COM2108 Functional Programming - Autumn 2025

WRITTEN REPORT: KARMA

DEADLINE: 3PM, FRIDAY 12th DECEMBER 2025

DESIGN

To start, I first created a top down design tree for the functions listed to implement. Not only did this let me get my head around how the system was supposed to work, it also let me know which ones to implement first, as the ones to implement first should be independent of other functions, as this means that they can be tested first. For example, these were: legalPlay, replenish cards, giveWastePileTo and shuffleDeck. After implementing these, I found that most shared code, for updating the player, so I quickly turned that into a function, which searches through the player list for a player matching the playerId, and swapping it out for the player passed in. I tested it quickly to confirm it worked, then tested the ones relying on it to ensure they worked with it. As a quick example, replenishCards works by comparing the number of cards less than 3 in the players hand to 0, taking the max, so that if the player has more than three cards, it won't start replenishing cards. If they do need to take cards, it will take the top card off the drawPile, and recurse, until there are three cards in the player's hand. It then updates the player using updatePlayer

Then, I could move up to the next layer, so functions like validPlays, basicStrategy.

I made a few helper functions to aid the design of the game, mainly lower level functions. I'll explain the most used here:

updatePlayer: this takes in a player object, and searches through the list of players in the gameState to find the one with the same pId. It then replaces that player with the new one

head': this is mostly used when I want to take the head of the discardPile, which could be empty. It is head but allows for an empty case, where it returns Nothing. It is then fed into a function which expects Maybe Card anyway.

removeCard: when a player plays a card, as there is only one of that card in the deck, it can search through the player's hand, faceUp and faceDown to find the card/cards played, and remove it from their cards.

nextPlayer: this will return the index of the next player in the player list. It accounts for the order (order in the gameState), and is helpful as it can be used in stealCard, which relies on getting the next index.

setupAndPlay: this gets the players from the gameState, deals to them, gets the starting player, and starts the gameLoop. Its seperate from playOneGame as the functions in setupAndPlay rely on there being a gameState, which isn't the case in playOneGame yet.

findStartPlayer: gets the starting player. It is first called with a rank of three, as is filters the players based on if they have threes or not, and chooses the one with the most threes. If no one has threes however, then it will recurse and take the next rank as the argument. I haven't put a base case in it, as so long as the players have been dealt to, it should always be able to find a player with cards.

outputStats: this is used in gameLoopWithHistory, it just adds what is happening that turn to the messages field in the gameState

runTournament: as I am unable to recurse on playTournament, I made runTournament to do it. Every time it is called, it plays one game, takes the results and combines them with another call to itself. It returns the results to playTournament

The way my gameLoop is implemented is: first, it calls applyStrategy, which will first figure out which strategy the player is using, using the strategy field stored in the player data-structure. Then it will use that to get a list of the cards to play, if it can. If the list is empty, then it will pick up the discard pile, otherwise it will add them to the top of the discardPile. Finally, it will decide, based on that card, if there are any actions to be made (like a 9 stealing a card or four of the same card deleting). gameLoop will then check if the current player is out of the game, by checking if hand, faceUp and faceDown are emtpy. If they are out, it adds the player to the finishedOrder list, and removes it from the playerList. Finally, it will set the currentIx to the next player, and recurse. If all but one player has finished, then it will add that player to the end of finishedOrder, and return it. gameLoop also is responsible for updating the currentIx, as this makes more sense, due to the fact that some elements of a player's go is still happening in gameLoop, such as the updating of the discardPile, and any actions that happen from it (like stealing from the next player, which would require a function to get the last player, or updating the direction of play, which wouldn't happen if I did it in apply strategy) and the check to see if they are out of the game.

basicStrategy works by first getting the player, then if the hand is empty, it will check the faceUp cards. If that is also empty, it will return the first card from faceDown (which doesn't really matter as you can't see it anyway). Otherwise it will find the lowest rank card from faceUp, and if the hand isn't empty, it will return the lowest rank from the hand. To extend this to basicStrategySets, when selecting the card I instead stored the rank of the lowest legal card, and then used a list comprehension to return the set of all cards matching that rank.

I have played this game a lot, albeit with slightly different rules with my family – its the go to card game for us. Because of this, I know the general strategy, which is what I implemented. However, due to deleting the pile not giving a new turn, you gain no real benefit from doing so, which made me change my strategy slightly. I spent a lot of time trying to make a good smart strategy from scratch, doing things like saving a 7 so I can't be caught out by it, only playing power cards one at a time, and playing aggressively if the next player was low on cards. I couldn't find a bug in my code, but also couldn't beat basicStrategySets consistently with it. Instead, I started by using basicStrategySets. The biggest improvement on this was giving the cards a better ranking system, where I used a function cardRank, which uses pattern matching based on the rank passed in to score it. To use this, I created another function minimumByCardRank, which returns the lowest power card from the cards passed in. I used it instead of minimum in basicStrategySets, so it would choose the cards better, and thus win more. I tried to implement saving 7s again, however it is generally not worth it to save a 7 over far more powerful cards – it's too situational. Another thing that worked is playing more aggressively when the next player to play is getting low on cards, then I will play a high card, in order to stop them from winning, and make them pickup. I made another helper function for this, which was maximumByCardRank, to find the most aggressive card I can play. It makes sure to 2s and 10s from it, as that would make it easier for the next player, instead of harder. One final thing that netted me an extra percent is not playing power cards one at a time, when the drawPile is empty, as this means that you want to get rid of cards as fast as possible. This lets me win around 47% of the games played, versus one basicStrategySets player, and a basicStrategy player. I have created a flowchart for the decision making, see appendix

The way I implemented playTournament is also nice. It takes in the number of games, but I found it nicer to break down into a helper function, called runTournament, which would run the tournament, and return a list of the winners, gained from running playOneGameTourney. Then playTournament could easily tally it all up and output it.

In the appendix I have a decomposition tree for when you call playOneGame (appendix 2) I didn’t create one for playOneGameWithHistory or playOneGameStep4, as they are almost exactly the same thing, with the exception of playOneGameWithHistory having a call to outputStats, to update the messages output. As calling playOneGameWithHistory is so similar, I didn’t create it here. I have also attached one for smartStrategy (appendix 1)

TESTING

I tested along the way, to ensure that the functions depending on others wouldn't break, and it would be easier to track down bugs. Again, as I planned what I was going to implement first, by breaking it down, I implemented functions bottom up, so that none relied on others. This meant that I could test them first, so when used in other functions, I could be relatively sure that the error occurred in the newly written function. Sometimes I wanted to make a new helper function, which meant that I made it after the functions above, but I still tested it properly to stop errors. To see the break down, see appendix 2. To do this, I loaded Karma into ghci, initialised a gameState and a couple of players like this:

```
gen <- newStdGen

let deck = [Card r s | s <- [Clubs], r <- [R2 .. R5]]

let p1 = Player 0 "Alice" [Card R5 Hearts, Card R7 Clubs] [] [] "basic"

let p2 = Player 1 "Bob" [Card R6 Spades] [] [] "basic"

let testState1 = GameState [p1, p2] 0 deck [Card R4 Diamonds] [] gen [] "" True False False False

let finalState1 = execState applyStrategy testState1
```

I set up the gameState and players in the gameState manually, and then ran it with a function to see what the output was. I can replace 'applyStrategy' in the code above with any function that relies on the state, which is rather convinient, and considering that I was testing at the end and not following test driven development took either similar or less time than Hunit or another testing framework. To try and minimise the amount of bugs, I used test data that was errornous, normal and boundrary.

An important bug that I found this way was in applyStrategy, where I wasn't applying the rules (burn on 10, burn on top four the same etc.) until the next turn, which lead to the next turn being played on an out of date discard pile. This is because I wasn't binding the updated value of discardPile after I had added the player's card to it. An easy fix, but a large bug, and one that wouldn't have appeared unless I specifically tested for it. I found it with this test:

```
ghci> gen <- newStdGen

ghci> let p1 = Player 0 "Alice" [Card R10 Hearts, Card RJ Clubs] [] [] "basic"

ghci> let p2 = Player 1 "Bob" [Card R6 Spades] [] [] "basic"

ghci> gen <- newStdGen

ghci> let p1 = Player 0 "A" [Card R10 Hearts, Card RJ Clubs] [] [] "basic"

ghci> let p2 = Player 1 "B" [Card R6 Spades] [] [] "basic"

ghci> let testState = GameState [p1, p2] 0 [] [Card R4 Diamonds, Card R3 Clubs] [] gen [] "" True False False False False 0

ghci> let finalState = execState applyStrategy testState

ghci> discardPile finalState

[]

ghci> burnedPiles finalState

[[Card {rank = R10, suit = Hearts},Card {rank = R4, suit = Diamonds},Card {rank = R3, suit = Clubs}]]
```

Another test that I did, again on apply strategy is an erroneous data – when the player has no legal cards.

```
gen <- newStdGen
```

```
let p1 = Player 0 "Alice" [Card R3 Hearts, Card R4 Clubs] [] [] "basic"

let p2 = Player 1 "Bob" [Card R6 Spades] [] [] "basic"

let testState = GameState [p1, p2] 0 [] [Card R7 Diamonds] [] gen [] "" True False False False False 0

let finalState = execState applyStrategy testState hand (head (players finalState))

discardPile finalState
```

This worked fine. By testing a normal input and erroneous input, I can be sure that the functions work well. On top of that, because of how strict Haskell is, it is not going to get any runtime errors, so by testing this way for logic errors in each individual function, there will be few errors. On top of this, due to the reliance of other functions on others, the others will also be tested even more.

I have also used the playTournament function a lot, running tens of thousands of games, which have all run and given non-erroneous results, so the main logic must be sound.

I also decided to not use Hunit, mainly because of time constraints. I would've had to learn how to use it, and spend time writing tests, which as I was not using test driven development, it was easier to just write the tests manually and run them. I did save the tests that I had written out into a google doc, meaning that if I wanted to reuse a gameState or player, I could just copy and paste into ghci.

In my testing, I found another bug where the game gets stuck in an infinite loop. I found this when I added in traceM to output the hands of each player, and saw they were swapping hands effectively. After having narrowed down the error, I found that it was not an issue with the code, but an issue with the strategies, as the players can get in a constant loop of picking up each others cards. Therefore I decided to add a maximum number of iterations (150) as it was beyond what I ever saw from a non looping game, and still didn't take too much time to reach. I added it to the gameState, and in gameLoop I increment the iterations, and check if it is over the max. If it is, I just return the finishedOrder there and then. Usually there is still a winner, as it happens most between two players.

Another bug with 7s not working properly in legalPlay, which I found through my manual testing. The fix was reasonably simple: the guards were the wrong way around, meaning I was checking if the cardRank was lower than the rank of the head of the discard pile. This meant it was never matched, so moving it to the top of the function, and adding an if statement to return True or False fixed it.

I changed it from this:

```
legalPlay Nothing _ = True
legalPlay (Just (Card pileRank _)) (Card cardRank _)
| cardRank == R2 || cardRank == R8 || cardRank == R10 = True
| cardRank >= pileRank = True
| pileRank == R7 && cardRank <= pileRank = True
| otherwise = False
```

To this:

```
legalPlay Nothing _ = True
legalPlay (Just (Card pileRank _)) (Card cardRank _)
| pileRank == R7 = cardRank <= pileRank
| cardRank == R2 || cardRank == R8 || cardRank == R10 = True
| cardRank >= pileRank = True
| otherwise = False
```

CRITICAL REFLECTION

I found this quite hard to get started on, as there were a lot of functions to implement, and as the structure was predetermined, I couldn't implement it how it felt natural to me. To get my head around it, making the top down diagram helped, so I would definitely do that again, however this did leave out a lot of helper functions that I ended up implementing, so perhaps some more time to think about overlap between the functions would have been beneficial.

I think that the hardest challenge at the start was probably that it was released before we had covered the necessary content to get further than about step 1, as states were one of the last things covered, which meant that I had to teach them to myself, which definitely took up some time. Like most things in this module though, they were hard to learn, but when I got my head around contexts, binding and modify, it was reasonably quick. However, due to this being one of the last things that was taught, there wasn't much time left to actually implement the solution, which lead to less time to test, and a program which is more prone to error.

When I was testing functions, obviously most things didn't work first time, but I found it incredibly hard to narrow down where bugs were coming from, as ghc error messages are vague to say the least. I could deal with them until step 3 functions, where they became more complex, and had to deal with monads more, which made me get fustrated, and find out about trace and traceM, which let me output to the terminal far more easily, and debugging considerably easier as I could check values and the execution flow to track down bugs. Had I learned about this earlier, I believe that development would've been far quicker, and the program would've been more bug free.

Something else that I found was Hlint, which let me adhere to the style guidelines of Haskell far easier. I ran it on my code at the end. This an traceM made my research into debugging Haskell very worth it, as it sped up my debugging, which proves that research before you start is worth it. Looking back, I wish I had done it earlier.

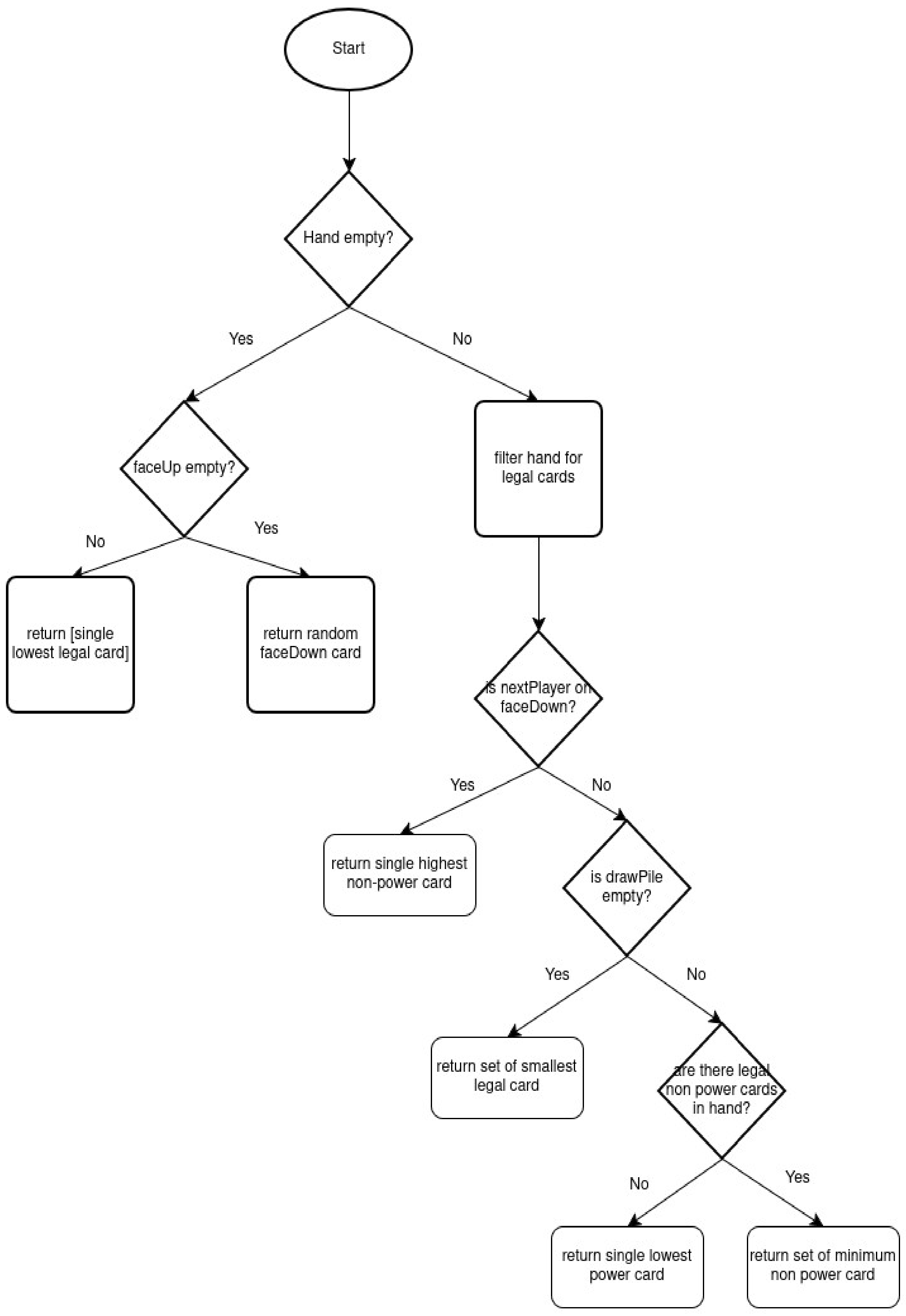
In terms of what went well, I think most of my functions take advantages of the benefits of haskell, namely list comprehensions and the ease of recursion to make short, simple and reusable functions, with a high degree of cohesion. Once I figured out how states work it became far easier to program, so next time I would learn that earlier.

I think my programming could definitely benefit from being slowed down, and giving more time to think about the solution to the function, as a lot of the time I would just jump in and write, which led to me missing simple things – for example I forgot to remove the card from the players hand in stealCard – which could have been avoided by planning. Another example of this is smart strategy, I started from scratch, but it would've been better to improve upon what was already there, using basic strategy. I also think that this could reduce the amount of nesting in some functions, meaning better code.

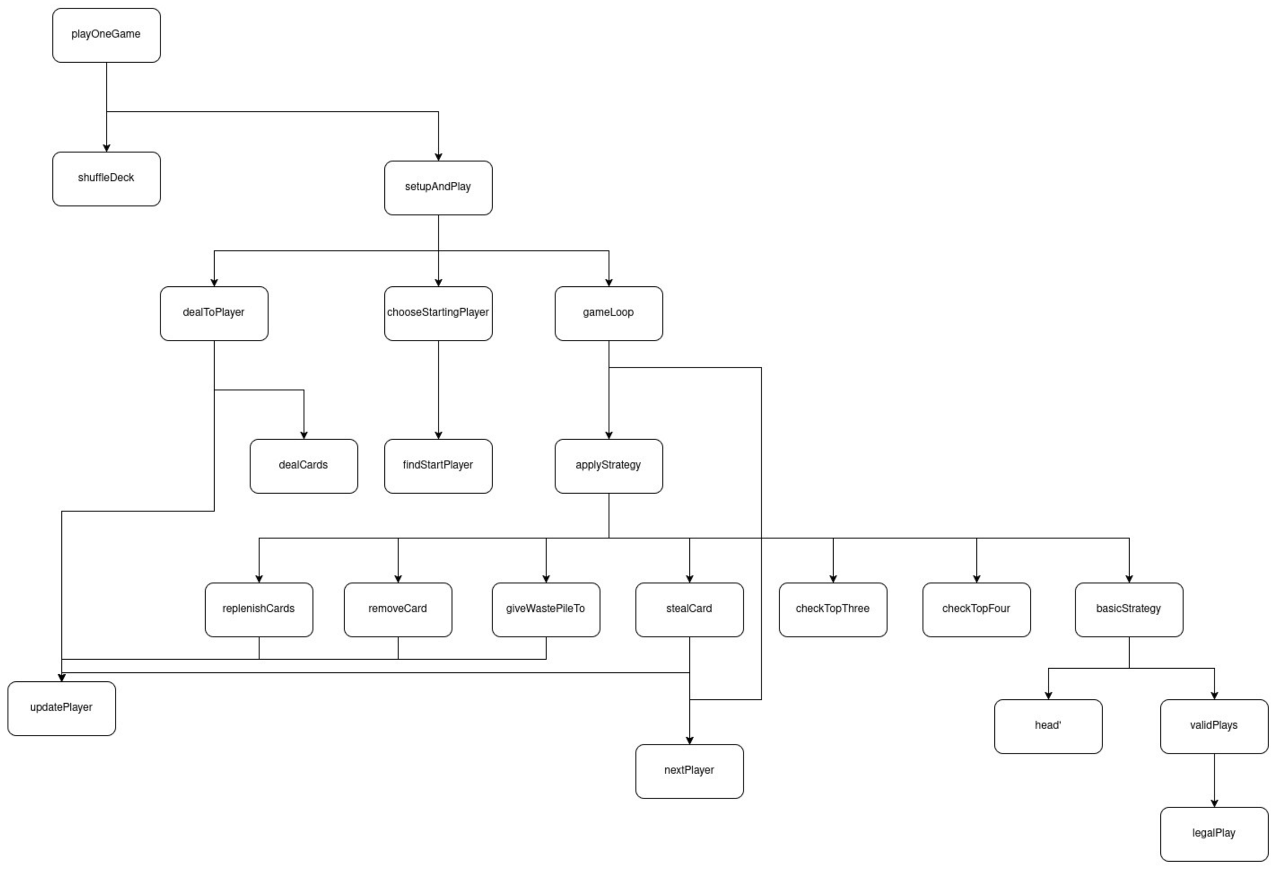
However, given the time defecit, I felt I came up with a good solution, particularly the smartStrategy, as I feel like winning nearly half the time in such a luck driven game is good.

APPENDIX

Appendix 1: Smart strategy


Start
Hand empty?
Yes
No
faceUp empty?
No
Yes
return [single lowest legal card]
return random faceDown card
filter hand for legal cards
is nextPlayer on faceDown?
Yes
No
return single highest non-power card
is drawPile empty?
Yes
No
return set of smallest legal card
are there legal non power cards in hand?
No
Yes
return single lowest power card
return set of minimum non power card

Appendix 2: playOneGame



No generative AI tools were used in the preparation of the solution to this work