EXPLANATION

MEMORY

The game strategy is based on the optimal melds that my player can form. Hence, the memory will contain two pieces of information:

1) deckHeight: the number of cards remaining in the stock pile
2) notInDeck: list of cards that have already appeared in my hand / top of the discard pile, hence not in the deck anymore

Memory is saved in the form "(deckHeight)[cardSequence]" where cardSequence is stored in the form card suit followed by two digits representing the card rank.
For example, "(10)[C00S01H02D03]" represents that there are 10 cards left in the stock pile and cards Card Club Ace, Card Spade Two, Card Heart Three and Card Diamond Four are not in the deck anymore.

Deck height

- Used to determine when to call Knock
- Decremented whenever someone draws from the stock

Card sequence

- Used to determine potential melds to be formed
- Card strings are added to the sequence whenever a new card that is not in memory yet surfaces on top of the discard pile or my player picks a card from the stock

In order to parse memory, parser functions are written to read deck height and card sequence from memory.

WORKINGS OF CODE / STRATEGY

Pick card

1) Checks whether to take from Discard or Stock

Form melds from current hand
↓
Convert deadwood cards into Leftovers
↓
Filter out pair leftovers (SetPair and StrPair)
↓
Check whether card on discard pile is able to form melds with any of the pair Leftovers
↓
Take from discard pile if yes, otherwise, take from the stock

This is to prevent the opponent from deducing what melds the player is forming by deducing from what cards the player takes from discard, hence only take from discard if guaranteed to make a non deadwood meld.

2) Updates memory

- Initializes memory upon the first round of the game by memorizing all first 10 cards player is dealt with
- Updates the memory to record the card that appears on top of discard pile
- Updates the memory to decrement the deck height if anyone draws from the stock

Play card

Builds a binary tree from memorized card sequence (including last picked card)
↓
Convert deadwood cards into Leftovers and sort them
↓
Identify whether their neighbours are still in the deck (not in memory tree)
↓
Discard a card from leftover that has no more neighbours in the deck;
If all have no neighbours in the deck, discard the leftover with lowest priority;
If no leftovers (all card in hand can form melds), discard the smallest card in hand
↓
Replace card to be discarded in hand with last picked card
and determine what to call based on new hand

Neighbours: neigbouring ranked / suit cards that a Leftover is able to form melds with

How to determine what to call?

1) Always call drop for the first round of a game by checking length of card sequence in memory

If it is the first round of the game, the memorized card sequence will contains at most 12 cards: 10 cards from dealt hand + 1 card from discard pile + 1 card player picked

2) Otherwise, call Gin as soon as possible

3) Otherwise, call Drop if deadwood count is more than 10

4) Otherwise, call Knock if deadwood count is less than 5, or less than 16 cards in the deck left

According to my research, a game typically ends after half the deck is gone. Hence, when the half deck mark is reached, my player will call knock the moment there are less than 10 deadwood points. Original deck height : 52 - 10 - 10 = 32, as each player is dealt 10 cards in the beginning, hence half deck is when there are 16 cards left.

5) If deadwood count > 5 and more than 16 cards in the deck left, Drop

In this case, my player is betting that there is a better chance to have lower deadwood cost.

Make melds

Straight melds have the most potential to grow, as a Straight set can consists of three to five cards and grow into ten cards, which can form a Gin, compared to Set melds can only consist of four cards maximum. Hence, Straight melds are prioritized.

Make Straight melds from current hand
↓
From the card that are unable to form straight melds, attempt to from Set melds
↓
The rest of the cards will be the Deadwoods

KEY FEATURES OF CODE

1) Defining data types for different purposes

- **Melder**
- **Potential**
- **Leftover**
- **Bintree - explained later**

Melder is used when forming melds and **Potential** is used to group cards that cannot form melds together, which will then be converted into **Leftover**s. Generally, Melder and Potential are defined for similar purposes, which is to prevent processing the same cards again. This will increase efficiency and prevent bugs due to reusing the same cards.

For example, after getting the sets of **cards that will form Straight melds,** they will be converted into melds and put into the **melded** compartment of the Melder data structure, while the **cards that will not form Straight melds** will be left in the **unmelded** compartment. Then, after getting sets of cards that will form Set melds, those cards will be melded and goes to the melded compartment. Finally, the rest will be converted into deadwoods and moved to the melded compartment.

The same goes with Potential, after identifying pairs of cards to form a Leftover from, we want to separate them from the rest of the cards. The gotGroup compartment is reserved for consecutively ranked/suited pairs of cards that can potentially form melds, while cards that cannot be paired stays in noGroup.

After grouping all Deadwood cards into Potentials, we can convert them into **Leftovers**, namely StrPair, SetPair and Single, which represents pairs of cards that can form straights, pairs of cards that can form sets and single cards. The purpose of defining leftovers is so that we can sort them to know which cards to prioritize, which will be explained later.

2) Leveraging instance derivations

Orderable Leftover

The orderable instance for leftovers is leveraged to sort the leftovers in increasing priority to aid deciding which card to discard. Lowest priority will be the first to be considered when discarding.

The reasoning of the order defined is as such:

- As stated in make melds, straights have greater possibility to grow compared to sets
- Cards of middle ranks are more valuable compared to the others as they have the ability to form straights that can grow either way (by getting smaller or larger cards), hence they are prioritized.
- Other than cards ranked 6 / 7, smaller ranked cards will be prioritized over greater ranked cards

Singles greater than 6 / 7 **<** Singles less than 6 / 7 **<** Singles which are 6's / 7's **<** Set pairs **<** Straight pairs greater than 6 / 7 **<** Straight pairs less than 6 / 7 **<** Straight pairs consisting of 6 / 7

Showable Cards

The showable instance is derived for compact serializing of cards, where suits are showed as:
Heart : 'H'; Diamond: 'D'; Spade: 'S'; Club: 'C'
while the ranks are represented in two digits starting from Ace: 00, Two: 01, …… King: 12

Equatable Draw

Self explanatory equality instance derived for Draw.

Foldable Bin Tree
To enable easy building of a binary tree form a list of cards using foldr, a Foldable instance of BinTree has been derived. This will keep code clear and neat.

3) Optimizing search complexity
For searching cards, linear search is used during updating card on discard into memory for pickCard function since there is only one card that we to check whether it already exists in memory or not. But for the playCard function, my player will check whether the neighbours of leftovers exist in the card sequence memorized until one that has no neighbours in the deck is identified. Since there might be a lot of searches that need to be done, a binary tree data structure is used. Building a tree once and using it for multiple searches allows optimization of complexity.

4) **Use of functors, applicatives, monads, foldables etc**
Concepts applied:

- Pattern matching
- Sugar syntax: filterStuff function
- Monads: Join, bind operators
- Functors and applicatives: (<$>), (*) , lift
- Foldables: foldl, foldr
- Parsers: a large portion of the code which is self - explanatory
- Currying: by passing an argument to a binary function and mapping it over a list
- In built functions introduced in the unit: all, head, take, drop, any, not, flip etc

5) Function composition, point free code and higher order functions
Function composition and point free code is used to ensure readability and neatness of code wherever possible, though not to the extent that it makes code ambiguous. Besides, higher order functions are also implemented to encourage reusability, for instance:

- safeTestRank: pass in functions pred and succ to find the verify whether cards are predecessing / successing in terms of rank
- safeFindRankNeighbour: pass in functions pred and succ to find cards of predecessing / successing rank
- findWho: returns functions pred and succ according to what should be testing against given card
- getLeftover: pass in functions, convertStraight, convertSet, to convert a list of cards into required data type
- filterStuff: pass in function to get cards of desired property (suit or rank)
- lift and satisfy