Lionie Annabella Wijaya
31316115

Game Strategy and Design of the Code

This implementation of TwentyOne player aims to play card according to current hand, maximize profit, and cut loss by using perfect basic strategy, card counting, and tracking face cards. Basic strategy (figure 1) is a set of rules for different types of hands where a hand consists of exactly two cards, to recommend the most suitable action that increases winning chance. Card counting (figure 2) is assigning each card to a count value and accumulate them over the game as running count to calculate true count for bidding deviation - greater true count indicates more greater-valued cards left in decks meaning player has an edge and should increase bid to earn profit, vice versa. Tracking face cards is done to calculate probability of dealer's card being a face card to decide taking insurance.

PLAYER HAND	DEALER'S UPCARD									
	2	3	4	5	6	7	8	9	10	A
HARD TOTALS										
5 TO 8	H	H	H	H	H	H	H	H	H	H
9	H	D	D	D	D	H	H	H	H	H
10	D	D	D	D	D	D	D	D	H	H
11	D	D	D	D	D	D	D	D	D	H
12	H	H	S	S	S	H	H	H	H	H
13	S	S	S	S	S	H	H	H	H	H
14	S	S	S	S	S	H	H	H	H	H
15	S	S	S	S	S	H	H	H	H	H
16	S	S	S	S	S	H	H	H	H	H
17+	S	S	S	S	S	S	S	S	S	S
SOFT TOTALS										
A2	H	H	H	D	D	H	H	H	H	H
A3	H	H	H	D	D	H	H	H	H	H
A4	H	H	D	D	D	H	H	H	H	H
A5	H	H	D	D	D	H	H	H	H	H
A6	H	D	D	D	D	H	H	H	H	H
A7	S	D	D	D	D	S	S	H	H	H
A8+	S	S	S	S	S	S	S	S	S	S
PAIRS										
2,2	SP	SP	SP	SP	SP	SP	H	H	H	H
3,3	SP	SP	SP	SP	SP	SP	H	H	H	H
4,4	H	H	H	SP	SP	H	H	H	H	H
5,5	D	D	D	D	D	D	D	D	H	H
6,6	SP	SP	SP	SP	SP	H	H	H	H	H
7,7	SP	SP	SP	SP	SP	SP	H	H	H	H
8,8	SP	SP	SP	SP	SP	SP	SP	SP	SP	SP
9,9	SP	SP	SP	SP	SP	S	SP	SP	S	S
10,10	S	S	S	S	S	S	S	S	S	S
A,A	SP	SP	SP	SP	SP	SP	SP	SP	SP	SP

S STAND H HIT D DOUBLE DOWN SP SPLIT

Figure 1 *Basic strategy chart*

Retrieved October 23, 2021, from https://www.playsmart.ca/table-games/blackjack/strategies/

$$\text{True count} = \frac{\text{Running count}}{\text{Remaining decks}}$$

2, 3, 4, 5, 6	+1
Ace, 10, Jack, Queen, King	-1
7, 8, 9	0

Figure 2 *True count calculation*

Retrieved October 23, 2021, from https://www.liveabout.com/beating-blackjack-with-simple-plus-minus-count-537127

To implement these strategies, some form of hard coding is unavoidable for basic strategy and memory is utilized for card counting and tracking face cards as information must be accumulated over time. To be clear and concise, the basic strategy consisting of three combinations: playing pair, soft hand, and hard hand are hard-coded into three separate functions that returns the advised action. Memory is a custom data type containing information collected from earlier turns including initial bid, last two actions played, running count, number of cards used, number of face cards used, and previous player information. Creating memory as custom data type with record syntax allows easier access later on when properties are needed. In each turn, new cards are fetched by comparing previous to current player information and used to update memory's properties which are running count, number of cards, and number of face cards. These properties are then used to calculate and determine bid as well as taking insurance. Memory is also used to handle back-to-back actions like DoubleDown (by checking last two actions) and provide bid amount (from initial bid) for actions that need it. Further discussion on memory can be read in the next section.

The main function playCard will encounter several scenarios which leads to different action selected and changes in memory. Therefore, pattern matching, guards, and cases are implemented to handle these scenarios. Two main scenarios are game just started and game has been running, which are checked by pattern matching the memory parameter of type maybe. Action at start of a game is always bid with minimum bid as currently there are no information yet to determine bid amount and a default memory string is returned together with the bid action. Whereas when game has been running for some time, memory contains enough information to guide our AI to pick suitable action and bid amount. For a running game, player can be either at the start of a hand or in the middle of a hand which is checked with cases to see if dealer has any card. Similar to starting a game, a starting hand should take bid action but this time bid amount is determined from true count calculated with information in deserialized memory. When dealer has an Ace after bidding turn, insurance is taken if the chance of dealer having a face card calculated is high enough. If it is not, main function calls for playOtherThanBidAndInsurance function to return action from one of these: action following DoubleDown, action advised by basic strategy (calling functions defined for basic strategy) when player has two cards, or action when player has more than two cards. For both situation where player is at the start or middle of hand, after an action is selected we need to update memory accordingly before returning selected action and memory string serialized from memory data. Smaller functions are created to handle the update for each properties inside memory, some takes previous property's value and update it with current

information provided, e.g. folding new cards count with previous running count as initial value and some simply takes new value and replace it, e.g. initial bid and new player info.

Several difficulties are encountered in implementing my ideas throughout this assignment. One of the interesting challenges is updating memory which consists of different components that mainly depends on tracking new cards. This can be confusing as player can only track players before their turn and tricky for some cases such as when player has split into multiple hands. To solve issue of not knowing player info after their turn, I saved player info as part of memory in each turn to be compared at the next turn at all stages including bidding as their previous cards should still appear before given the option to bid. The data type of list which is instance of functor, foldable, and monad allows us to process the player info parameter and the hand inside the player info in a simple and sequential manner using a few custom and default functions to obtain the new cards in each turn. Another challenge is reducing redundant codes and checking patterns, initially I have pattern matched all game states and let basic strategy functions to immediately return action and updated memory. However, lines relating to updating memory are almost similar in each game state with minor modification and continuously repeated in each pattern as well as in the basic strategy functions. Not only it is redundant, but this also causes the game to run slower as pattern match that calls the basic strategy function unnecessary repeated the update memory lines which adds more complexity. To solve this issue, I decided to only return action advised for basic strategy function and delegate memory update to be done only in main playCard function. To further reduce redundant code in playCard's pattern matching, I combined patterns when player is in the middle of the game into a single one and break it down inside using into two cases, one case for the start of a hand and another for the middle of a hand. This allows a single where to be used in the pattern match instead of copy pasting for multiples patterns.

Memory and Parsing

Memory is serialized into string which properties are separated with semicolon as delimiter after each turn and deserialized into memory data type at the start of each turn. To serialize the memory data, we use both show and custom functions to turn data type into string. To deserialize memory string, BNF grammar is implemented into parsing through parser combinator. Parser combinator concept is used here to build a more defined, higher-order parser function which consist of smaller parser inside. This is important to complete our parsing since our memory contains multiple information of different type and each property inside the memory may further consists of different parsers.

Memory BNF Grammar

<memory>	::=	<initBid> ";" <lastTwoActions> ";" <count> ";" <numCards> ";" <numFaces> ";"<lastPlayerInfo>
<initialBid>	::=	<digits>
<lastTwoActions>	::=	<action> | <action> <action>
<count>	::=	<negativeDigits> | <digits>
<numCards>	::=	<digits>
<numFaces>	::=	<digits>
<lastPlayerInfo>	::=	[,<playerInfo>...]
<playerInfo>	::=	<playerId> " " <hand>
<playerId>	::=	"\" """ <string> "\" """
<string>	:: =	character | <string> character | ""
<hand>	::=	[,<card>]...]
<card>	::=	<suit> <rank>
<negativeDigits>	::=	"-" digits
<digits>	::=	<digit> || <digits> <digit>
<digit>	::=	"0" ... "9"
<action>	::=	"H" | "S" | "B" | "D" | "Z" |"I"
<suit>	::=	"S" | "C" | "D" | "H"
<rank>	::=	"A" | "2" | "3" | "4" | "5" | "6" |"7" | "8" | "9" | "T"

According to the BNF grammar, memory consists of initial bid at start of a hand, last two actions done before current turn, number of cards used, number of face cards used, and last player info saved from previous turn. Likewise, parseMemory function follows the same structure to this grammar and is built of multiple property related parsers. Each property parsers may be built of more smaller parser functions, one of the more complex example is lastPlayerInfo property, which is parsed using parsePlayerInfo that parses a list of player info. Player info is parsed using parseAPlayerInfo that parses ID and hand, and so on until terminal is reached. Several properties related to number such as initBid, count, numCards, and numFaces depends on the same parser, digits parser. Initially in the main parseMemory, I could simply create parser for non-delimiter characters to parse these properties and simply read it into number, however I decided to use digits parser and other helper parsers to be consistent with the BNF grammar.

Functional Programming and Haskell Features

This assignment follows functional programming style and uses Haskell features learned throughout the unit. Functions are broken down into multiple, smaller functions to aid different functionalities, increase readability, and avoid redundancy in code. For example, the main playCard functions calls multiple, smaller function to check different conditions and may call subfunctions to get an action based on basic strategy. Concept of higher-order function is used as well to create more defined and complex functions, for instance, implementation of parser combinators to deserialize memory. Haskell features like pattern matching, cases, and guards are used to check conditions and different scenarios in the game starting from determining current game state, action to take based on basic strategy, getting count value of a card, getting bid amount based on true count, and more. Custom data type using record syntax is done in memory data type, allowing access to information inside memory without the need to define accessors manually. Functor typeclass is useful to deal parameter in form of containers such as list of cards or player info which can be mapped to perform operation or get specific information, e.g. getting all IDs from players and ranks from cards. Foldables typeclass is used often as well to reduce some value from containers, e.g. when updating running count over new cards. Monoid is used during memory serialization as strings are of type monoid and can be concatenated together. Monads are widely used in parser either using bind operators or syntactic do notation, and used to flatten/join in other occasions such as combining all cards from a list of player info and getting all new cards in current turn. Codes are written in point free style for better readability with techniques introduced such as eta reduction, operation sectioning, composition, and flip.

References

Blackjack Strategies. (n.d.). PlaySmart. Retrieved October 23, 2021, from https://www.playsmart.ca/table-games/blackjack/strategies/

How to Beat Blackjack With a Simple Plus Minus Count. (2019, January 22). LiveAbout. Retrieved October 23, 2021, from https://www.liveabout.com/beating-blackjack-with-simple-plus-minus-count-537127