## Game Strategy:

### Pick Card Strategy:

In this feature, the player picks up a card from the top of the discard pile if that card forms a meld with the hand of the player. Otherwise, the player picks up the card from the Stock. Picking up a card from the stock pile may form a meld or a deadwood. It is better to pick up a card from the stock when the card on top of the discard pile does not form meld as picking a card from stock will have 0.5 probability for forming meld. Memory in the pick card function is passed as an empty playCard function when the memory received by the pickCard function is nothing, otherwise the old memory is passed to the play Card function. The main source of the memory is controlled by the playcard function.

### Play Card Strategy:

This strategy is the heart of the game. In order to decide which card I want to discard, I will create two lists. The first list will contain the cards which will not form future melds and we will exclude the card which we just picked from this list, the second list will be of the cards which contain deadwood cards of the hand + the card we picked; we will also exclude the card which we just picked after forming this list. Now, in order to choose the card that I want to throw, I will first check list1 which contains cards that will not form future melds. If it is not empty, I will throw the card from this list which has the highest rank.If that list is empty, then I will look at the list which contains Deadwood cards (list 2) and will throw the card of highest rank from this list. There is also a possibility that both list1 and list2 are empty. In this case, it means that we might have to remove a card from our hand which forms a meld of size 4 or 5 in order to adjust a smaller meld. After choosing the card we want to throw. We will check the total deadwood value of our hand+card we picked- the card we wanna throw. Then afterwards, we will check our memory that if the memory is “”(empty) this means its start of the game and it is a first round and we can not call gin or knock and we can only call discard with card we have chosen to throw or if the score stored in the memory and the current score are same this means we can call knock or gin. There is another situation if the score stored in the memory and the current score are different, then we cannot call knock or gin as it is the first turn of a new round.

We will modify the memory in the playcard if the memory is empty or the score stored in memory and the current score is different, then we will put the current score in the memory. If the score in memory and the current score are the same then we will not change memory.

**Functions which are used all over the assignment:**

```
-- Sorts hand
quicksort :: [Card] -> [Card]
quicksort [] = []
quicksort (x:xs) =
    let lessthanpivot = quicksort [a | a <- xs, a <= x]
        biggerthanpivot = quicksort [a | a <- xs, a > x ]
    in  lessthanpivot ++ [x] ++ biggerthanpivot
```

This quicksort function is used to sort the cards according to the order given to them in data type of cards.

```
--Sorts hand according to rank
quicksortbyRank :: [Card] -> [Card]
quicksortbyRank [] = []
quicksortbyRank (x:xs) =
    let lessthanpivot = quicksortbyRank [a | a <- xs, getRank a <= getRank x]
        biggerthanpivot = quicksortbyRank [a | a <- xs, getRank a > getRank x ]
    in  lessthanpivot ++ [x] ++ biggerthanpivot
```

This function sorts the hand(list of cards) according to the rank of the cards.

```
-- Retrieves Suit of the hand
getSuit:: Card -> Suit
getSuit (Card s _ ) = s

-- Retrieves Rank of the hand
getRank:: Card -> Rank
getRank (Card _ r) = r
```

The getSuit function retrieves the suit of the Card and the getRank function retrieves the rank of the card.

**Important functions used for pickCard:**

**callCheckStraight:**

```haskell
-- This is used to check if the Card can be added to hand [Card] to form a straight.If it does it returns Tr
-- This is a caller function which uses helper functions called checkStraight,quicksort,getSuit in-order to
-- Main thing about this function is that it filters the hand [Card] according to the suit of the Card
callCheckStraight:: Card -> [Card] -> Bool
callCheckStraight c hand  = checkStraight (fromEnum (getRank c))
                                          (map (fromEnum . getRank)
                                               (quicksort (filter (\x -> getSuit x == getSuit c) hand))
```

**callCheckStraight function** take a card for which we want to check that we have a straight in our hand and this card suit is used to filter our existing hand in order to get cards of our hand which have the same suit as the card for which we want to find the straight and then we will sort this list using quicksort and then we will convert this filtered hand into list of integers where integer correspond to the enum value of rank of the card. We will pass this integer list and the enum value of the rank of the card to **check the straight function** which will return true if the card given forms a straight with the hand.

**checkStraight:**

```haskell
-- This function is Main Decision Maker for Straights.
-- Int  is Rank of the Card
-- [Int] is Rank of cards in a hand which belong to the Same Suit as Card
-- Returns True if the Card forms a straight with hand
checkStraight:: Int -> [Int] -> Bool
checkStraight c hand
    | c == 0 = isCardPresent 1 hand && isCardPresent 2 hand
    | c == 12 = isCardPresent 11 hand && isCardPresent 10 hand
    | c == 1 = (isCardPresent 0 hand && isCardPresent 2 hand) ||
               (isCardPresent 2 hand && isCardPresent 3 hand)
    | c == 11 = (isCardPresent 10 hand && isCardPresent 12 hand) ||
                (isCardPresent 10 hand && isCardPresent 9 hand)
    | otherwise = (isCardPresent (c+1) hand && isCardPresent (c-1) hand) ||
                  (isCardPresent (c+1) hand && isCardPresent (c+2) hand) ||
                  (isCardPresent (c-1) hand && isCardPresent (c-2) hand)
```

The **checkstraight function** gets an integer which represents the enum value of the card rank for which we wanna check straight and the second thing [Int] are the enum value of card ranks of the same suit as the card for which we wanna

check straight. We should first look at enum values for each card rank inorder to understand how this function works.The following table represents that:

| Rank | Enum value (INT) |
|---|---|
| Ace | 0 |
| Two | 1 |
| Three | 2 |
| Four | 3 |
| Five | 4 |
| Six | 5 |
| Seven | 6 |
| Eight | 7 |
| Nine | 8 |
| Ten | 9 |
| Jack | 10 |
| Queen | 11 |
| King | 12 |

This function checks that if the given card can form the smallest straight 3 meld then it returns true otherwise it returns false. This function checks this by looking at the following conditions:

1) If the card whose straight we are looking for has enum value of its rank to be 0 then we should have integer value of 1 and integer value of 2 in our hand to form straight.
2) If the card whose straight we are looking for has an enum value of its rank to be 12 then we should have integer values of 11 and 10 in our hand to form straight.
3) If the card whose straight we are looking for has enum value 1 then we should have integer 0 and integer 2 in our hand to form straight.

4) If the card whose straight we are looking for has enum value of its rank as 11 then we should have integer 10 and integer 12 in our hand to form straight.
5) For all the other cards we have a strategy that it can be a middle card in the straight or the last card or the first card.We check three conditions for this and if any of the three conditions are true then it forms a straight.Assume the card for which we are checking straight has enum value of its rank R.Then we will check the following conditions:
    a) Are R+1 and R-1 present in our hand [Int]
    b) Are R-1 and R-2 present in our hand [Int]
    c) Are R+1 and R+1 present in our hand [Int]

**isCardPresent function:**

```
-- This function checks If the card is present in the hand.
-- Int is rank of the Card which is being checked if its present in hand
-- [Int] is Rank of cards in a hand which belong to the Same Suit as Card
isCardPresent :: Int -> [Int] -> Bool
isCardPresent _ [] = False
isCardPresent c (x:xs)
  | c == x = True
  | otherwise = isCardPresent c xs
```

**isCardPresent function** just checks if the Int(Enum value of card rank) is present in the[Int](Enum values of card ranks in our hand) then returns True.

**checkSet function:**

```
-- This function checks if the Card forms a meld of type Set with hand [Card].
-- In order to achive this it calls helper function called checkSetOtherConditions.
checkSet:: Card -> [Card]-> Bool
checkSet c hand
  | length (filter (\x -> getRank x == getRank c && getSuit x /= getSuit c) hand) < 2 = False
  | otherwise = checkSetOtherConditions
                  (filter (\x -> getRank x == getRank c && getSuit x /= getSuit c) hand) hand 0
```

**checkSet function** is used to check if the card given forms a set meld with our hand. This is a caller function which filters the hand[Card] containing only the card of the same ranks, as the card for which we are checking the set. If after filtering our hand if the length of the filtered hand is less than 2 then it means the given card cannot form the set meld as the smallest set meld length is 3. After that, if the length of the filtered hand is greater or equal to 2 then we check other conditions for the given card in order to check if it will form a meld by calling **checkSetOtherConditions function**.

**checkSetOtherConditions function:**

```
-- Major Conditions checked by this function if the Card forms a meld of type set with the Hand
checkSetOtherConditions:: [Card] -> [Card] -> Int -> Bool
checkSetOtherConditions [] _ len = len  >= 2
checkSetOtherConditions (x:xs) l len = if callCheckStraight x l then
                                            checkSetOtherConditions xs l len
                                       else
                                            checkSetOtherConditions xs l (len+1)
```

This function takes the list of cards(list1) which are formed with our given card and the list of cards of our original hand and an integer value which is set to be 0. We will increment this integer value by 1 when the card in the list1 only form set meld with our hand. If it forms both set meld and straight meld then this integer value will not be incremented and when list1 is empty we check that our counter should have atleast value of 2 as adding the given card and these 2 cards which only form set will be able to form the smallest set meld of length 3.

**Important functions used for playCard:**

```
playCard :: PlayFunc
playCard card s m hand = (playfunc card (quicksort (card:hand)) (quicksort hand) s m, memoryScore s m)
```

**playCard function** will send a card,sorted (hand+ the picked card), sorted hand, score to the **playfunc function.** Playcard will also call memoryScore function and pass score and memory to this function.

**playfunc function:**

```
playfunc :: Card -> [Card]->[Card]->(Score,Score)-> String -> Action
playfunc card newhand hand
  = mainActionDecider
      (quicksortbyRank
         (filter (/= card) (formDeadWoodCards newhand newhand)))
      (quicksortbyRank
         (filter (/= card) (cardsWhichCantFormFutureMelds newhand newhand)))
      newhand hand card
```

**playfunc function** is basically a caller function that further calls **formDeadwoodCards,cardsWhichCantFormFutureMelds** whos results are a list of cards which are deadwood cards and another list which contain deadwood cards which can't form melds in future. These both list are then sorted

according to the rank of cards and filtered so they do not contain the picked card and passed to a function called mainActionDecider along with score memory,existing hand and newhands which contains the picked card added to existing hand.

**formDeadWoodCards:**

```
-- Function which takes hand and return cards which are present in the deadwood
formDeadWoodCards:: [Card]->[Card] ->[Card]
formDeadWoodCards [] _ = []
formDeadWoodCards (x:xs) hand = if not (callCheckStraight x hand) && not (checkSet x hand) then
    x:formDeadWoodCards xs hand
else
   formDeadWoodCards xs hand
```

This function takes two lists which are newhands( picked card + existing hand). Then these two lists are used to detect which cards do not form melds and are added to a new list which is returned and can be called the list of deadwood cards.

**cardsWhichCantFormFutureMelds:**

```
 -- Gives list of cards which cant form melds in Future
 cardsWhichCantFormFutureMelds:: [Card]->[Card]->[Card]
 cardsWhichCantFormFutureMelds [] _ = []
cardsWhichCantFormFutureMelds (x:xs) hand = if not (callFutureCheckStraight x hand) &&
                                             not (checkFutureSet x hand)
                                               then x:cardsWhichCantFormFutureMelds xs hand
                                          else
                                               cardsWhichCantFormFutureMelds xs hand
```

This function return the list of cards which will not form future melds and they do that by calling some additional functions called callFutureCheckStraight and checkFutureSet and if both these functions return false for the card then it adds that card to the list which is to be returned.

**callFutureCheckStraight function:**

```
-- This use to check if the card on discard can be added to hand to form future straight.If it does it retur
callFutureCheckStraight:: Card -> [Card] -> Bool
callFutureCheckStraight c hand  = furturecheckStraight (fromEnum (getRank c))
                                                       (map (fromEnum . getRank)
                                                         (quicksortbyRank
                                                           (filter (\x -> getSuit x == getSuit c) hand)))
```

This function is basically a caller function which takes a card for which we check the future straight and the hand and then it filters the hand so it only contain cards in hand which have same suit as the card for which we are checking the

straight. After this we convert this hand further into ints according to the rank of the card and call a function called **futurecheckStraight** which returns true if the card forms a future straight meld.

**futurecheckStraight function**:

```
-- This checks if the card can form straight in future
furturecheckStraight:: Int -> [Int] -> Bool
furturecheckStraight c hand
    |  c == 0 = isCardPresent 1 hand
    |  c == 12 = isCardPresent 11 hand
    | otherwise = isCardPresent (c+1) hand || isCardPresent (c-1) hand
```

**futurecheckStraight function** checks that there is at least another card which differs by 1 when the enum value of rank is in the hand. It return true if this condition meets.For example if we are checking future straight meld for **Card Spade Ace**
Then there should be **Card Spade Two in the hand** in order for this function to return true.

**checkFutureSet function:**

```
-- This checks if the Future set can be formed
checkFutureSet:: Card -> [Card]-> Bool
checkFutureSet c hand
 | not(any(\x -> getRank x == getRank c && getSuit x /= getSuit c) hand) = False
 | otherwise = futureCheckSetOtherConditions
                    (filter (\x -> getRank x == getRank c && getSuit x /= getSuit c) hand) hand 0
```

checkFutureSet takes the card for which we are looking at the future set meld. It filters the hand so it should contain cards which are of the same rank as the card for which we are looking for a future set meld. If this filtered hand length is 0 then we return False as this card cant form future meld.If not then we check other conditions for future set meld for this card by using another function called **futureCheckSetOtherConditions.**

**futureCheckSetOtherConditions function:**

```
- This check some other conditions that card supplied to form a set is not present in a straight
- and it forms a set.If these conitions are true it returns True.
futureCheckSetOtherConditions:: [Card] -> [Card] -> Int -> Bool
futureCheckSetOtherConditions [] _ len = len >= 1
futureCheckSetOtherConditions (x:xs) l len = if callFutureCheckStraight x l then
                                        futureCheckSetOtherConditions xs l len
                                    else
                                        futureCheckSetOtherConditions xs l (len+1)
```

**futureCheckSetOtherConditions function** returns true if the number of cards which only form a future set meld is greater than 0.

**mainActionDecider function:**

```
mainActionDecider:: [Card]-> [Card]->[Card]-> [Card]->Card->(Score,Score)-> String-> Action
mainActionDecider d fd newhand hand pick s m
  | not (null fd) = totalMeld (last fd)  newhand s m
  | not (null d) = totalMeld  (last d)  newhand s m
  | otherwise = totalMeld (rearrangeMelds
                        (formSetCards newhand (notinStraight newhand newhand)++
                         helperCardfunc (formStraightCards newhand newhand)) hand pick) newhand s m
```

This function decides from which list should we throw the card from. This function has three options. The first option and top priority is to throw the last card of the list of cards which will not form a future meld represented by fd. If the list represented by fd is empty then throw the last card of the list which contain deadwood cards.If this list is also empty then that means all the cards(cards in our hand + the picked card) are forming melds and now we have to resize melds and need to throw card from the largest meld. Rearranging of melds and choosing a card to throw is done by another function called rearrangeMelds which we will see later in this report. However, after choosing the card we call another function called totalMeld which calculates the total deadwood value of our hand + the picked card excluding the card which we decided to throw and then that function further calls action function which decides what Action to take.This function is important to choose what card we wanna throw. The additional functions like formSetCards, notinStraight, helperCardfunc and formstraightCards would be seen later in this report

**rearrangeMelds function:**

```
-- This function rearranges the melds when all cards (hand cards + picked card) form melds
-- It removes the card from larger melds in order to make space for smaller melds
rearrangeMelds:: [[Card]]-> [Card]-> Card-> Card
rearrangeMelds [] h _= if  not (null (quicksortbyRank (formDeadWoodCards h h)))
                       then last (quicksortbyRank(formDeadWoodCards h h))
                   else
                       last (quicksortbyRank h)
rearrangeMelds (x : xs) h pc
  | length x >= 4 && pc /= head x = head x
  | length x >= 4 && pc == head x = x !!(length x - 1)
  | otherwise = rearrangeMelds xs h pc
```

**rearrangeMelds function** checks for the meld of size greater than 4 or 5 and if it finds the meld of that size then it removes the first card of that meld if that first card is not equal to the card we picked from Stock or Discard. If the first card is equal to the card picked then we remove the last card from that meld.If we aren't able to find a meld of size greater or equal to 4 then eventually we remove the highest rank card from our original hand which is a very rare case.

**formSetCards function:**

```
-- This function form nested cards lists where each nested list has a set formed
formSetCards:: [Card]->[Card]->[[Card]]
formSetCards [] _ = []
formSetCards (x:xs) hand = if checkSet x hand then
                           filter (\z -> getRank z == getRank x) hand :
                           formSetCards (filter (\k -> getRank k /= getRank x) xs) hand
                       else
                           formSetCards xs hand
```

This function take two lists which both represents our hand and it takes a card from the hand and check if it forms set with our hand by using the function checkset used before and if it does then it filters all the cards of the same rank from our hand and append that filtered list to recursive call of formSetCards which receives a filtered list which does not contain the cards which have already been chosen in the last recursive call and like this it forms nested list of cards where each  list in nested list represents a meld and the size of that meld is retrieved by the length of the the inner list in the nested list.

**formStraightCards function:**

```haskell
-- This Function filters the cards used in the straight
formStraightCards:: [Card]->[Card]->[Card]
formStraightCards [] _ = []
formStraightCards (x:xs) hand = if callCheckStraight x hand then
    x : formStraightCards xs hand
else
    formStraightCards xs hand
```

This function uses the older function called callCheckStraight in order to create a list which contains the cards which form straight melds.

**helperCardfunc function:**

```haskell
-- This form straight melds in card form where each nested list length determines whats the size of the
helperCardfunc:: [Card] -> [[Card]]
helperCardfunc hand =  formStraightMeldCardForm (filter (\ x -> getSuit x == Spade) hand)
  ++
    formStraightMeldCardForm (filter (\ x -> getSuit x == Club) hand)
     ++
       formStraightMeldCardForm (filter (\ x -> getSuit x == Diamond) hand)
        ++  formStraightMeldCardForm (filter (\ x -> getSuit x == Heart) hand)
```

This function takes the card list formed by **formStraightCards function** and then it fiters that list by each suit and call **formStraightMelds** which return a nested card list where each inner list of nested list represent a straight meld and the size of that straight meld is equal to the length of the inner list of nested list. Then it combines melds of each suit into one nested list.

**formStraighMledCardForm function:**

```haskell
-- This function form the nested card list according to melds formed
formStraightMeldCardForm :: [Card] -> [[Card]]
formStraightMeldCardForm [] = []
formStraightMeldCardForm (x:xs) = (x:formConsectiveCardList x xs) :
                                  formStraightMeldCardForm (formListofNonConsectiveCards
                                                            (x:formConsectiveCardList x xs) (x:xs))
```

This function returns straight melds in card form where each inner list of nested list represents a straight meld and the size of this meld is represented by the inner list length.

**formConsectiveCardList function:**

```
-- This function is used to form straights.This function creates list of consective ranks
formConsectiveCardList :: Card -> [Card]-> [Card]
formConsectiveCardList _ [] = []
formConsectiveCardList c (x:xs) = if fromEnum (getRank x)-fromEnum(getRank c) == 1
                                  then x : formConsectiveCardList x xs
                                else
                                  formConsectiveCardList c xs
```

This checks that the difference between the last card and next card is 1 and adds them into a list. If the difference is not equal to 1 then it wont add this card to the list.

**notinStraight function:**

```
-- This function filters the hand by removing all cards which are present in straight
notinStraight:: [Card]-> [Card]->[Card]
notinStraight [] _ = []
notinStraight (x:xs) hand = if callCheckStraight x hand then
          notinStraight xs hand
else
  x: notinStraight xs hand
```

This function returns the card which aren't in the straight

**formListofNonConsectiveCards function:**

```
-- This function is used to form straights and this function creates a list of card
-- which werent used in the last straight
formListofNonConsectiveCards :: [Card]->[Card] -> [Card]
formListofNonConsectiveCards _ [] = []
formListofNonConsectiveCards [] (y:ys) = y:ys
formListofNonConsectiveCards (_:xs) (_:ys) = formListofNonConsectiveCards xs ys
```

This function returns the card which was not used in the last straight meld.

**totalMeld function:**

```
totalMeld:: Card -> [Card]->(Score,Score)-> String -> Action
totalMeld c newhand
  = actionDecider
      (totalDeadWoodValueOfHand
         (formDeadWoodCards
            (filter (/= c) newhand) (filter (/= c) newhand)))
      c
```

This function helps to calculate the totaldeadwood value of the which contains the picked card and all the cards of the original hand except the card which we decided to throw. The totalDeadWoodValueOfHand calculates the value of total deadwood and passes this value and card to the actionDecider function which makes the decision of Action.

totalDeadWoodValueOfHand and deadWoodValueOfCard function:

```
-- This function calculates total deadwood value for the hand
totalDeadWoodValueOfHand:: [Card] -> Int
totalDeadWoodValueOfHand [] = 0
totalDeadWoodValueOfHand (x:xs) = deadWoodValueOfCard x + foldr ((+) . deadWoodValueOfCard) 0 xs

-- Tells the deadwood value of each Card
deadWoodValueOfCard:: Card -> Int
deadWoodValueOfCard c
  | getRank c == King = 10
  | getRank c == Queen = 10
  | getRank c == Jack = 10
  | otherwise = fromEnum(getRank c)+1
```

These two functions are responsible for calculating the total deadwood value of hand.

**actionDecider function:**

```
actionDecider:: Int -> Card->(Score,Score)-> String -> Action
actionDecider value c s m
  | value == 0 && roundDetection s m = Action Gin c
  | value < 10 && roundDetection s m = Action Knock c
  | otherwise = Action Drop c
```

**actionDecider function** takes the total deadwood value as value and it takes the card which it wanna throw as c and then it uses another function which uses roundDetection function which uses score represented by s and memory represented by m.

```
-- Return True if its First Round else False if some other round
roundDetection:: (Score, Score)->String -> Bool
roundDetection s m
  | (m == "") || (s /= getMem (parse choices m)) = False
  | otherwise = True
```

The roundDetection function uses a parser and has two functionalities. If memory is empty that means the game has just started and it's the first round of the game. If the score stored in memory is different from the current score supplied then it is also a first round and we cannot call knock or gin. If the score supplied and the score in the memory is the same then it is not the first round and we are allowed to call knock or gin and it will return true else it returns false.

**memoryScore function:**

```
-- Memory Function which decides what to pass in memory
memoryScore :: (Score, Score)->String -> String
memoryScore s m
  | (m == "") || (s /= getMem (parse choices m)) = "(" ++ show (fst s) ++ "," ++ show (snd s) ++ ")"
  | otherwise = m
```

This function updates the memory. If the start of the game and memory is empty or if the score saved in memory is different from the current score then update the memory with the current score being saved in memory. If the score is the same in memory and current score, then dont change memory and send back the old memory.

**Important function used for makeMelds:**

**makeMelds function:**

```
makeMelds :: MeldFunc
makeMelds _ _ hand = callToMakeStraightMeld (quicksort hand) ++
                     callToMakeSetMeld (quicksort hand) ++
                     formDeadWood (quicksort hand) (quicksort hand)
```

This function is just a caller function which concatenates the Melds returned by functions called callToMakeStraightMeld, CallToMakeSetMeld and formDeadWood.

**callToMakeStraightMeld and CallToMakeSetMeld functios:**

```
-- This function is called by makeMelds to form Melds of type Straight
callToMakeStraightMeld:: [Card] -> [Meld]
callToMakeStraightMeld c = helperMeldfunc (formStraightCards (quicksort c) (quicksort c))

-- This function is called by makeMelds to form Melds of type Set
callToMakeSetMeld:: [Card] -> [Meld]
callToMakeSetMeld c = formSetCardMelds (formSetCards c (notinStraight c c))
```

These two functions are also caller functions for helperMeldfunc and formSetCardMelds.

**helperMeldfunc function:**

```
-- and then merge them into single meld list
helperMeldfunc:: [Card] -> [Meld]
helperMeldfunc hand = makeStraightMelds (formStraightMeldCardForm (filter (\ x -> getSuit x == Spade) hand))
  ++
    makeStraightMelds (formStraightMeldCardForm (filter (\ x -> getSuit x == Club) hand))
      ++
        makeStraightMelds (formStraightMeldCardForm (filter (\ x -> getSuit x == Diamond) hand))
          ++ makeStraightMelds (formStraightMeldCardForm (filter (\ x -> getSuit x == Heart) hand))
```

This function is just like helperCardfunc, but in this there is another extra functionality of calling makeStraightMelds which returns straight melds.

**formSetCardMelds function:**

```
formSetCardMelds:: [[Card]] -> [Meld]
formSetCardMelds [] = []
formSetCardMelds xs = concatMap decidetypeofSetMeld xs
```

This function takes the nest card list formed by formSetCards and then converts them into melds using decidetypeofSetMeld function.

**decidetypeofSetMeld function:**

```
-- This decides what type of Set Meld is stored in [Card]
decidetypeofSetMeld:: [Card]-> [Meld]
decidetypeofSetMeld c
  | length c == 3 = [Set3 (head c) (c !! 1) (c !! 2)]
  | length c == 4 = [Set4 (head c) (c !! 1) (c !! 2) (c !! 3)]
  | otherwise = []
```

This function just checks the length of the inner card list sent by formSetMCardMelds to determine if it is a set meld of type Set3 or Set4.

**makeStraightMelds and decidetypeofStraightMeld function:**

```haskell
makeStraightMelds:: [[Card]] -> [Meld]
makeStraightMelds = concatMap decidetypeofStraightMeld

-- This checks whats the form of StraightMeld store in [Card]
decidetypeofStraightMeld:: [Card]-> [Meld]
decidetypeofStraightMeld c
  | length c == 3 =  [Straight3 (head c) (c!!1) (c!!2)]
  | length c == 4 =  [Straight4 (head c) (c!!1) (c!!2) (c!!3)]
  | length c == 5 =  [Straight5 (head c) (c!!1) (c!!2) (c!!3) (c!!4)]
  | length c == 6 =  Straight3 (head c) (c !! 1) (c !! 2) :
                     [Straight3 (c !! 3) (c !! 4) (c !! 5)]
  | length c == 7 =  Straight3 (head c) (c !! 1) (c !! 2) :
                     [Straight4 (c !! 3) (c !! 4) (c !! 5) (c !! 6)]
  | length c == 8 =  Straight4 (head c) (c !! 1) (c !! 2) (c !! 3) :
                     [Straight4 (c !! 4) (c !! 5) (c !! 6) (c !! 7)]
  | length c == 9 =  Straight3 (head c) (c!!1) (c!!2)  :
                     Straight3 (c !! 3) (c !!4) (c!!5) :
                     [Straight3 (c !! 6) (c!!7) (c!!8)]
  | length c == 10 = Straight5 (head c) (c!!1) (c!!2) (c!!3) (c!!4) :
                     [Straight5 (c !! 5) (c!!6) (c!!7) (c!!8) (c!!9)]
  | otherwise = []
```

**makeStraightMelds** function takes the nested list created by **formStraighMledCardForm** and then passes each inner list to **decidetypeofStraightMeld function** to decide what type of straightMeld is formed.

**Pasrer implementation:**

**Satisfy:**

```haskell
satisfy :: (Char -> Bool) -> Parser Char
satisfy f = do
  c <- character
  let next = if f c then pure else unexpectedCharParser
  next c
```

This function is used from tutorial 11 and it returns a parser that produces a character but fails if input is empty and if character does not satisfy the given predicate.

**digit:**

```haskell
digit :: Parser Char
digit = satisfy isDigit
```

This function is taken from tutorial 11 and it checks if the given char is a digit or not using a builtin function called isDigit.

**scoreParserForTwoDigits:**

```haskell
-- This parses the score tuple of the form (twodigits,twodigits)
scoreParserForTwoDigits:: Parser (Int,Int)
scoreParserForTwoDigits = do
  _ <- is '('
  a <- digit
  b <- digit
  _ <- is ','
  c <- digit
  d <- digit
  _ <- is ')'
  pure (digitToInt a *10+ digitToInt b,digitToInt c *10+ digitToInt d)
```

I have saved the score in tuple form within my memory. This function checks if the score of the player and score of the opponent has double digits then it parses it.Example “(10,20)”

**scoreParserForOneDigits:**

```haskell
scoreParserForOneDigits:: Parser (Int,Int)
scoreParserForOneDigits = do
  _ <- is '('
  a <- digit
  _ <- is ','
  c <- digit
  _ <- is ')'
  pure (digitToInt a,digitToInt c)
```

I have saved the score in tuple form within my memory. This function checks if the score of the player and score of the opponent has one digit then it parses it.Example “(1,7)”

**scoreParserForOneDigitAndTwoDigit:**

```
scoreParserForOneDigitAndTwoDigit = do
  _ <- is '('
  a <- digit
  _ <- is ','
  b <- digit
  c <- digit
  _ <- is ')'
  pure (digitToInt a,digitToInt b *10+ digitToInt c)
```

I have saved the score in tuple form within my memory. This function checks if the score of the player is one digit and score of the opponent is double digit then it parses it. Example “(3,28)”

**scoreParserForTwoDigitsAndOneDigits:**

```
scoreParserForTwoDigitAndOneDigit:: Parser (Int,Int)
scoreParserForTwoDigitAndOneDigit = do
  _ <- is '('
  a <- digit
  b <- digit
  _ <- is ','
  c <- digit
  _ <- is ')'
  pure (digitToInt a*10+digitToInt b,digitToInt c)
```

I have saved the score in tuple form within my memory. This function checks if the score of the player is in two digit and score of the opponent is in single digit then it parses it. Example “(45,7)”

**choices:**

```
choices:: Parser (Int,Int)
choices = scoreParserForTwoDigits |||
          scoreParserForOneDigits |||
          scoreParserForOneDigitAndTwoDigit |||
          scoreParserForTwoDigitAndOneDigit
```

This decides what type of parser should it use for score saved in memory. If one parser fails then it tries the other parses until it finds the correct parser to parse the memory.

**getMem:**

```
getMem :: ParseResult a -> a
getMem (Result _ cs) = cs
getMem (Error _) = error "Score out of Bounds"
```

This function helps us to extract our data structure out of the Parser Result a. Whereas, a is our data structure in which this case is a tuple containing score of player and opponent.