FIT 2102 Assignment 2 Report

Tarun Menon 29739861 Highest Elo : 1419 Highest Rank: 19

Code and Strategy

To begin we shall discuss the overall workings of the implemented code as well as the strategy and thought process that went into creating it, we shall mainly focus on the core functions pickCard, playCard and makeMelds which in turn use other functions that we will also discuss.

```
pickCard :: ActionFunc
pickCard topOfDiscard score mem _ crds
      | potentialScore <= currentScore = (Discard,memOut)
      | otherwise = (bestDraw,memOut)
      where curr = makeMelds score "" crds
            potential = makeMelds score "" (crds ++ [topOfDiscard])
            currentScore =  foldl (+) 0 $ map cardPoints curr
            potentialScore = foldl (+) 0 $ map cardPoints potential
            memOut = updateMemory score crds mem
            playerMem = getMem $ parse playerMemoryParser memOut
            bestDraw = analyseBestDraw (cardsPlayed playerMem) topOfDiscard
```

Our code begins with the pickCard function, the strategy in implementing this function was to of course decide whether the visible card from the discard pile or the unknown card from the stock is more valuable to draw. Thus in order to calculate this I decided to use memory and probability in order to thoroughly analyse which is the better option, this is specifically done through the function analyseBestDraw which we shall briefly discuss.

```
analyseBestDraw :: [Card] -> Card -> Draw
analyseBestDraw cards discard
            | averageValueOfRemainingDeck <= valueOfDiscard = Stock
            | otherwise = Discard
            where deck = createDeck
                  remainder = deck \\ cards
                  valueOfRemainingDeck = foldl (+) 0 $ toPoints <$> remainder
                  averageValueOfRemainingDeck = valueOfRemainingDeck `div` length remainder
                  valueOfDiscard = toPoints discard
```

As seen above, analyseBest draw takes in all cards seen by the player in a list as well as the card currently on top of the discard pile, it then generates an imaginary full deck and removes all cards from the deck which the player has already seen, this is done using the players memory. Finally at averages the remaining cards in the deck and compares this value to that of the discard pile card, based on the lower value we select return either Stock or Discard.

Going back to pickCard, it seems we have now decided which card is best to draw, however this is only the case if the discard card doesnt make a meld. Our code and strategy first checks whether the visible discard card can be used with the cards in our hand to increase the number of melds / reduce the amount of Deadwood in our hand (All melds are created using makeMeld which is discussed in depth later). This is done by utilizing fold over two sets of melds, one containing the discard card and one without. If the meld including the discard card has less Deadwood it is then optimal to select this card. Only when this is not the case do we utilize the analyzeBestDraw function to use probability and memory to make statistical “guess”.

Now that we have selected the best card to draw, we move on to the next phase of the game which is to select our action for this turn. We select the most optimal Action for our player in the playCard function which is seen in the screenshot below.

```haskell
playCard :: PlayFunc
playCard drawn score mem crds
      | foldl (+) 0 (map cardPoints finalMelds) == 0  && newRound meme == 1 = (Action Gin chosenDiscard,mem)
      | foldl (+) 0 (map cardPoints finalMelds) <= 10 && newRound meme == 1 = (Action Knock chosenDiscard,mem)
      | otherwise = (Action Drop chosenDiscard,mem)
      where currentMelds = makeMelds score mem (crds ++ [drawn])
            withoutDrawMelds = makeMelds score mem (crds)
            noDrawValue = maximumBy (\x y -> compare (cardPoints x) (cardPoints y)) withoutDrawMelds
            highestValue = maximumBy (\x y -> compare (cardPoints x) (cardPoints y)) currentMelds
            chosenDiscard = if (head (meldToCards highestValue) /= drawn)
                            then head $ meldToCards highestValue
                            else head $ meldToCards noDrawValue
            finalHand = (crds ++ [drawn]) \\ [chosenDiscard]
            finalMelds = makeMelds score mem finalHand
            meme = getMem $ parse playerMemoryParser mem
```

The strategy behind the playCard function is to check whether it is possible to call Gin or Knock based on the current hand, if so then we should immediately do so otherwise we should call Drop to continue making more melds as the game progresses, finally it also selects the best card to discard this turn in a similar concept to that observed in pickCard.

The way playCard achieves the first component of the strategy of calling Gin and Knock is as so, first we create the best set of melds we can based on our current hand, this is done using makeMeld. We then use foldl over this set of melds to reduce the points of hard to a single value, if this value is 0 this means there is no Deadwood in our hand and thus we can call Gin, likewise if the value is not 0 but less than 10 this means we can call Knock. Furthermore there is one last constraint stipulating that we can only call Gin and Knock if it is not the first turn, this is done using the memory to compare the current score with the previous, if they are different this is a new round.

The strategy behind the second component of playCard where we decide the best card to discard works similarly to pickCard. Here we create two sets of melds, one containing the drawn card and one without, we then find the highest value card (eg. Largest Deadwood) in the set of Melds with the drawn card and discard it, however if this card is the drawn card we instead discard the highest value card from the other set, this is to ensure that while we discard the most optimal card we never discard the drawn.

```haskell
makeMelds :: MeldFunc
makeMelds score mem l
    |length nonZeroMelds > 0 = out
    |otherwise = Deadwood <$> l
    where sorted = sortBy (comparing getRank) l
          cardPermutations =  subsequences sorted
          meldPerms = nub $ filterPermutations cardPermutations
          possibleMelds = map createMeld $ meldPerms
          nonZeroMelds = concat possibleMelds
          chosenMeld =  maximumBy (\x y -> compare (scoreHand $ head x) (scoreHand $ head y)) possibleMelds
          remainingCards = l \\ (meldToCards $  head $ chosenMeld)
          out = chosenMeld ++ (makeMelds score mem remainingCards)
```

The final phase of the game revolves around selecting our best set of melds, this is done in our final core function makeMelds.

The thought process going into implements makeMelds was that based on my current hand, how could I select the best meld. However I realised that if I used recursion I could actually make an entire set of Melds based on first my hand, then my hand without the cards needed for the previous meld!

As such the strategy of makeMelds worked as so, we first sorted our hand based on rank and then created all possible permutations of cards based on the hand. With these permutations we use filterPermutations which

filters out all permutations of cards which don't create a meld. We then remove the duplicates from the permutation of melds and use concat to squash them into a single list which now contains all unique melds which can be made from the given hand. Finally, we use maximumBy to select the highest value meld (Value of cards in meld) and select that as our chosen meld for this iteration. Lastly, we remove the cards used to create this meld from our hand and recursively call makeMeld with our remaining hand whilst concatenating our chosen melds.

The result of this recursive process is a set of melds created from each iteration, we finalise the process by calling Deadwood on the remaining cards in hand which couldn't be placed into melds.

As such to summarise the overall strategy of my implementation, in the first phase we optimally select the best card to draw based on whether it fits into a meld, if not we use probability and memory to make the best decision. We move on to the next phase where we call Gin or Knock if we can do so, if not we call drop and move on. Finally, we recursively call makeMeld to create the best possible set of melds we can based on our hand using permutations.

Memory and Parsing

The last section I would like to briefly discuss is the use of memory and parsing in this assignment, memory plays an important role in this implementation as it allows us to make informed decisions on drawing using probability as well as ensuring that during action selection we are not breaking the rules of acting in the first turn. As such we shall first see how exactly memory is updated before looking at parsing it.

```
updateMemory :: (Score,Score) -> [Card] -> Maybe String -> String
updateMemory s c existMemory = case existMemory of
              Nothing -> createBaseMemory s c (Nothing)
              (Just a)  -> update s c a
```

```
update ::  (Score,Score) -> [Card] -> String -> String
update s c mem = flag ++ show s ++ [ x | x <- show totalCards, not (x `elem` ",") ]
         where pm = getMem $ parse playerMemoryParser mem
               flag = if curScore pm == s then "1" else "0"
               totalCards = if flag == "0" then c else sortBy (comparing getRank) $ nub $ c ++ cardsPlayed pm
```

The function update is aptly responsible for updating our memory, it works in tandem with updateMemory to either create a base memory on the first turn or update existing memory. It first parses the current memory and stores it into a playerMemory object pm. We then use accessor functions for our playerMemory data type to retrieve the flag and total cards seen already. We update the flag by checking discrepancies with the stored and current score whilst total cards add's new cards to the stored set of cards. The memory is later used for optimizing draws and plays.

The last thing to discuss is the parsing, we create a set of parsers and create an ADT in order to create and read player memory. I began by creating a parser for each suit and rank as seen in the picture below. This process is repeated for each suit and rank.

```
diamondParser :: Parser Suit
diamondParser = string "Diamond" >> pure Diamond
```

```
aceParser :: Parser Rank
aceParser = string "Ace" >> pure Ace
```

I then combined these sets of parsers to create a stronger parser capable of parsing all suits and ranks.

```
suitParser :: Parser Suit
suitParser = diamondParser ||| heartParser ||| spadeParser ||| clubParser
```

```
rankParser :: Parser Rank
rankParser = aceParser ||| twoParser ||| threeParser |||
```

Next I combined these parsers to create a cardParser capable of parsing strings into a card object, a parser for reading strings into a score tuple was also created.

```
cardParser :: Parser Card
cardParser = do
   a <- suitParser
   _ <- spaces
   b <- rankParser
   _ <- spaces
   return (Card a b)
```

```
scoreParser :: Parser (Score,Score)
scoreParser = do
  _ <- is '('
  a <- numParser
  _ <- is ','
  b <- numParser
  _ <- is ')'
  return  (a,b)
```

Finally we create an ADT for our player memory which contains a flag for the round, the current score and all cards seen in play so far.

```
data PlayerMemory = PlayerMemory {newRound :: Int, curScore :: (Score,Score), cardsPlayed :: [Card]}
    deriving (Show)
```

Lastly, we combine all these different parsers we have created into one single parser capable of reading strings into this ADT for later easy data retrieval and use.

```
playerMemoryParser :: Parser PlayerMemory
playerMemoryParser = do
    x <- read <$> list digit
    a <- scoreParser
    _ <-spaces
    _ <-is '['
    b <- list cardParser
    _ <- is ']'
    return (PlayerMemory x a b)
```

Tarun Menon 29739861