foldl fisherYatesStep (initial (head l) gen) (numerate (tail l))
  where
    toElems (x, y) = (elems x, y)
    numerate = zip [1..]
    initial x gen = (singleton 0 x, gen)
```

Figure 5: An image that shows the fisherYates function

The Fisher–Yates shuffle is an algorithm used for generating a random permutation of a sequence. In other words the algorithm shuffles the sequence. It is used in the code to shuffle the deck at the start of the game. The algorithm works like this:
1. Pick a random number K between one and the number of remaining numbers in the sequence.
2. Counting from the beginning of the sequence, remove the K:th element in the sequence, and add it to the end of a separate sequence.
3. Repeat from step 1 until all the elements have been removed from the original sequence.
The sequence of elements written down in step 2 is now a random permutation of the original sequence.[3] This algorithm was not made by the group and was copied from the Haskell wiki.[2]
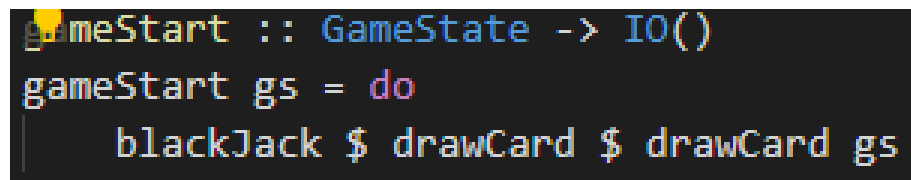
5.4 playerDrawCard function

```
playerDrawCard :: GameState -> GameState
playerDrawCard gs = gs { playerHand = playerHand gs ++ [head $ deck gs], deck = tail $ deck gs}
```

Figure 6: An image that shows our playerDrawCard function

This function is what makes the player able to draw a card from the deck. It takes the current "gamestate", then updates the players hand and the deck. It takes the players hand and adds the head of the deck to it(the deck is a list), which means that it adds the card that is on the top of the deck to the players hand. It then returns the "gamestate" that is after it has added the card that is on the top of the deck to the players hand. This means that the "gamestate" after the function has run is the very similar to the "gamestate" before, the

only difference is that the card that was on top of the deck is now in the players hand. There is an identical function for making the dealer draw a card.
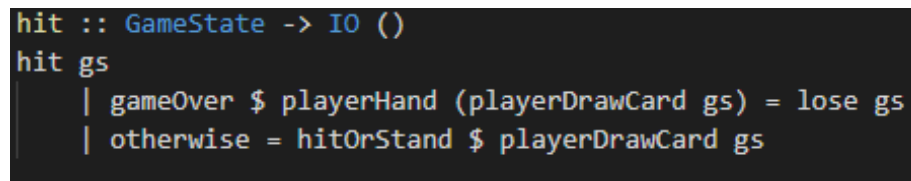
5.5   gameStart function

```
gameStart :: GameState -> IO()
gameStart gs = do
    blackJack $ drawCard $ drawCard gs
```

Figure 7: An image that shows our gameStart function

gameStart is the function that handles the very first part of a game of blackjack. The function hands out two cards to the player and two cards to the dealer from the deck. When the player and the dealer has been given two cards there have to be a check if the player has achieved blackjack. This check is handled in the blackjack function and if the player has got a blackjack the game is over, the player won. If the game is not over the blackjack function will call the hit and stand function which is the next part of the game. The "hitOrStand" function is basically only an interface using strings in the terminal and depending on what input the function gets it will call either the "hit" function or the "stand" function.

5.6   hit function

```
hit :: GameState -> IO ()
hit gs
    | gameOver $ playerHand (playerDrawCard gs) = lose gs
    | otherwise = hitOrStand $ playerDrawCard gs
```

Figure 8: An image that shows our hit function

The hit function gives the player another card (by calling the "playerDrawCard" function) which is stored in the "playerhand" part of the "gamestate" data structure described earlier. If the value of the hand is larger than 21 the game should be over, the player lost. The "gameOver" function is used to check that condition. If the game is not over the player should be returned to the "hitOrStand" menu once again and get to choose what to do.

5.7 stand function

```
stand :: GameState -> IO ()
stand gs = do
    if calculateAceHand (dealerHand gs) > 17 then calculateResult gs else
        stand $ dealerDrawCard gs
```

Figure 9: An image that shows our stand function

The stand function is run by the dealer. This function first checks if the result from calculateAceHand is less than 17 with the dealers hand as an argument. If that is the case, then it runs the calculateResult function to see if the player or the dealer won the game. If it isn't it will keep running the dealerDrawCard function which let's the dealer draw a card until it satisifies the condition and runs the calculateResult function.

5.8 cardValue function

```
cardValue :: Card -> Int
cardValue (Card Two _) = 2
cardValue (Card Three _) = 3
cardValue (Card Four _) = 4
cardValue (Card Five _) = 5
cardValue (Card Six _) = 6
cardValue (Card Seven _) = 7
cardValue (Card Eight _) = 8
cardValue (Card Nine _) = 9
cardValue (Card Ten  _) = 10
cardValue (Card Jack _) = 10
cardValue (Card Queen _) = 10
cardValue (Card King _) = 10
cardValue (Card Ace _) = 1
```

Figure 10: An image that shows our cardValue function

This function converts the cards that the player is holding to regular integers. The way this works is that every case is specified from the card "two" all the way up to "ace" and their respective integer value. It is also generalised so that it works the same way with every type of card there is, hence the underscore inside the parentheses. This is later used to be able to calculate the sum of the player and the dealers cards in their hands respectively.

5.9 hasAce function

```
hasAce :: Hand -> Bool
hasAce [] = False
hasAce ((Card Ace _): _) = True
hasAce (x:xs) = hasAce xs
```

Figure 11: An image that shows our hasAce function

This function is used as a helper function to the function "calculateAceHand". It takes a "hand"(which is a list of cards) as it's argument and then checks the "hand" to see if it contains an ace or not. If there are more than one card in the "hand" it keeps running the function on the list until it has checked every card in the "hand" and then returns either true or false depending on whether there was an ace in the "hand" or not.

5.10 calculateHand function

```
calculateHand :: Hand -> Int
calculateHand [] = 0
calculateHand (x:[]) = cardValue x
calculateHand (x:xs) = cardValue x + calculateHand xs
```

Figure 12: An image that shows our calculateHand function

This function is also used as a helper function to the function "calculateAce-Hand". This function takes a "hand" and goes through it recursively and adds all the elements together to get the combined value of the "hand". This might seem as the main calculate function but since the ace can have either value 1 or 11 we need another function to make the calculation part of the program complete.

5.11 calculateAceHand function

```
calculateAceHand :: Hand -> Int
calculateAceHand [] = 0
calculateAceHand hand = if hasAce hand && (calculateHand hand + 10 <= 21) then calculateHand hand + 10 else calculateHand hand
```

Figure 13: An image that shows our calculateAceHand function

CalculateAceHand is the function we use to determine the real value of the hand. It uses both the hasAce and calculateHand function by simply adding 10 to the value of calculateHand if the hasAce boolean is true and the value of the hand plus 10 is less than or equal to 21.

5.12   calculateResult function

```
calculateResult :: GameState -> IO ()
calculateResult gs = do
    if gameOver(dealerHand gs) then win gs else
        if calculateAceHand (dealerHand gs) >= calculateAceHand (playerHand gs) then lose gs else win gs
```

Figure 14: An image that shows our calculateResult function

CalculateResult is the function used to determine who won. If the value of the dealers hand is greater than 21 the player has won. If not the function checks who had the highest value of their hand. The highest value win. The function runs the win or lose function depending on if the player won or not.

6   Shortcomings

The known shortcomings of the program are some functionality that the group wanted to implement but did not really have the time to. The most significant one is how the game looks, the group wanted to implement some solution to that by creating a interface in gloss. It would probably have been an effective solution to that but the first idea was to just make it text based and it was first at the end of the project that the group thought about creating an interface with gloss, but it was ultimately decided that it was too late to make a change that substantial and still be finished with the program. Another thing that the group wanted to implement was that the player should have the option to split, alongside hit and stand. This was also not implemented because the group wanted to first finalize the basic choices hit and stand, before thinking about anything else. Therefore it would have been really stressful to try and implement an option like that when we were already short on time and had a functioning program. The final thing that the group thought about implementing was some kind of betting system, so that the player would be able to bet money every time they would play a game. This was also not implemented because of lack of time, and also that the betting part of the game was not something that the group found to be essential to the game as a whole.

References

[1] HitorStand.net. Blackjack rules. `http://www.hitorstand.net/strategy.php`.

[2] Wiki.haskell.org. Random shuffle. `https://wiki.haskell.org/Random_shuffle`.

[3] Wikipedia.org. Fisher-yates shuffle algorithm. `https://en.wikipedia.org/wiki/Fisher-Yates_shuffle`.