FIT2102 Assignment 2 Report (Twenty-One)

Name: Khong Lap Hoe  
Student ID: 32114818  
Tutorial Group: MA Group 6, tutor Dr. Vee

**Code Design**

The design of the code started off with a custom memory data-structure, that is a record-syntax style **GameMemory** that stores non-trivial information, which can then be serialized and deserialized such that we can use these information to enable card-counting strategies to play the game. With each call of **playFunc**, the code design follows the cycle below, and this design makes it easy to keep track of memory flow across game iterations when designing further aspects of the code.

The code design aims to breakdown complex functions into smaller ones like in playerInfosToCount rely on smaller helper functions to aggregate so it is easier to solve a complex problem, and using function searches from Hoogle to shorten code length. Making small maintainable functions will allow for better testing, as we can ensure the correctness of larger functions by making sure small functions are implemented properly. Much time is spent here to refactor code.

One possible flaw in the code design is using large where blocks for mathematical calculations to our strategy. Personally, I feel like it is fine as this makes it clear that the calculation for this particular portion of the code. In fact I would argue that creating functions for one-time calculations would be harmful to code readability as having it outside the where block means it could be used elsewhere, but in fact the particular calculation is only for a specialized function. Most of the mathematical calculations are done in **basicStrategy** and **biddingStrategy**, and the math to these two calculations are completely different.

**BNF Grammar**

My memory string consists of 8 different non-trivial information about the game. Each of these information is separated by a “|”. We need a separator because we do not know how many characters does a non-terminal has. Like in cardsSeen, the number ranges from 0 to 156. The parser combinator helped me complete the parsing as we **need to parse both strings and integers**, namely actions and all the other info, and a parser combinator combined both of these different parsers into one using monads, that is presented by the do-notation in the code. The BNF grammar for the parser below is rather simple, but the memory has a structure of **Bid | Previous Action | Rank | winStreak | Running Count | Current Count | Cards Seen | Current Cards Seen** and serves the purpose of card counting for bidding strategies.

<gameMemory> ::= <number> “|” <actions> “|” <number> “|” <number> “|” <number> “|” <number> “|” <number> “|” <number>

<number> ::= <digit><number> | “”

<actions> ::= “H” | “S” | “B” | “D” | “d” | “L” | “I”

<digit> ::= “-“ | “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”

BNF Grammar for Parser Combinators

**Functional Programming and Haskell Language Features Used**

Almost every section of the code tried to apply Haskell Features and Functional Concepts ranging from simple pattern matching to Applicatives. In the code, we can see **pattern matching** is used to determine the three possible playing outcomes of playFunc. There are only 3 situations that could arise as describe in code, where the rest of the impossible patterns will give an error. This is possible by leveraging **wildcards** as well, to exhaust all possible input. Transitioning from PureScript, we should always provide **type signatures** to our functions and constants, and not to rely on Haskell’s type-inference system, which is done for all functions.

**Partially Applied functions** are also heavily used, such as in the lookup tables, where we just want to return an action, without having to know what the bid is. This makes bidding much more dynamic, like the action **DoubleDown** is stored as a partially applied function. In order to satisfy the return type, we can use consts to negate the second argument for actions that do not require an input bid like Hit or Stand. From lambda calculus, we have learnt how to perform **eta reduction**, **operator sectioning** and **composition**, and it is also widely used in my code such as in the **foldr** for **playerInfosToCount** to improve code-readability, as we shred away unnecessary arguments from the function.

Since Haskell is a functional language, we can’t avoid using recursion, due to its declarative nature like in the modified **lookupQ** function where we use recursion to find the correct key. **Maybe types** are also used to ensure that partial functions such as **lookUpQ** has *something to return* in case the **lookUpTable** does not contain the item needed. In which we know that we should Hit instead. Another variant of using **Maybe types** are also in **playerInfosToLength/Count** where we can either specify a Just **playerId** to not take into account of information about a player in **[PlayerInfo]** or specify **Nothing** to aggregate everything in **[playerInfo].**

**Functor typeclasses** are used to apply functions over a structure, such as a list, which are instances of the functor typeclass, for example obtaining a list of hands from **[playerInfo]**, which is *convenient*. The **foldable typeclass** works particularly well in my code as foldr is often used to aggregate list instances, such as in **addToMemory**, and **playerInfosToLength/Count** where playerInfo is converted to a desired metric. As list is an instance of the foldable typeclass, foldr is already pre-defined. **Applicative typeclasses** are used to apply **funcList** (list of **(->)r** ’s) to **GameMemory** using **sequenceA**. It is because a list of functions is just a list of applicatives, as functions are instances of Applicative. Hence, we can quickly obtain **GameMemory** information this way in the form of a list. **Monads typeclasses** are often used to in parsers such as in parseMemory where do notation is used instead.

**Evolution of Code**

The most interesting part of the code is the **lookUpTable functionality**, **addToMemory function** and **the memory metrics stored**. The basic strategy was first implemented as guards, quickly refactored into a more maintainable table upon ED Forum suggestions. The **addToMemory function** was designed to change **multiple aspects** of the memory while still maintaining its succinctness by **folding over a list of newData** with the required change. The memory metrics also stored **rank** and **winStreak**, and this is often the mindset of gamblers where **the more you win the more you bet!** So, the bidding number is also based on the number of games that lead to a positive change in ranking compared to other players. The code was originally 400 lines long without comments and the hardest challenge was trying to show that I understood most of what was taught in the unit! After rereading tutorials and attending consult, I have managed to refactor the code to 200 lines long without comments. I have also spent a great amount of time to understand how information is passed on from one game to another so that cards are counted properly.

\*THIS REPORT IS OF 2 PAGE IN LENGTH IF DIAGRAMS AND BNF ARE REMOVED.