Name: Lim Jing Ee
ID : 31961274

FIT 2102 Programming Paradigms Assignment 2 – TwentyOne Report

Strategy

In a game of TwentyOne, there are 2 roles, that is the *Dealer* and *Players.* Essentially, during the game, it's the *Players* against the *Dealer.* Hence, your opponent is the *Dealer.* Therefore, when playing a hand. It is extremely important to play based on the dealer's upcard. Also, 3 deck of cards is used for the game and is only reshuffled when the cards are used up. Hence there will be 156 possible cards, 3 possible cards that are the same and 12 cards each for a Rank. Hence, strategy used:

- Play based on Dealer's upcard (Poor/Fair/Good/Strong)
- Calculate probability of desired cards (Big-Cards/Small-Cards)

Design of the code

Code source is divided into 2 sections: Action & Memory

Action

This part consists of functions to calculate the Action of a Player. Main function is ***action.*** In *action* is where other functions are called: *twoCards, getDCard, memoAction, doubleAction, splitAction, getPPOints, pSmallCards, pBigCards, division, isEqual, flatten.*

Memory

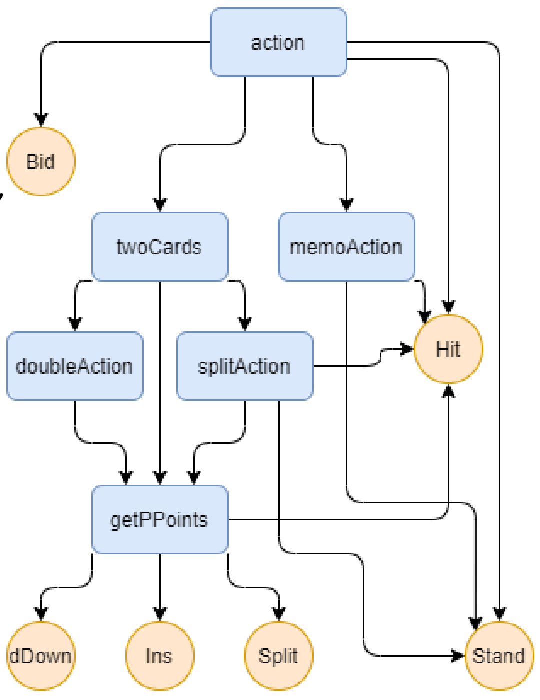
This part consists of functions to update the player's memory. Main function is ***newMemory***. In *newMemory* is where other functions are called: *getActMem, justMemo, actMemo, updatePlayedCards, updatePlayedCards', sumPlayedCards etc.*

Function *playFunc* will only have to call the 2 main functions that is *action* and *newMemory* along with their required output. Therefore, the function output will have a call like this:

```
playCard dcard ppoints pinfo pid memo hand = (action hand dcard memo pid ppoints, newMemory hand dcard pinfo memo pid ppoints)
```

Reason why the main functions are chained with so many other functions is to properly sectioned possible scenarios encountered during a game of TwentyOne. Why chained with so many functions? E.g. in *action*:

Player will have 2 cards or more. Before the start of each round (player has 0 cards), player must perform action Bid. However, actions such as DoubleDown, Split and Insurance are only allowed at the start of the round (player has 2 cards). Hence the function *twoCards* to attend these possible actions. Nonetheless, within *twoCards,* it consists of functions *doubleAction* and *splitAction.* This is to assure if DoubleDown is the best action or to otherwise Hit. Similarly, *splitAction* checks if Split is the ideal action or to otherwise Hit or Stand. To check whether the actions is ideal, players will look into the dealer's upcard via *getDcard* and check the probability of desired cards via *pSmallCards* or *pBigCards* which are not illustrated in the flow graph to the right. If DoubleDown, Insurance or Split is the ideal action, player will check if they have enough points to execute the respective functions with the help of function *getPPoints*. If insufficient points, it will Hit. Also, if memory contains actions such as Hit or Stand after DoubleDown. They will be executed via *memoAction.* Each functions used are commented/documented with its purpose in the code.



**Simplified action flow graph*

Memory and Parsing

<u>BNF Grammar</u>

```
<memory> ::= <actions> ”,” <playedcards><action>
<actions> ::= <action><action>
<action> ::= “H” | “S” | “i” | “”
<playedcards>::= <cards><cards><cards><cards><cards><cards><cards><cards><cards><cards><cards><cards><cards>
<cards> ::= <number><rank>
<number> ::= <digit> | <digit><digit>
<digit> ::= “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | ”8” | “9”
<rank> ::= “A” | “B” | ”C” | “D” | “E” | “F” | “G” | “I” | “N” | “J” | “Q” | “K”
```

In Memory, we will need to store the number of cards played to calculate the probability of the being dealt a desired card. Thus, for every round of the played we will need to store the cards that were played. A data type *Played* was introduced to store and show number of cards played. In memory, A=Ace, B=2, C=3, D=4, E=5, F=6, G=7, I=8, N=9, T=10, J=Jack, Q=Queen, K=King. Reason is because the <number> are already digits. Furthermore, “H” is not 8 because “H” is already used in <action>. Eg:

```
P = Played {{ace'=1, two'=2, three'=3, four'=0, five'=0, six'=0, seven'=0, eight'=0, nine'=0, ten'=0, jack'=0, queen'=0, king'=0}
show p = “,1A2B3C0D0E0F0G0I0N0T0J0Q0K”
```

Parsing plays an important role in the project. A function *parseUntil* is crucial because it helps with extracting relevant data from the memory which is of type String. From then we can use *getBefore* or *getAfter* obtain data from the output of parseUntil. Eg:

```
memo = “HS,0A0B0C0D0E0F0G0I0N0T0J0Q0K”
getAfter (parse (parseUntil ‘,’) memo) = “0A0B0C0D0E0F0G0I0N0T0J0Q0K” (card part)
getBefore (parse (parseUntil ‘,’) memo) = “HS” (action part)
```

Functional Programming and Haskell Language Features Used

Pattern matching, Gaurds, If-then-else were used in the project to match possible arguments for the function. Gaurds were used a lot due to the variety of conditions needed to be checked with.

Record syntax was used to as a data type for the number of cards played. It is called *Played*. Custom instance was created to show said data type in the desired form.

where and *let* keywords were used to create multiple function definitions and variables that are visible within the scope.

<$> was used in the project to apply a function into a context. For example, it was used to apply a function to a list.

Folds were essential in the project as there were cases where we needed to the reduce function. For example, *foldl* was used to update the memory of the cards that were played because there are a list full of cards played and all of them had to be updated one after another.

Parsers were crucial for extracting desired and relevant data from the memory that was filled with mixture of action part and card part. Parsers helped extract specific part of the memory and specific data of the part extracted. There is a section specifically or parser at the bottom of the Memory section.

do notation was used were used in a few of the parser functions instead of applying Monad’s flip operator to help ease readability of the code.