COM2108: Functional Programming

Report for Dominoes Assignment

Year 2 (2023/24), Autumn Semester

Tamanna Mishra

Design

The specific game DomsMatch.hs replicates is fives-and-threes dominos. The rules for the game is as follows:

- ➔ Each player starts with a hand of N dominos.
- ➔ Players add a domino to the board i.e. 'dropping' a domino in turn and accumulate the fives-and-threes scores of their 'drops'.
- ➔ The player who drops the first domino gets the fives-and-threes score of its total spots.
- ➔ If a player does not have a domino that they can play on the current board, they 'knock' to indicate that they are skipping their turn. They cannot knock if they have a domino that they can play.
- ➔ Play continues until neither player can play (either because each player has run out of dominos or is knocking).
- ➔ If a player 'chips out' (plays a domino so that they have no dominos left in their hand) they score one point for this (on top of whatever they score for playing the tile).

After understanding the game and how it works, I analyzed the *scoring scheme*:
After a player has added a domino to the board, the total pips on the two open ends (the outermost tiles on the left and right ends) are summed up.

- ➔ If the sum is a multiple of 3, the player receives points equal to the sum divided by 3.
- ➔ If the sum is a multiple of 5, the player receives points equal to the sum divided by 5.
- ➔ Alternatively, if the sum equals 15, the player is awarded 8 points.
- ➔ For any other number that isn't a multiple of 3 or 5, nor equal to 15, the player receives 0 points.

Additionally, if the last tile in the player's hand is played, they receive an additional point on top of the score calculated from the sum of the pips on the two open ends.

Now that I had understood the concept, I decided to move on to working on the code.
To start off with, I looked at the 3 basic functions and noted down a basic definition of each.

a. **Scoreboard**
   Scoreboard returns the score based on the scoring scheme analyzed above of the board passed into the function as well as the Boolean value which indicates whether the domino just played is the last domino from the player's hand.
   I've used another function to award points according to the scoring scheme analyzed above.
   Function Signature:
   Board -> Bool (true if the domino just played is the last domino) -> Int (score)
   i. calcScore: given the sum of the pips on the outermost tiles on the board, calcScore returns the score in terms of the fives-and-threes dominoes scoring scheme.
      Function Signature: Int (raw score) -> Int (score according to fives-and-threes dominoes)
b. **Blocked**
   The blocked function takes the player's hand and a board state and indicates whether any tiles in the hand can be played on the board or not.
   Function Signature:
   Hand -> Board -> Bool (true if any tiles in hand can be played on the board)

c. **playDom**

The playDom function plays a given domino at the given end if it is possible to play it there and returns a Maybe Board – it returns the new Board state if the domino can be played, otherwise Nothing if the domino can't be played.

Function Signature:

Player -> Domino -> Board -> End -> Maybe Board (potential board state if domino is played)

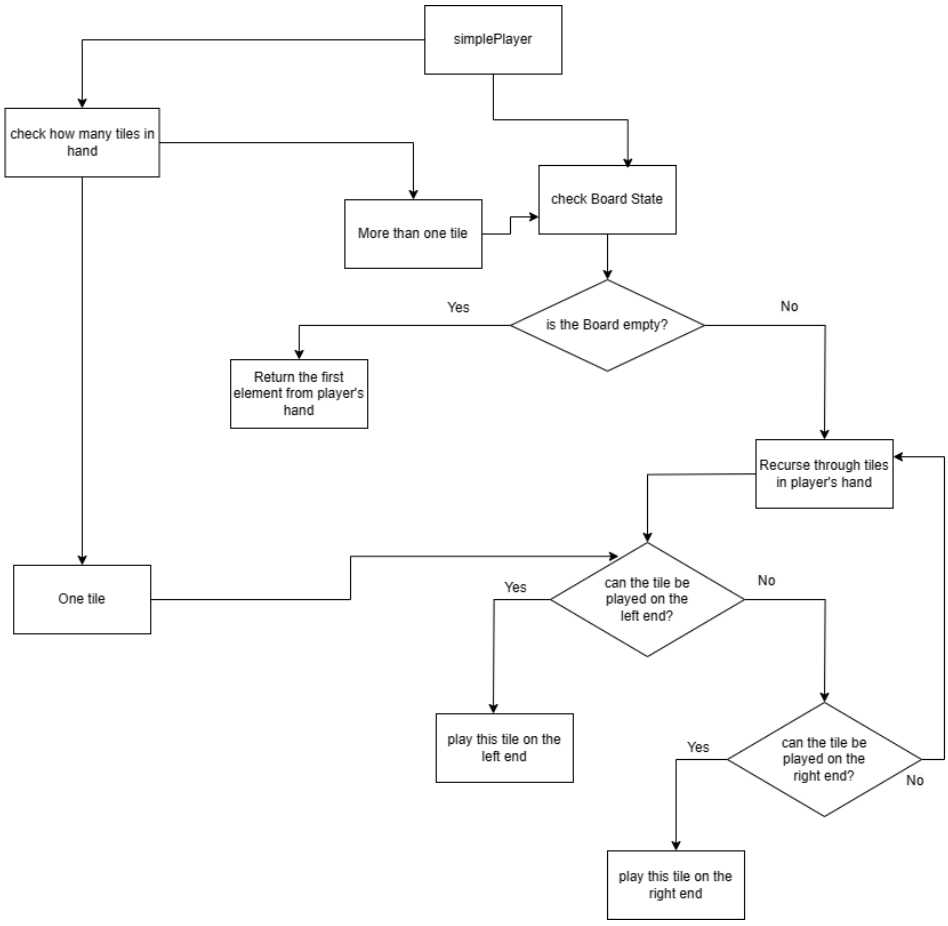
After implementing the scaffolding functions, I worked on implementing the SimplePlayer.

SimplePlayer

simplePlayer is a player without any "cleverness" and doesn't use strategies. It takes a hand of dominoes, a board state, the current player, and the current scores, and returns a domino to play and the side to play it on.

Cases implemented:

1) If the board is empty, play the first tile in hand.
2) If there's only one tile in hand, play that tile irrespective of board state.
3) If the board isn't empty, play the first "playable" domino at whichever end it can be played.



The next task was to implement a SmartPlayer, a player with some sort of “cleverness”, which would be demonstrated through implementation of at least three strategies.

smartPlayer
smartPlayer takes a hand of dominoes, a board state, the current player, and the current scores, and returns a domino to play and the side to play it on.

It considers *3 different scenarios (i.e., strategies*):

1) If the board is empty, decide the first drop/move.
2) If there's only one tile in hand, play that tile on whichever end possible. If it's possible to play on both ends, it uses simulatePlayAndGetScore and plays on the highest-scoring end.
3) It checks the list of tiles playable at the left end and at the right end, checks which among both them produces the highest score, and plays the highest scoring tile.

Functions used:

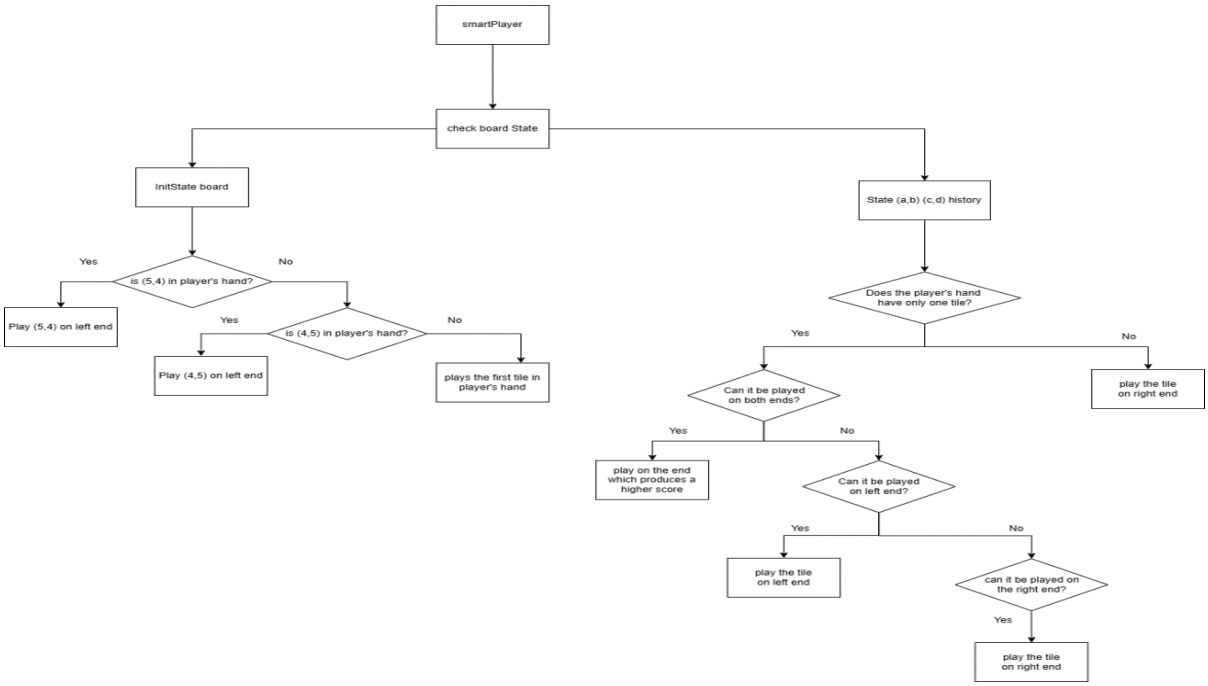
- ➔ domInHand: check if a particular domino is contained within a hand
  Function Signature: Domino -> Hand -> Bool
- ➔ canPlay: Given a domino, an end to play it at and a board state, canPlay checks if the domino can be played at the given end.
  Function Signature: Domino -> End -> Board -> Bool
- ➔ simulatePlayAndGetScore: Given the current player, a domino that can be played at both ends and the current board, the function simulates playing the tile on both ends and compares the scores gained from playing on left end and on right end. It returns a 3 part tuple consisting of the domino, the end at which playing the tile would produce a higher score and the score produced.
  Function Signature: Player -> Domino -> Board -> (Domino,End,Int)
- ➔ possPlays: Given the player's hand and the board, possPlays returns a 2 part tuple consisting of a list of dominos that can be played at the left end and of a list of dominos that can be played at the right end.
  Function Signature: Hand -> Board -> ([Domino],[Domino])
- ➔ playTheseTiles: Given a list of 2 part tuples consisting of a domino and the end to play it at and the board, it returns a list of 3 part tuples consisting of the domino, end and the score produced from playing the domino.
  Function Signature: [(Domino,End] -> Board -> [(Domino,End,Int)]
- ➔ sortScores: Given a list of 3 part tuples consisting of a domino, end and score, sortScores sorts the list based on the score in descending order (highest to lowest).
  Function Signature: [(Domino, End, Int)] -> [(Domino,End, Int)]

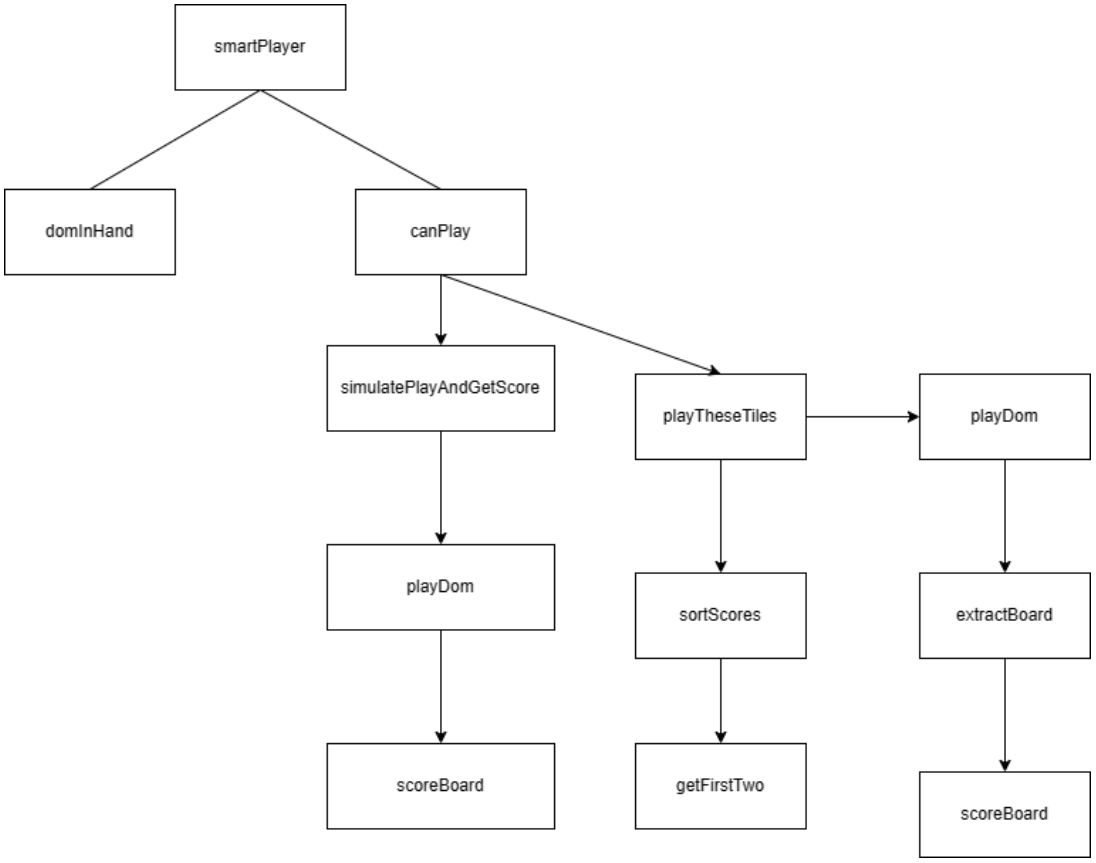
Helper functions:

- ➔ getFirstTwo: returns the first two elements of a 3 part tuple.
  Function Signature: (a,b,c) -> (a,b)
- ➔ extractBoard: Given a Maybe Board, extractBoard returns just the board.
  Function Signature: Maybe Board -> Board

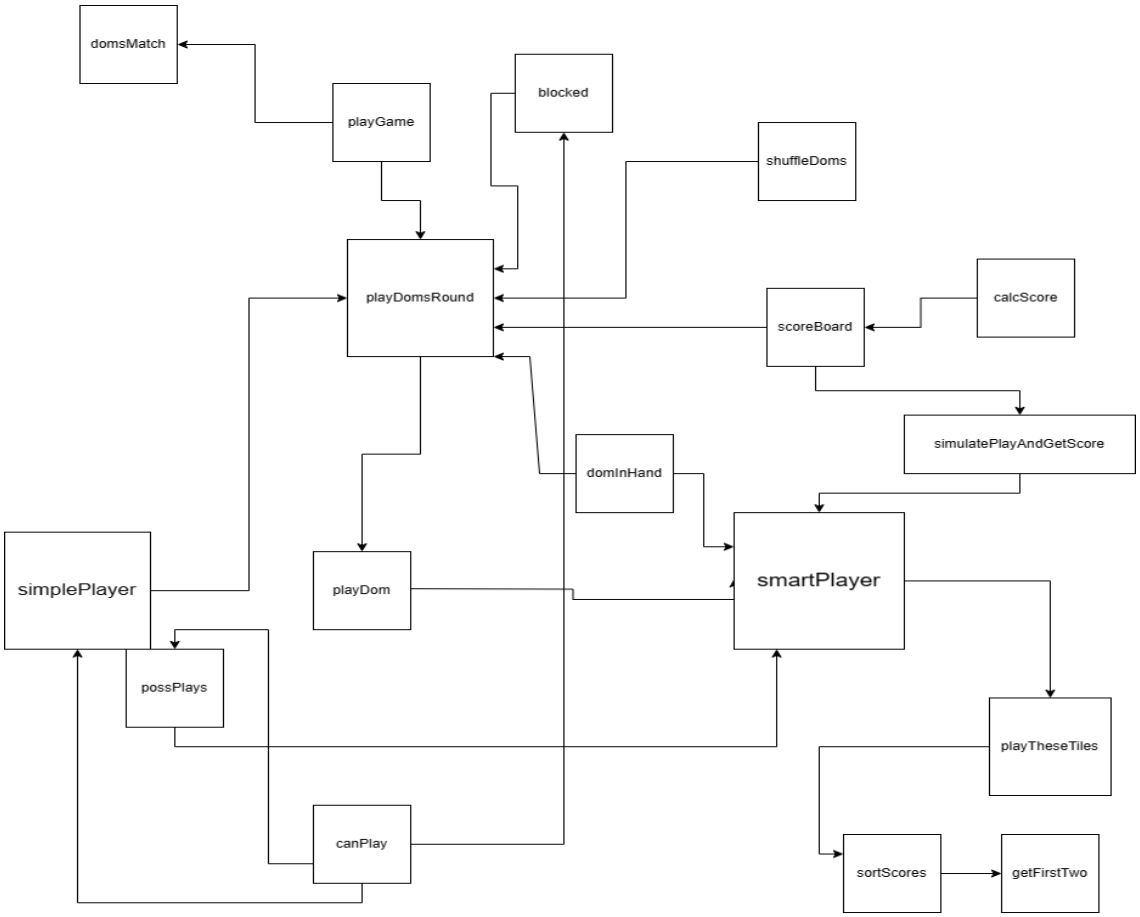
Since the implementation of a function is to be done in a top-down manner, I’ve made a flowchart for organization and to keep track of which functions are needed in order to implement the strategies I’ve decided to add to the smartPlayer.

Function Decomposition/Flowchart:





Here is a flowchart showcasing the overall implementation of the program.



Testing

This section is for including test data for each function implemented in the program and explaining the rationale behind choosing the testing data as well as the output produced.
The order in which the functions have been listed under this section represents the order in which I have implemented them in the actual code.

calcScore

Inputs	Output	Reason
Int	**Int**	**All outputs are returned because of scoring scheme in 5s-and-3s dominoes.**
15	8	15 is composed of five 3s and three 5s, thus 8 is returned.
5	1	5 is composed of one 5, thus 1 is returned.
9	3	9 is composed of three 3s, thus 3 is returned.
22	0	Any score not divisible by 3 or 5 nor both will cause an output of 0 to be returned.

scoreBoard

Inputs		Output	Reason
Board	**Bool**	**Int**	
InitState	False	0	If there's an attempt to find the score of an empty board, the score returned is 0 since there are no dominoes.
(State (1,2) (3,4) [])	True	2	The pips on the outermost ends are added, which is then converted to the 5s-and-3s score, and then 1 is added to this score because the Boolean value indicates that the last domino in hand is being played.
(State (4,2) (3,1) [((1,1),P2,1)])	False	1	The pips on the outermost ends are added, which is then converted to the 5s-and-3s score and then returned as output.

Blocked

Inputs		Output	Reason
Hand	**Board**	**Bool**	
[] (empty hand)	(State (1,2) (3,4) [])	True	Since the hand is empty, True is returned to indicate that no domino can be played from the player's hand, irrespective of the board.
[(3,4),(1,2),(4,5),(5,4)]	InitState	False	Since the board is empty, False is returned to indicate that any domino from the player's hand can be played.
[(1,2)]	(State (2,3) (4,1) [])	False	False is returned because there's a domino that can be played from the player's hand based on the board state.

PlayDom

Inputs				Output	Reason
Player	**Domino**	**Board**	**End**	**Maybe Board**	
P1	(1,2)	InitState	L	Just (State (1,2) (1,2) [((1,2),P1,1)])	Since the board is empty, any domino can be played and thus the board gets updated to have (1,2) on both ends.
P2	(2,3)	(State (1,2) (3,4) [])	R	Nothing	Since (2,3) can't be played on the given board, Nothing is returned.
P1	(1,2)	(State (1,2) (3,4) [((1,2),P2,1)])	L	Just (State (2,1) (3,4) [((1,2),P2,1),((1,2), P1,2)])	Since (1,2) can be played on the left end of the board, a Just Board output is returned.

simplePlayer

Inputs				Output	Reason
Hand	**Board**	**Player**	**Scores**	**(Domino,End)**	
[(4,5),(3,4)]	InitState	P1	(21,15)	((4,5),L)	The board is empty so the first tile in hand will be played at the left end.
[(2,1),(3,4)]	(State (4,3) (2,1) [])	P1	(0,0)	((2,1),R)	Since the first tile in hand is playable at the right end of the given board, it is played at the right end.
[(1,2)]	(State (2,1) (1,5) [])	P2	(17,15)	((1,2),L)	There's only one tile in hand so we check if it can be played at either left or right end. It's playable at left end.

canPlay

Inputs			Output	Reason
Domino	**End**	**Board**	**Bool**	
(1,2)	L	InitState	True	Since the board is empty, any domino can be played so True is returned.
(3, 4)	R	State (2, 3) (4, 5) [((2, 3), P1, 1), ((4, 5), P2, 2)]	False	(3,4) cannot be played on right end of the board, thus False is returned.
(5,5)	L	State (5,1) (2, 4) [((5,1), P1, 1)]	True	(5,5) can be played on the left end of the board, thus True is returned.

possPlays

Inputs		Output	Reason
Hand	**Board**	**([Domino],[Domino])**	
[(6, 6), (3, 3), (2, 2)]	InitState	([(6,6),(3,3),(2,2)],[(6,6),(3,3),(2,2)])	Since the board is empty, the tiles in hand can be played on either hand and thus, both lists are the same.
[(1, 2), (3, 4), (5, 6)]	(State (2, 3) (4, 5) [])	([(1,2)],[(5,6)])	(1,2) can be played on left end and (5,6) can be played on right end, thus the left list has (1,2) and the right list has (5,6).
[(4, 5), (2, 3), (1, 6)]	State (3, 5) (1, 2) [((1, 2), P1, 1)])	([(2,3)],[(2,3)])	(2,3) is the only playable tile for the current board and on both ends as well so it is in both lists.

simulatePlayAndGetScore

Inputs			Output	Reason
Player	**Domino**	**Board**	**(Domino,End,Int)**	**This function is only called when there's one tile in hand and when this tile is playable on both ends of the board.**
P1	(1,2)	InitState	((1,2),L,2)	Since the board is empty, (1,2) is played on the left end and the score is 2 because the board is updated as State (1,2)(1,2) and there's only one tile in hand.
P1	(4, 3)	State (3, 4) (6, 4) [((3, 4), P2, 1), ((6, 4), P1, 2)]	((4,3),R,3)	(4,3) can be played on both ends but playing on right end gives a higher score so it is played on the right end.
P2	(2, 4)	State (2, 3) (5, 2) [((2, 3), P1, 1), ((5, 2), P2, 2)]	((2,4),L,3)	Playing (2,4) on the left end produces a higher score so it is played on the left end.

Critical reflection

My experience navigating Haskell was indeed challenging. In the beginning, it felt like I was relearning programming from scratch and the transition from considering solutions to implementing them in Haskell proved to be quite a drastic change in difficulty.

Despite my preference for object-oriented programming (OOP), this module broadened my perspective, strengthening my fundamentals. The absence of common OOP features such as for loops, mutable variables, inheritance, etc. required me to acquire a good understanding of recursion and other workarounds which reshaped how I came up with solutions for problem-solving in Haskell.

It emphasized the significance of adaptability and flexibility in approaching programming problems. I believe that my patience while debugging and experimenting to trace errors as well as constantly looking at the documentation played a huge role in increasing my understanding of Haskell.

After not achieving the required threshold mark, I questioned my proficiency and understanding of Haskell.

It seemed daunting to me at the time, and I couldn't understand what I was doing wrong until I realized that I was using the same approach as I had used for OOPs. So, I decided to review the lecture slides again to re-learn the language efficiently by identifying the concepts I lacked an understanding of.

After putting time and effort into this assignment, I feel that I am now well-equipped to pass the second attempt of the quiz. Moving forward, I plan to use these experiences to tackle similar challenges in the future.