**FIT2102, Semester 2, 2021, Assignment 2: TwentyOne**

Hee Zhan Zhynn
31989403

My code mainly uses the datatypes defined in *Type.hs* and type signatures are included as part of the written parsers and functions. Guards and do-blocks are used to improve readability and comments are also included for non-trivial code block. Under the function *playCard,* I coded it in such a way that doubleDown has the second highest precedence followed by decision to obtain *Charlie*. This is shown by placing the code in an earlier number of lines compared to other decision making conditions, the player will then try to *doubleDown* or draw a card to obtain *Charlie* whenever possible compared to the normal *Hit* or *Stand.* For example, when the hand value is less than 17, the player checks whether it is possible to obtain *Charlie* before checking whether it should *Stand* instead*. Player.hs* is mainly broken into three parts for readability and maintainability, they are namely *import* block, *playCard* function block and *Parsers.*

For the function *playCard*, the memory is Nothing on the first round. I store the player's points during the bidding round in the memory so that for each time the function is called, I parse the player's points in the memory to pull out the value stored and compared it with the current points. If the absolute difference between the two points is equal to twice the initial bid, it means that the action *doubleDown* is played. In the player's next two turns, the player should automatically make the decision to *Hit* and then *Stand*. Thus, actions for a valid *doubleDown* are satisfied by comparing the player's points at the start of each round to the current points at each turn. At the start of each new round, I reinitiate the memory to store the player latest's points, that is during the bidding where the dealer's up card is Nothing. I have also defined a parser for cards where I will extract the largest card on hand from the memory. This is used to decide if the player should go for *Charlie* if it already has four cards where the largest card in the memory is small enough and the total hand value is less than 17. The AI will follow a heuristic strategy based on the dealer's up card and player's hand value, it will take the necessary actions to get the highest possible hand value while not risking going *Bust.*

**BNF**

```
<cardPoints> ::= <card> “;” <points>

<card> ::= <suites><rank>

<points> ::=  <digits>

<digits> ::= <oneDigit> | <twoDigits> | <threeDigits> | <fourDigits> | <fiveDigits>

<fiveDigits> ::= <digit> <digit> <digit> <digit> <digit>

<fourDigits> ::= <digit> <digit> <digit> <digit>

<threeDigits> ::= <digit> <digit> <digit>

<twoDigits> ::= <digit> <digit>

<oneDigit> ::= <digit>

<suites> ::= “S” | “C” | “D” | “H”

<rank> ::= “A” | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | “T” | “J” | “Q” | “K”

<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Our memory is stored in the form of “[largest card];playerPoints” where ‘;’ is used as a separator for the card and player’s points. Since our memory is stored as a string, I would need to parse it to extract the data. To parse a card string, I first need to define our suit and rank parser. The “|||” operator allows me to apply the first parser and then an alternate parser if the first fails, this allows us to encode the alternatives in the BNF grammar rules. Over here, I used the bind >> operator to sequence two actions while ignoring the result from the first, this allows us to parse the suit and rank accordingly. I then have a parseCard parser which parses a card string along with a parseHand which parses a string of concatenated cards. The card parser allows me to extract the cards in my memory, I can use it with *sortOn* to extract the largest or smallest card value.

As for the player’s points, *parseScore* is a parser that uses list, satisfy and digit to parse a string of digit only characters. Since our memory is stored in the form of “[largest card];player’s points” with “;” as the separator, our *parseScore* parser will only parse the points which are digits and extract it out from the memory string. I used *reverse* to get the score part in the memory.

There are attempts to apply Haskell concepts to the code written wherever possible. Guards | is heavily used in my *playCard* function to replace if-else statements. I have also included typeclass functor such such as fmap <$> to calculate the value of the dealer’s up card while typeclass monad operators such as bind >>= and pure are used for my parser function. For my *getResult* parser function, I used pattern matching to match a certain pattern and return the data needed while other patterns will result in a parsing error, also known as error handling. This produces an exhaustive pattern matching.

The strategy used is simply getting the highest hand value possible without *Bust*-ing while always bidding the minimum bid. My AI would *Stand* when the hand value is more than 16 and decide whether to *Stand, Hit* or *doubleDown* based on the current hand value and dealer’s up card. It *Stand* because the chances of getting a card value smaller than 5 is low, making the probability of going *Bust* high when the hand value is more than 16, thus *Stand*-ing will be a safe option. The AI will *stand* when the dealer’s up card value is between 2 and 6, *Hit* otherwise. This is again based on the general probability of counting cards that the dealer will have a higher probability of going *Bust* when more cards are drawn, therefore, *Stand*-ing is low risk and increases the chance of winning, however, if the dealer has a high up card value, it is worth risking drawing a card to achieve a higher hand value despite having the risk of going *Bust* since the dealer might already have a higher hand value compared to the AI.

The biggest challenge was in implementing the *doubleDown* action as it follows a strict rule, that is *Hit* and then *Stand.* I had difficulty in making sure that the AI *Stand* right after the *Hit* action and not to be conflicted with the normal *Hit* and *Stand* during a non-*doubleDown* action. In the end, I managed to implement the correct *doubleDown* action sequence by storing the player’s points during the bidding round as memory and compare it with the current points during each turn such that if a *doubleDown* action was made and the difference between the points is twice the initial bid, it is understood that the following two actions should be *Hit* and *Stand* respectively. Therefore, I was able to make the next two actions as required for *doubleDown.*

Due to limited knowledge and time constraint, extensions for the game had not been done as I was focusing on code testing and improving my player’s strategy.

## References

Higher Media (2011). Learn blackjack strategy while you play! Retrieved from http://www.hitorstand.net/strategy.php

Monday Morning Haskell (2020). Blackjack: Following the Patterns. Retrieved from https://mmhaskell.com/blog/2020/4/27/blackjack-following-the-patterns

Tim (2021). Parser Combinators. Retrieved from https://tgdwyer.github.io/parsercombinators/

Tim (2021). Data Types and Type Classes. Retrieved from https://tgdwyer.github.io/haskell2/

Tim (2021). Monad. Retrieved from https://tgdwyer.github.io/monad/

Wilfred (2013). Blackjack. Retrieved from https://github.com/Wilfred/Blackjack/blob/master/Blackjack.hs