COM2108 Functional Programming Grading Assessment

Sam Taseff

Design

The red underlined text in the design section represents identified function decompositions, many were decided in this document, others came naturally from implementing the solution - these are explained with comments and later on in the testing section.

scoreBoard

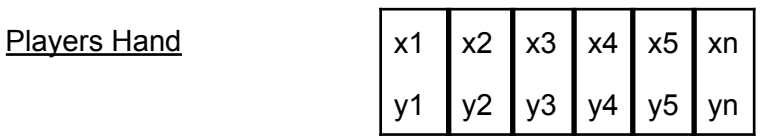
This function will implement the 5s and 3s scoring algorithm for the two outer domino pip values on the board.

Plan

- If the board state is *InitState,* then the score will be 0 (No dominoes)
- Otherwise:
    - Calculate the total pips on the outside of the domino structure - the pattern match can just capture the left-most pips of the left domino and the right-most pips of the right domino and ignore all else
    - Check if this total is a multiple of 5 or 3
    - Calculate how many 5s and 3s are contained wholly within the total
    - If it is a multiple of 5, add the number of 5s in the total to the score
    - Do the same with 3
    - If it is the final domino, add 1 to the score
- Return the score

blocked

This function takes a hand of dominoes and a board and determines if the player can make a move. This can be thought of algebraically as follows:



- $(a \in X) \vee (b \in X) \vee (a \in Y) \vee (b \in Y) \iff "The player is not blocked"$
- This means we can flatten the list of dominoes in the player's hand into a list of pips, and then just check if either the left or right most number of pips on the board are elements contained within.

Plan

- If the board is in *InitState,* then return False, as the player may play any domino from their hand
- Otherwise:
    - Pattern match the leftmost pips and rightmost pips from the board type
    - Fold the hand into a flattened list of pips
    - Check if the leftmost pips or the rightmost pips appear in the flattened list

playDom

This function plays a domino on a board state and returns the resulting state, if it cannot be played, it returns Nothing.

Plan

- If the board is in the *InitState,* then return a board state with the domino as both the left and right domino with history containing one entry with the relevant parameters from the input. This duplication will function the same as if there is just one domino on the board.
- Otherwise:
    - If the domino can be played on this end of the board:
        - Orient the domino correctly depending on which end and the corresponding pips
        - Return a *State* with the relevant end replaced with the domino being played, as well as having it appended to the history list.
    - Otherwise, return *Nothing*

I will need to implement a *canPlay* and *orientDomino* function for use in my plan for *playDom.*

canPlay

This function takes a domino, an end, and a board and decides if the domino can be played there.

This function will just check if either of the pips on the domino equate to the pips on the specific end of the domino board, if it happens to be *InitState,* then it will return True.

After implementing this function, I realized I could update my implementation of blocked to make use of this function for each end of the board - but I decided against this, as my current implementation felt concise and obvious enough.

flipDomino

This function makes the desired side (pips) of a domino face a certain direction or perspective. This can be implemented with two guards for each case, with if statements for each direction.

simplePlayer

The simple player will just find a legal domino in its hand and play it, with no logic otherwise.

To achieve this, I need to generate two lists, every domino I can place on the left side and every domino I can place on the right, using the player's current hand. For this task, I will need to calculate every domino that can be played on either side of the board, this could be done using a list comprehension with the *canPlay* function, but I have a feeling that I will need this for the *smartPlayer*, so I will implement it as a separate function - *possPlays.*

possPlays

This function can be implemented using two filters on the hand for the left and right sides of the board, with the *canPlay* function.

smartPlayer

The smart player must select from a number of different strategies each turn, depending on the game state. I have developed three different strategies, listed below in the order of precedence they take during selection.

To the right, you can see a visual representation of when each strategy comes into play during a game.

Situation	Signifiers	Strategy
The player is close to winning: *The finishing strategy*	The player's score is within a potential play of 61. This can be calculated with: *currentScore + averagePipsInHand * winSensitivity >= 61*	Check for a domino that can produce the winning score when played on the board, if there isn't one, find a domino that has the highest chance to make the opponent knock (as it is not possible to win this turn).
The opponent is close to winning: *The blocking strategy*	The opponent's score is within a potential play of 61. This can be calculated with: *currentOpponentScore + averagePipsInOpponentHand * lossSensitivity >= 61*	Calculate the domino with the highest probability to block the opponent.
"Midway" through the game: *The midway strategy*	Otherwise	To calculate the best possible play with the given hand, I can assign some score to each domino and pick the highest scoring one. The logic will be in how each domino is scored, here is a function I have come up with: *dominoScore = (HowMuchTheyScoreMe * weight1 - MaximumScoreForOpponent * weight2)*

The Finishing Strategy

I know the *winSensitivity* value is positive and cannot go much above 2 or 3 before becoming useless, due to this very small range for a decent value, I manually tested the finished function using *domsMatch* games to get a value for it. This value turned out to be *2.1*

After this precondition is met, determining the domino that will win the game goes as follows:

- Given a hand, a board, the current score and a target score, determine if there is a play which achieves the target score given the current state of the game and return it, if there is none, block the opponent.

- To determine if there is a winning domino play:
  - Calculate the required score
  - Use the *possPlays* function to filter down the hand to playable dominoes
  - Get the board states from playing each of these dominoes
  - Filter the board states using the *scoreBoard* function to those which achieve the required score
  - Return *Nothing* if the list is empty, or the head if not
- The design behind blocking the opponent is expanded upon in the next section

The Blocking Strategy

Similar to the *winSensitivity,* I know the value for *lossSensitivity* is positive and cannot go above 2 or 3 before becoming useless. I manually tested the finished function using *domsMatch* games to get a value for it. This value turned out to be *0.3*

To play the domino with the highest chance of making the opponent knock:

- Given a hand, a board, the set of all possible dominoes (*domSet)* and the player (not the opponent):
  - Use the *possPlays* function to filter down the hand to playable dominoes
  - For each potential play:
    - Determine the board state after playing that domino, call this the board'
    - Calculate all the possible plays the opponent could make in response to this board'
      - Determine what dominoes the player *could* have:
        - Find out when the opponent had to knock from the history, get the list of pip values on the outer ends of the board that caused these knocks
        - Filter the set of *all* dominoes down, by removing those in the current player's hand, removing those played on the board', and removing those with pip values the opponent had to knock for
      - Filter this list to only dominoes that can be played on the board' using *canPlay*
    - Pair each play with the length of possible response plays from the opponent
  - Return the play with the lowest amount of response plays possible for the opponent

The Midway Strategy

- Take the current hand and board state and return a domino with a side to play it on:
  - Filter the hand into playable dominoes.
  - Iterate over the hand and, for each domino, calculate the best way to play it and determine the resultant score, multiply it by *weight1.*
    - To calculate the best way to play a domino
      - If the domino can be played on the left, calculate the resultant score from playing it on that side, otherwise, its score is -1
      - Do the same with the right side
      - Choose the side with the higher score
  - Subtract from each of these scores the best possible play the opponent could make in response to the board state created by playing each domino, this is multiplied by *weight2.*
    - To calculate the best play the opponent could make in response to playing this domino:
      - Calculate all the possible plays the opponent could make in response to this board'

            - Pick the play which scores the highest
        - The weighted *maximumScoreForOpponent* will then be subtracted from the maximum achievable score for each domino in the hand.
        - Then select the domino with the highest score.

The weights in the functions would need to be determined through testing. My methodology for this will be simplistic: I will set *weight1* to zero, then find the best *weight2* to accompany it - since this is only optimizing two variables, I should get a value for *weight2* that is correctly scaled with *weight1* being 1.

To find this weight value, I implemented a couple utility functions, each one utilizing the previous:

- *domsMatchRandomizedAverage:* Plays a number of *domsMatch* games across different seeds, then averages the wins for each player.
- *domsMatchRandomizedAverage:* Takes two players and returns the percentage difference in average wins between them.
- *testSmartPlayerWeight2:* Calculates the percentage difference in win rate for the smart player of a range of values for *weight2*, I can then pick the highest one and use that as pretty good value for *weight2*. I can rerun the function to converge on smaller and smaller ranges for the best value.

For the sake of brevity, I have left these functions out along with the test suite functions. The optimal value for weight2 turned out to be *-0.2*

Testing

Functions are listed in order of implementation.

The history array of a board state can be safely ignored (empty) for the following tests, since it is not necessary for their logic:

- *scoreBoard*
- *blocked*
- *canPlay*
- *possPlays*
- *bestEndToPlay*
- *winningDomino*
- *canPlayWithPip*

General Utility Functions

These functions didn't seem to merit testing, as they can be validated by simple inspection.

- *fst3/snd3/lst3:* Returns the desired component from a 3-tuple of any type
- *removeCopies*: Takes a list and removes every duplicate element
- *occurances:* Returns the number of times an element is contained within a list

DomsMatch Utility Functions

These functions can be confirmed to be correct since they simply return a value for every patten match, which are exhaustive.

- *getScore*: Extracts the score from a *Score* variable given a *Player*
- *getHistory*: Extracts the history list from a board state
- *leftDom*/*rightDom*: Gets the desired outer domino from a board state
- *opponent*: Given a *Player,* returns the opponent

scoreBoard

#	Description	Test Parameters	Expected Output
1	Empty Board	`InitState False`	`0`
2	A single 3 last	`(State (2,2) (2,1) []) True`	`2`
3	Multiple 5s coming last	`(State (6,1) (1,4) []) True`	`3`
4	Multiple 5s and 3s	`(State (5,3) (3,10) []) False`	`8`
5	Board with a no 3s or 5s possible	`(State (2,4) (4,0) []) False`	`0`
6	Edge case with 0 total dots	`(State (0,0) (0,0) []) False`	`0`
7	Double on one end	`(State (6,6) (2,3) []) False`	`8`
8	Playing one double at the start	`(State (3,3) (3,3) []) False`	`2`

blocked

#	Description	Test Parameters	Expected Output
1	Empty Board	`[(1,2), (7,3), (8,3)] InitState`	`False`
2	Blocked on both sides	`[(0,4), (3,9), (8,3)] (State (6,1) (1,5) [])`	`True`
3	Empty Hand	`[] (State (6,1) (1,5) [])`	`True`
4	Multiple options	`[(0,4), (3,9), (8,3)] (State (3,1) (1,4) [])`	`False`
5	Blocked on the left	`[(0,4), (5,9), (8,3)] (State (6,1) (1,5) [])`	`False`
6	Blocked on the right	`[(6,4), (5,6), (8,3)] (State (6,1) (1,2) [])`	`False`
7	One option, two sides	`[(0,4), (5,2), (8,3)] (State (2,1) (1,2) [])`	`False`

canPlay

#	Description	Test Parameters	Expected Output
1	Empty Board (Should be true for every value of the first two parameters)	`(4,2) L InitState`	`True`
2	Domino used on the wrong side	`(5,2) L (State (3,3) (4,5) [])`	`False`
3	Domino used on the correct side	`(5,2) R (State (3,3) (4,5) [])`	`True`
4	Flipping the domino in the test above	`(2,5) R (State (3,3) (4,5) [])`	`True`
5	Unplayable domino (Pips on the inside match, however) - Left	`(2,1) R (State (6,1) (2,3) [])`	`False`
6	Trying the above test on the right side	`(2,1) L (State (6,1) (2,3) [])`	`False`

orientDomino

This function is simple enough to test every single possible case.

#	Description	Test Parameters	Expected Output
1	Flipping the value of 4 from the left	`4 L (4,2)`	`(2,4)`
2	Flipping the value of 4 from the right	`4 R (4,2)`	`(4,2)`
3	Flipping the value of 2 from the left	`2 L (4,2)`	`(4,2)`
4	Flipping the value of 4 from the right	`2 R (4,2)`	`(2,4)`
5	Flipping a non-existent value	`8 R (4,2)`	`An exception`

playDom

Here, I focused on if the function correctly responded to the *End* parameter in every case as this would greatly affect its outputs with respect to the history and board state, it should accept no illegal moves due to its use of the *canPlay* function as a guard to the rest of the logic. I am also only considering the last entry in the history array for testing rather than a true representation of the board history for some tests that do not necessitate it.

#	Description	Test Parameters	Expected Output
1	Empty board	`P1 (5,4) InitState L`	`Just (State (5,4) (5,4)` `[((5,4), P1, 0)])`
2	Impossible play (only on the left side)	`P1 (5,4)` `(State (3,2) (2,4) []) L`	`Nothing`
3	Legal right play (works both sides)	`P1 (2,1)` `(State (2,0) (0,1)` ` [((2,0), P1, 0),` `  ((0,1), P2, 1)]) R`	`Just (State (2,0) (1,2)` `[((2,0), P1, 0)` `  ((0,1), P2, 1),` `  ((1,2), P1, 2)])`
4	Legal left play (works on both sides)	`P1 (2,1)` `(State (2,0) (0,1)` ` [((2,0), P1, 0),` `  ((0,1), P2, 1)]) L`	`Just (State (1,2) (0,1)` `[((1,2), P1, 2),` `  ((2,0), P1, 0)` `  ((0,1), P2, 1)])`

possPlays

#	Description	Test Parameters	Expected Output
1	Empty board	`[(1,2), (3,5)] InitState`	`([(1,2), (3,5)],` ` [(1,2), (3,5)])`
2	Potential on both sides	`[(1,2), (3,5), (4,1)]` `(State (1,6) (4,3) [])`	`([(1,2), (4,1)],` ` [(3,5)])`
3	Overlapping potential	`[(6,4), (4,5), (4,1)]` `(State (6,6) (4,4) [])`	`([(6,4)],` ` [(6,4), (4,5), (4,1)])`
4	No plays	`[(6,4), (4,5), (4,1)]` `(State (0,6) (4,0) [])`	`([],` ` [])`
5	Small number of plays	`[(6,4), (4,5), (4,1)]` `(State (0,6) (3,1) [])`	`([],` ` [(4,1)])`

simplePlayer

Some basic testing in *GHCi* shows the *simplePlayer* to be functional. Since it's random, I'm looking for roughly equal wins and no exceptions.

```
Unset
ghci> domsMatch 200 7 61 simplePlayer simplePlayer 7777
(111,89)

-- Lower target score
ghci> domsMatch 200 7 32 simplePlayer simplePlayer 7777
(95,105)

-- Higher number of dominos in hand
ghci> domsMatch 200 10 32 simplePlayer simplePlayer 7777
(97,103)
```

getAveragePip

#	Description	Test Parameters	Expected Output
1	All the same	`[(4,4), (4,4), (4,4)]`	`4.0`
2	An average I can do in my head	`[(6,5), (5,4), (6,4)]`	`5.0`
3	A fractional average	`[(6,5), (1,1), (6,2)]`	`3.5`

sequentialHistory

#	Description	Test Parameters	Expected Output
1	Start of the game	`[]`	`[]`
2	Plays on the left and right	`[((1,5), P1, 2),` ` ((5,4), P1, 0),` ` ((4,4), P2, 1)]`	`[((5,4), L, P1, 0),` ` ((4,4), R, P2, 1),` ` ((1,5), L, P1, 2)]`
3	Only playing on the right	`[((5,4), P1, 0),` ` ((4,4), P2, 1),` ` ((4,5), P1, 2)]`	`[((5,4), L, P1, 0),` ` ((4,4), R, P2, 1),` ` ((4,5), R, P1, 2)]`
4	Only playing on the left	`[((0,3), P1, 2),` ` ((3,1), P2, 1),` ` ((1,1), P1, 0)]`	`[((1,1), L, P1, 0),` ` ((3,1), L, P2, 1),` ` ((0,3), L, P1, 2)]`

buildBoardState

Here I test if I can get the complete original board state just from its history.

#	Description	Test Parameters (GHCi prompt)	Expected Output
1	Plays on the left and right	`(buildBoardState.sequentialHistory.getHistory)` ` (State (1,5) (4,4)` `  [((1,5), P1, 2),` `  ((5,4), P1, 0),` `  ((4,4), P2, 1)])`	`(State (1,5) (4,4)` ` [((1,5), P1, 2),` `  ((5,4), P1, 0),` `  ((4,4), P2, 1)])`
2	Only playing on the right	`(buildBoardState.sequentialHistory.getHistory)` ` (State (5,4) (4,5)` `  [((5,4), P1, 0),` `  ((4,4), P2, 1),` `  ((4,5), P1, 2)])`	`(State (5,4) (4,5)` ` [((5,4), P1, 0),` `  ((4,4), P2, 1),` `  ((4,5), P1, 2)])`
3	Only playing on the left	`(buildBoardState.sequentialHistory.getHistory)` ` (State (0,3) (1,1)` `  [((0,3), P1, 2),` `  ((3,1), P2, 1),` `  ((1,1), P1, 0)])`	`(State (0,3) (1,1)` ` [((0,3), P1, 2),` `  ((3,1), P2, 1),` `  ((1,1), P1, 0)])`

opponentKnocks

#	Description	Test Parameters	Expected Output
1	No-one has knocked yet (P1 checking for P2 knocks)	`P1 (State (3,4) (4,5) [((3,4), P1, 2), ((4,4), P1, 0), ((4,5), P2, 1)])`	`[]`
	No-one has knocked yet (P2 checking for P1 knocks)	`P2 (State (3,4) (4,5) [((3,4), P1, 2), ((4,4), P1, 0), ((4,5), P2, 1)])`	`[]`
2	Both players have knocked (P1 checking for P2 knocks)	`P1 (State (0,3) (5,2) [((0,3), P1, 3), ((3,4), P2, 2), ((4,4), P1, 0), ((4,5), P2, 1), ((5,2), P1, 4)])`	`[5,0]`
	Both players have knocked (P2 checking for P1 knocks)	`P2 (State (0,3) (5,2) [((0,3), P1, 3), ((3,4), P2, 2), ((4,4), P1, 0), ((4,5), P2, 1), ((5,2), P1, 4)])`	`[5,4]`
3	Only one person knocked (P1 checking for P2 knocks)	`P1 (State (0,3) (5,2) [((3,4), P2, 2), ((4,4), P1, 0), ((4,5), P2, 1)])`	`[]`
	Only one person knocked (P2 checking for P1 knocks)	`P2 (State (0,3) (5,2) [((3,4), P2, 2), ((4,4), P1, 0), ((4,5), P2, 1)])`	`[5,4]`

opponentsPossibleDominoes

#	Description	Test Parameters (GHCi prompt)	Expected Output
1	No dominoes played, no dominoes in my hand	`opponentsPossibleDominoes [] InitState domSet P1`	`domSet`
2	No dominoes played yet - Start of the game	`let hand = [(1,1), (1,0), (0,0), (2,2)] answer = filter (\d -> not ((d `elem` hand))) domSet in opponentsPossibleDominoes hand InitState domSet P1`	`answer`
3	What could P2 have, if they haven't knocked at all?	`let hand = [(1,1), (1,0), (0,0), (2,2)] board = (State (0,3) (5,2) [((0,3), P2, 3), ((3,4), P1, 2), ((4,4), P1, 0), ((4,5), P2, 1), ((5,2), P1, 4)]) answer = filter (\d -> not ((d `elem` hand) || (d `elem` (map (\(d, p, m) -> d) (getHistory board))))) domSet in opponentsPossibleDominoes hand board domSet P1,`	`answer`

4	What could P2 have if they have knocked?	`let hand = [(1,1), (1,0), (0,0), (2,2)] board = (State (0,3) (5,2) [((0,3), P1, 3), ((3,4), P2, 2), ((4,4), P1, 0), ((4,5), P2, 1), ((5,2), P1, 4)]) answer = filter (\d -> not ((d `elem` hand) || (d `elem` (map (\(d, p, m) -> d) (getHistory board))))) domSet in length (opponentsPossibleDominoes hand board domSet P1)`	`< length answer`

bestEndToPlay

#	Description	Test Parameters	Expected Output
1	Start of the game (will always prefer *L*)	`bestEndToPlay (2,3) InitState`	`(L, 1)`
2	Two possible ends	`bestEndToPlay (2,3) (State (3,3) (1,2) [])`	`(R, 3)`
3	Only one possible end	`bestEndToPlay (2,3) (State (3,3) (0,1) [])`	`(L, 1)`

possibleOpponentPlays

#	Description	Test Parameters (GHCi prompt)	Expected Output
1	If P1 has most of a certain pip in their hand, what would the other player have?	`board = State (3,5) (4,1) [((3,5),P2,3), ((5,4),P1,0), ((4,4),P2,1), ((4,1),P1,2)]` `hand = [(5,0), (5,1), (5,2), (5,5), (3,0), (3,1)]` `-- Opponent's possible dominoes: [(0,0),(1,0),(1,1),(2,0),(2,1),(2,2),(3,2), (3,3),(4,0),(4,2),(4,3),(5,3),(6,0),(6,1), (6,2),(6,3),(6,4),(6,5),(6,6)]` `possibleOpponentPlays hand board domSet P1`	Valid plays from the opponent's possible hand of dominoes: `[((1,0), R, 1), ((1,1), R, 1), ((2,1), R, 1), ((3,2), L, 1), ((3,3), L, 0), ((4,3), L, 1), ((5,3), L, 2), ((6,1), R, 3), ((6,3), L, 0)]`
2	What if there are no plays for the opponent?	`board = State (0,5) (4,0) [((0,5),P1,3), ((5,4),P1,0), ((4,4),P2,1), ((4,0),P1,2)]` `hand = [(3,0), (2,0), (1,0), (0,0), (6,0), (3,1)]` `-- Opponent's possible dominoes: [(1,1),(2,1),(2,2),(3,2),(3,3),(4,1),(4,2), (4,3),(6,1),(6,2),(6,3),(6,4),(6,6)]` `possibleOpponentPlays hand board domSet P1`	`[]`

3	On a controlled set of dominoes, calculate the possible hand of the opponent which is known	`board = State (1,0) (0,0)` `[((1,0), P1, 0), ((0,0), P2, 1)]` `hand = [(3,1), (2,1), (1,0), (1,1)]` `domSet' = [(1,1), (1,0), (0,0), (2,2),` `(2,1), (2,0), (3,3), (3,2), (3,1), (3,0)]` `-- Opponent's possible dominoes:` `[(2,2), (2,0), (3,3), (3,2), (3,0)]` `possibleOpponentPlays hand board domSet' P1`	`[((2,0), R, 1),` `((3,0), R, 0)]`

possibleResponsePlays

Due to this function simply playing a domino on a board state and then returning the output of *possibleOpponentPlays* on that resultant board state with the resultant hand, I feel I can safely skip this function due to all of its sub-functions already being completely verified. The testing would look much the same as above.

bestResponsePlay

This function just takes the output from *possibleOpponentPlays* and picks the entry with the highest score value. If the output list is empty, it returns *Nothing,* otherwise it returns *Maybe (Domino, End, Int).* Due to the simplicity of this function, I believe I can also safely omit its testing - it would simply be picking the highest scoring entry out of the output from the *possibleOpponentPlays* testing above. This could be inlined, but I believe the program benefits from the readability.

winningDomino

#	Description	Test Parameters	Expected Output
1	No winning domino	`[(3,0)]` `(State (0,1) (5,4) []) 30 33`	`Nothing`
2	A winning play with other valid plays	`[(3,0), (5,0), (0,0)]` `(State (0,1) (5,4) []) 30 33`	`Just ((5,0), L)`
3	Multiple winning plays, including a double	`[(3,3), (5,0), (0,3)]` `(State (0,1) (5,3) []) 30 32`	`Just ((0,3), L)`
4	Multiple winning plays, including a double - the other option	`[(3,3), (5,0)]` `(State (0,1) (5,3) []) 30 32`	`Just ((3,3), R)`

canPlayWithPip

#	Description	Test Parameters	Expected Output
1	Start of the game	`1 (InitState) (1,0)`	`(True, True)`
2	Start of the game, alternate pip	`0 (InitState) (1,0)`	`(True, True)`
3	Cannot play on either side	`0 (State (0,3) (4,0) []) (1,0)`	`(False, False)`
4	Can play on both sides	`1 (State (0,3) (4,0) []) (1,0)`	`(True, True)`
5	Can play only on one side	`1 (State (0,3) (4,1) []) (1,0)`	`(True, False)`
6	Can play only on the other side	`4 (State (0,3) (4,1) []) (1,4)`	`(False, True)`

blockingPlay

#	Description	Test Parameters (GHCi prompt)	Expected Output
1	If P1 has most of a certain pip in their hand, what should they play to block the opponent?	`board = State (3,5) (4,1)` `[((3,5),P2,3), ((5,4),P1,0), ((4,4),P2,1),` `((4,1),P1,2)]` `hand = [(5,0), (5,1), (5,2), (5,5), (3,0),` `(3,1)]` `blockingPlay hand board domSet P1`	A play that would put a 5 on the outside: `((5,1), R)`
2	What if the opponent knocked last move when *0* and *5* were on the outer ends?	`board = State (3,5) (4,0)` `[((3,5),P1,3), ((5,4),P1,0),` `((4,4),P2,1), ((4,0),P1,2)]` `hand = [(5,0), (5,1), (5,2), (5,5), (3,0),` `(3,1)]` `blockingPlay hand board domSet P1`	A play that would create as many zeros as possible: `((3,0), L)`
3	On a controlled set of dominoes, play the domino which will is known to block the player.	`board = State (1,0) (0,0)` `[((1,0), P1, 0), ((0,0), P2, 1)]` `hand = [(3,1), (2,1), (1,0), (1,1)]` `domSet' = [(1,1), (1,0), (0,0), (2,2),` `(2,1), (2,0), (3,3), (3,2), (3,1), (3,0)]` *`Opponent's possible dominoes:`* *`[(2,2), (2,0), (3,3), (3,2), (3,0)]`* `blockingPlay hand board domSet' P1`	A play that will block the opponent: `((0,1), R)`

theBestPlay

The outputs of this function are heuristic, I do not know if they are the best possible. To test this function, I took my *simplePlayer* implementation and modified it to use this function to select which domino it should play (with no extra logic) and then played a number of games against an unmodified (random) *simplePlayer* to see how effective *theBestPlay* truly is in a game.

Each test is run with 1000 games, a target score of 61, and a hand size of 7.

#	Seed	Test Parameters (GHCi prompt)	Output (Wins)
1	0	`domsMatch 1000 7 61 simplePlayer simplePlayer_BestDom 0`	`(32, 968)`
2	1	`domsMatch 1000 7 61 simplePlayer simplePlayer_BestDom 1`	`(34, 966)`
3	2	`domsMatch 1000 7 61 simplePlayer simplePlayer_BestDom 2`	`(43, 957)`
4	3	`domsMatch 1000 7 61 simplePlayer simplePlayer_BestDom 3`	`(38, 962)`
5	4	`domsMatch 1000 7 61 simplePlayer simplePlayer_BestDom 4`	`(40, 960)`

On average, we see a win difference of +92.5% for the *theBestPlay* function. I believe this gives reasonable grounds to say that the function at the very least outputs *decent* plays if it can win over 90% of the time against a pure random player.

smartPlayer

To test my smartPlayer implementation, I examined its win rate against the simplePlayer as well as its execution time (when ran in GHCi, interpreted).

#	Seed	Test Parameters (GHCi prompt)	Output (Wins)	Time (s)
1	0	`domsMatch 10000 7 61 simplePlayer smartPlayer 0`	`(203, 9797)`	`206.2`
2	1	`domsMatch 10000 7 61 simplePlayer smartPlayer 1`	`(219, 9781)`	`207.5`
3	2	`domsMatch 10000 7 61 simplePlayer smartPlayer 2`	`(220, 9780)`	`205.9`
4	3	`domsMatch 10000 7 61 simplePlayer smartPlayer 3`	`(215, 9785)`	`205.9`
5	4	`domsMatch 10000 7 61 simplePlayer smartPlayer 4`	`(205, 9795)`	`206.2`

On average, the *smartPlayer* wins 97.9% of the time against a random player and takes 206.3 seconds to run over 10,000 games. I believe that the *smartPlayer* is as clever as it claims.

Reflection

This module has been my first introduction to Haskell as well as the functional programming paradigm, its lack of familiar concepts such as traditional iteration like for-loops, mutable variables and implicit conversion made for a difficult transition for me at the beginning. Up until now, every programming task I've been faced with was solved thinking imperatively - I thought about my solutions as a series of steps and operations with a working memory "scratch pad" to go from an input state to an output state.

In Haskell, this thinking didn't work, *there aren't even variables,* at least by default. I had a lot of difficulties during the lab sheets - I cobbled solutions together with gigantic *where* clauses because that was familiar to imperative programming.

Taking on this assignment helped me to eventually realize I shouldn't be thinking from the perspective of designing an algorithm/black box that manages input data, but rather from the perspective of the inputs - realizing my solutions as a series of successive mutations on arrays of data. Once I understood this data-oriented approach, I was quite painlessly able to iteratively converge on the output data I wanted for each function as I implemented it. This made for very satisfying and succinct coding, compared to an imperative language like Java, where you must contend with OOP boilerplate and unforeseen emergent interactions between your classes.

My biggest takeaway from this module was learning this new way of thinking, and then discovering all the functional programming interfaces that existed in the languages I had used before, which make any problem involving arrays much more approachable. I now make extensive use of the Java Stream API for my Systems Design module, this API provides many of the list processing constructs I have become familiar with in Haskell, and it has significantly shortened my code while still being comprehensible. I believe I am a better programmer now because of this module, even if I don't continue to use Haskell, the data-oriented problem-solving approach intrinsic to functional programming is something that I will carry forward in my education and career.