COM2108 Functional Programming
Grading Assignment

*Deadline: 3pm, Wednesday 6th December 2023.*

This is a grading assignment, hence worth 60% of your overall mark **if you pass the threshold assessment.** The flip side is that if you pass the threshold assessment, you get 40% for the module even if you get 0 on this assignment. The grading assignment deadline is *before* the later threshold assessment attempt sittings, so you should aim to attempt this assignment even if you have not yet passed the threshold assessment. The work you do on this assignment should improve your understanding of threshold concepts and so help you perform better on the threshold assessment.

Introduction

This programming assignment is based around the game of dominos. The first thing to do, if you are not already familiar with the game, is to learn about it. Wikipedia can help you here:
http://en.wikipedia.org/wiki/Dominos
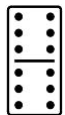
Here is an example of a dominos **board** part-way through a game

left end   *right end*   *Board1*
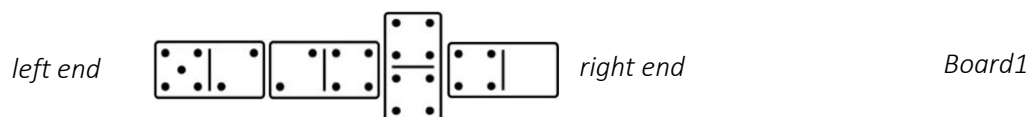
If the next player has in her **hand** the following:

Hand1
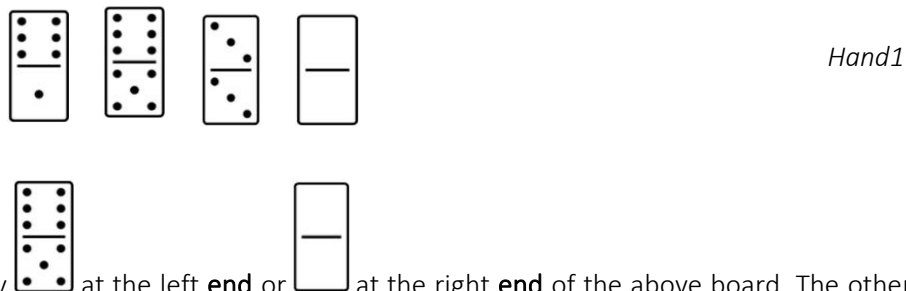
Then she may play at the left **end** or at the right **end** of the above board. The other dominos may not be played. Some dominos games allow plays up and down as well as left and right from a double, otherwise known as a "spinner", but we will not allow such moves.

A standard set of dominos contains all the permutations from to, 28 in all.

There are many variations on the games of dominos and how it is scored. In this assignment you will implement fives-and-threes dominos players (details to follow) and test them against each other by playing simulated matches.

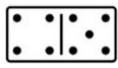Some hints are provided about good play. **You should experiment with your dominos players to see if implementing each hint does indeed improve the player's game.** You are not expected to program all the hints and you may try other tactics.

PLAYING A ROUND OF FIVES-AND-THREES DOMINOS

To play a round of fives-and-threes dominos, you proceed as follows:

- Each player starts with a hand of *N* dominos. The remaining dominos take no part in this round – they are 'sleeping'. *N* is usually 7, but a variation of the rules uses 9 dominos if there are only two players.
- Players add a domino to the board ('dropping' a domino) in turn and accumulate the fives-and-threes scores of their 'drops'. See below for details of how the score is calculated after each drop.
- The player who drops the first domino gets the fives-and-threes score of its total spots (e.g. if the first drop is , which we will represent by (4,5), the player who dropped it scores 3 – full details of how to calculate the score are provided below).
- 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).

SCORING FOR FIVES-AND-THREES DOMINOS

After a player has added a domino to the layout, the pips (number of dots) on the two open ends are added up. Points are awarded as follows:

Pip total	Score	Justification
3	1	One 3
5	1	One 5
6	2	Two 3s
9	3	Three 3s
10	2	Two 5s
12	4	Four 3s
15	8	Five 3s *and* Three 5s
18	6	Six 3s
20	4	Four 5s
Other values	0	Not a multiple of 5 or 3

PLAYING A MATCH OF FIVES-AND-THREES DOMINOS

A 2-player fives-and-threes dominos match is organised as follows:

- A match consists of N games, typically 3 in a real-life setting. **The winner is the player who has won most games.** With our tireless computer players, N can be much greater than 3.
- Each **game** involves a sequence of **rounds** (as defined above). The two players take it in turns to drop first in each round.

- A game terminates when one player or the other wins by achieving an accumulated score, over the rounds, of **exactly V**, where V is the target value. V is typically 61, but a different value can be chosen, such as 31 or 121.
- At the end of a round, if neither player has reached V, a new round starts, and the scores obtained within this round are added on to the scores after the last round.
- **You must finish exactly**, for example if your target is 61 and your score is 59, you need to play a domino which scores 2 to win the game… but if that domino is your last domino, you need to score just 1 from the domino that you play, because you also get a point for "chipping out." If you score more than the target ('going bust'), your score remains the same (e.g. if the target is 61, your score is 59 and you play a domino which scores 4, your score remains at 59).

WHAT IS BEING ASSESSED

In this assignment, we are **not** looking at who can write code to produce the best fives-and-threes dominos player. If you would like to try that challenge, you will have that opportunity at the end of semester, but that will not impact upon your mark for this module.

What we are assessing here is:

1. The design of your solution, demonstrating that you have given some thought to the design of your player. It should be clear that you have given thought to the functional decomposition, and that you have experimented with the suggested strategies. (20%)
2. The testing of your solution, evidencing that you have tested your code thoroughly. You should **not** provide a random set of test data and output. You should give a rationale for the test cases that you have chosen. You may illustrate you testing with a carefully selected set of examples demonstrating that a function is working as intended, but make sure that it is clear to the reader why this set has been chosen. (30%)
3. The implementation of your solution, for both *correctness* and *clarity*. Your code should do what it is intended to do, but it should also be well-presented and understandable by others. This means sensible layout, code presented in a logical order, good choice of function and parameter names, consistency in style, clear comments (header documentation, function documentation, inline documentation *where it is needed*). (40%)
4. Your **critical reflection** about what you have learned when studying functional programming. (10%)

More detail on each of these points is provided towards the end of this document.

DOMSMATCH.HS

You should take a copy of DomsMatch.hs from Blackboard and add your code to the bottom of this file. Note that near the top of this file there is the line "import System.Random". System.Random is a library which is *not* installed by default, but it is essential in this project. You need to install it *outside* of ghci, at the command/terminal prompt, using the command:

    cabal install --lib random

Once you have done this, start ghci, then type

    import System.Random

at the Haskell prompt. If you **don't** get an error, you have successfully installed the library.

This is the **only** library that you should install this way. If you try to use any other libraries that you have installed this way, your code will not run when it is submitted. This will result in a score of 0 for the implementation.

DATATYPES

The key datatypes are provided for you in the file *DomsMatch.hs*. **You should not change these datatypes**. These are:

- **Domino** – representing a domino tile.
- **DominoBoard** – which contains a current board state – either the initial (that is, empty) board, or the current state, represented by: the left-most domino (with left-most pips in the first position in the tuple), the right-most domino (with right-most pips in the second position in the tuple), and the history of the game.
- **History** – the layout of the board from left at the start of the list to right at the end of the list, where each domino is combined with which player played it and which turn at which it was played.
- **Player** – essentially two labels to identify the individual players.
- **End** – indicating the end at which a tile is placed.
- **Scores** – a tuple of scores representing (player 1's score, player 2's score)
- **MoveNum** – to keep track of what was played when.
- **Hand** – a set of dominos
- **DomsPlayer** – this is the type of a *function*. A function of this type will return the move that it will make given the current Hand, Board, and Scores (Player is also an argument, indicating which player *this* player is, which may be useful if you wish to refer to the History when deciding on your move.)

You can add extra datatypes if you wish.

SCAFFOLDING FUNCTIONS

A number of scaffolding functions are provided for you in DomsMatch.hs. The top-level function is **domsMatch** which as the name implies, plays a dominos match. It takes six arguments: the number of games to play in a match, the initial number of dominos in a hand, the target score, the two player functions, and a seed for the random number generator. Lower-level functions play a dominos game and a dominos round. Right now, these functions don't work properly, because they depend upon **scoreBoard, blocked** and **playDom** which are currently incorrectly implemented. The first thing you need to do is implement these functions.

SCORING A BOARD

The first function you must implement is **scoreBoard**. **scoreBoard** takes a **Board** and a **Bool** as arguments, where the Bool is True if the domino just played was the last domino from the hand, and False otherwise, and returns the score that would be received by creating this board state. For example, if the first player played (4,5), the board would contain just this domino, and so would score 3 (three threes – unless for some weird reason you were playing with an initial hand size of 1, in which case the score would be 4). If the next domino played was a (5,1) (which could only be played to the right end), the score on this board ([(4,5)(5,1)]) would be 1 (4+1 = 5; one five – again, unless you were playing with an initial hand size of 2...).

Currently this function simply returns 0 for every input.

CHECKING IF A PLAYER IS BLOCKED

The next function to implement is **blocked**. **blocked** takes a **Hand** and a **Board** and is *meant* to return **True** *only* if there is no domino in the hand that can be played on the board. Currently this function always returns **True**.

PLAYING A DOMINO

Now you need to implement **playDom**, which given a **Player**, **Domino**, a **Board** and an **End**, should play the domino at the given end if it is possible to play it there. The return type should be a **Maybe Board**. (See *Errors and Uncertainty* recording.) Currently this function always returns **Nothing**, but it should *only* return Nothing if it is not possible to play that domino at that end on the board.

OTHER POSSIBLE BASIC FUNCTIONS

Some other functions that *might* be useful include:

- **canPlay:** a predicate (function that returns True or False) given a **Domino**, an **End** and a **Board**, returns **True** if the domino can be played at the given end the board.

- **played:** a predicate returning **True** if a given **Domino** has already been played on a given **Board**.

- **possPlays:** given a **Hand** and a **Board**, return all the **Dominos** which may be played at the left **End** and all those that may be played at the right **End**. The return type should be a pair, where each item in the pair is a list of dominos.

- **doms2scoreN**: given a **Board** and an **Int** n, return all the **Dominos** not already played which could be played to give a fives-and-threes score of n and the **End** at which to play each one.

You do not have to implement these functions; and before you even think about implementing them, you should complete your design and identify where (if anywhere) they will be used. You are certainly going to need to implement other functions than these as well.

CREATING A SIMPLE PLAYER

In DomsMatch.hs, you are provided with a data type for a **DomsPlayer**. This type is for a function that, given a particular **Hand**, **Board**, **Player** and **Scores** it returns a tuple **(Domino, End)** indicating the domino to be played and the end at which to play it.

You must implement a function that will play a valid move when called. You should not attempt to implement any sophisticated reasoning at this stage. Your function must be called **simplePlayer** and it must be of type **DomsPlayer**. You do not have to check is it is possible to play a domino; you can assume that this function will *only* be called if the player has at least one domino in their hand that can be played on the current board.

Remember, just one simple opponent is all that is needed at this stage.

CREATING A "SMART" PLAYER

The next stage is to implement a "smart" player. **Don't jump straight into coding.** Think about the strategy you will use. Playing the same strategy at every move is unlikely to be successful. How will you decide which strategy to use when? How will you implement these strategies? Think about the design *before* you start coding. Also think about how you are going to test your functions to be sure that they are implementing the strategies that you want.

Remember the aim here is not to create the *best* player, but to implement something that demonstrates *some* level of "smarts." If you want to try to create the smartest player you can, by all means try, but remember that the "cleverness" of your solution is not what is being assessed; a solution that implements a semi-smart strategy *well* will have much better outcomes than one that implements a "best in show" strategy using very poor style.

Your function must be called **smartPlayer** and it must be of type **DomsPlayer**. To be considered a valid "smart player," you must implement *at least* three different strategies and logic that will select different strategies depending on the game state.

You may wish to try more than one smart player, and even pit them against each other, but when you submit your code you must choose **one** of these players and name the function **smartPlayer**.

EXAMPLE OF A MATCH

To run a basic match, try pitting two instances of your simplePlayer against each other, e.g.:

```
ghci> domsMatch 100 7 61 simplePlayer simplePlayer 1
(48,52)
ghci> domsMatch 100 7 61 simplePlayer simplePlayer 2
(48,52)
ghci> domsMatch 100 7 61 simplePlayer simplePlayer 4
(62,38)
```

Note that in each case, I am playing 100 games, with an initial hand size of 7, and a target score of 61. The only thing that differs is the random number seed. If I played again with the same seed I (should, if implemented correctly) get the same result. In theory these are two evenly matched players, so you would expect them to win about 50% of the time. You can see though that this is not always the case.

If I play 10000 games instead of just 100 with a seed of four, my result gets much closer to 50% wins:

```
ghci> domsMatch 10000 7 61 simplePlayer simplePlayer 4
(5091,4909)
```

TIPS FOR SKILLED PLAY

Here are some hints about good play:

- Some players normally play the highest scoring domino unless it risks the opponent scoring more (e.g. if the ends are 6 and 0 and your highest scoring domino is (6,6), play it unless the opponent could play (0,3) or (0,6).
- Beware of playing a dangerous domino unless you can knock it off (that is, add a domino to it on a subsequent move), e.g. playing (6,6) if there are no more 6s in your hand is risky. Your opponent might play a domino on the other end that leads to you being blocked.

- The History tells you what dominos remain (though remember, not all will be in play) and therefore what to guard against.
- You can use the History to work out what dominos your opponent may have (e.g. if the ends are 5 & 5 & the opponent doesn't play (5,5) the opponent doesn't have that domino.
- Knowledge about what your opponent is knocking on is particularly useful.
- If you have the majority of one particular spot value (e.g. four 6s) it's a good idea to play that value, especially if you have the double.
- If you have a weak hand, try to 'stitch' the game, i.e. make both players knock.
- If you are getting close in the end game, you should obviously see if you have a winner. If not, try to get to 59, because there are more ways of scoring 2 than scoring 1,3 or 4.
- If the opponent is getting close in the end game, look for a domino which will prevent her/him winning on the next play, or reduce the chances of this happening.
- If you have 'first drop' onto an empty board, a popular choice is (5,4), because it scores 3 but the maximum you can score in reply is 2.

DESIGN FOR THE DOMINO PLAYERS

The aim is to programme smart domino players; essentially, all your players must do is decide which Dom to play at which end. You can experiment by having your players compete amongst themselves.

You are free to programme your domino players any way you like **but remember you must submit your design**. However…

- Except in the end game, search techniques won't get you far (what's the goal state?)
- Instead you want to base your players on the hints (and others you may think of or discover)
- You should design a framework along these lines before you code any players.

You must implement at least **two** different domino players – the **simplePlayer** and the **smartPlayer**. If you have created a good design, it should be possible to write the high-level comments for each of your functions *before* you write a single line of code. Then as you implement your code, make sure that it aligns with the design that you have captured in the comments. You may implement more than two players but if you do this, you **must** have a **simplePlayer** and a **smartPlayer** function, and these (and their subsidiary functions) will be the focus when we are assessing your code.

If you have other code that is not needed for your solution of these two functions:

- If your code does not load or run, you will get 0 for the implementation of your code, even if the **simplePlayer** and/or **smartPlayer** functions would work if you had omitted the additional code.
- If your file loads and runs, any additional code will not be checked for correctness, but it **will** be considered in the judging of **style** of your coding.
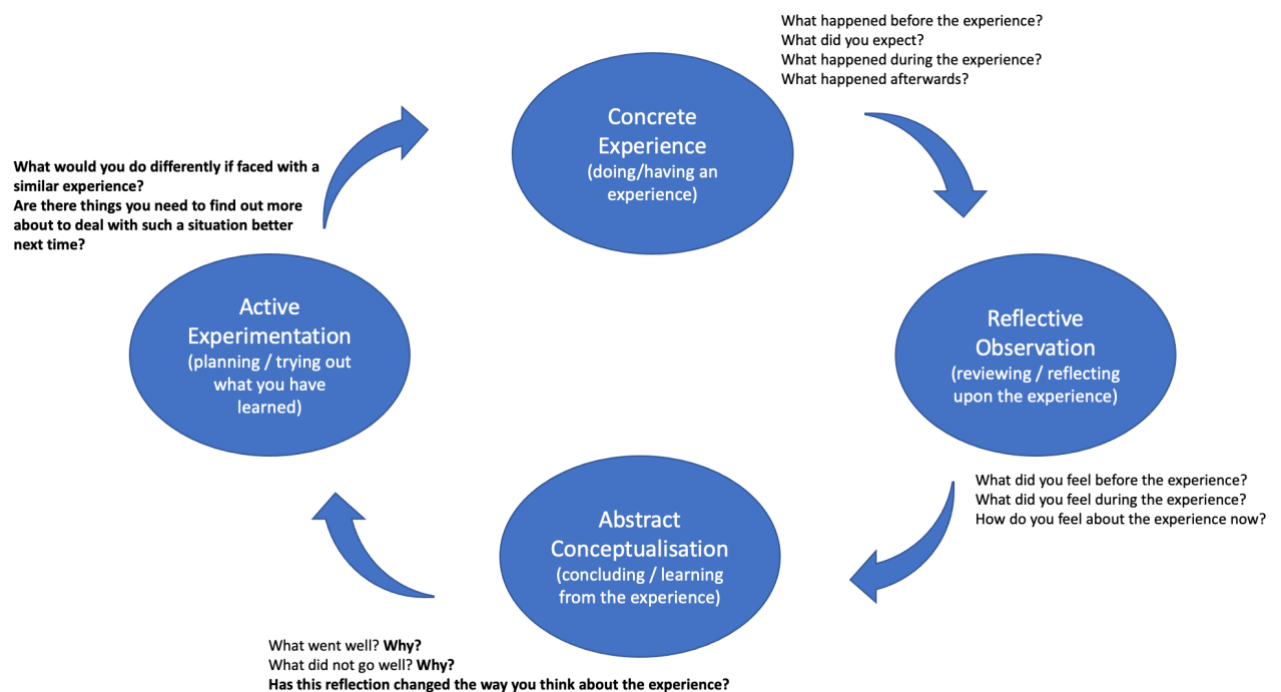
WHAT TO HAND IN

There are two separate submission points. One is for your **report** which should include your design, testing and critical reflection. This should be submitted as a single PDF file; the name of the file is not important. The other is for your **code** which should be in the file **DomsMatch.hs**. Please do not rename this file.

Your report should have three sections:

1. Design. Before you even think about the code, you should be thinking about the design. In this section of the report, you must clearly explain (diagrams can be helpful) the design of your solution. This should clearly be a top-down design, with clear function decomposition.

2. Testing. As you implement your design, add a section to the document you created. In this section, clearly indicate:
   a. The order in which you implemented your functions.
   b. For each function:
      i. The rationale behind the testing for that function
      ii. Some illustrative examples, showing the test input and output. This (small) set of examples should clearly demonstrate that your function is working as in- tended. (A table would work well here!)

   Think carefully about your tests. We do not want to see every test that you performed, but we want to see evidence that you chose an appropriate range of test data and that your functions will work both for general and edge cases.

3. Critical reflection. The final section of your report should be a Critical Reflection on your experiences. What is a critical reflection? It is based around the idea that all our experiences should be part of an ongoing cycle of reflection:

What happened before the experience?
What did you expect?
What happened during the experience?
What happened afterwards?

Concrete Experience (doing/having an experience)

What would you do differently if faced with a similar experience?
Are there things you need to find out more about to deal with such a situation better next time?

Active Experimentation (planning / trying out what you have learned)

Reflective Observation (reviewing / reflecting upon the experience)

What did you feel before the experience?
What did you feel during the experience?
How do you feel about the experience now?

Abstract Conceptualisation (concluding / learning from the experience)

What went well? **Why?**
What did not go well? **Why?**
Has this reflection changed the way you think about the experience?

In this final section, you should write a brief (no more than half a page) critical reflection that picks up on **one or two specific experiences** either from the assignment or the module more widely. Additional guidance on writing a critical reflection will be released in week 9, after the threshold test.

MARKING SCHEME

Element	Checked for	Proportion of marks
Design	For each of **simplePlayer** and **smartPlayer**, clearly present your design. What additional functions are needed? How does the data get used?	20
Testing	For each section, a clear statement of the order of implementation of functions (showing a sensible choice of order) and evidence that you have considered both general and special cases in your test sets.	30
Critical Reflection	You have identified a specific experience and explained how this has changed your practice in the longer term.	10
Implementation (correctness)	**scoreBoard**, **blocked** and **playDom** are implemented and perform correctly against test data.	5
	Each of **simplePlayer** and **smartPlayer** will play against itself giving ~50% wins over a large match size. (Some variation is to be expected, as shown above.) This should work for different variations of initial hand size, target score and seed value.	10
	smartPlayer will play against a secret opponent. The opponent will fit the rules for a **DomsPlayer** function. The performance of the player is does not matter, what matters is that it *does* manage to play against this unknown opponent.	5
Implementation (style)	Sensible naming convention, layout, ordering of functions, length of lines, general readability. Coding style is consistent throughout. Header and (sensible) function comments are included, inline comments used where necessary (too many are as bad as too few). ****The marks for style and comments will be scaled proportionally to the correctness of your code.** If you get 5/5 for scoreBoard, blocked and playDom, 10/10 for playing against self but 0/5 for playing against the unknown player, and 12/20 for style, your overall mark for implementation would be calculated as: $$\frac{5 + 10 + 0}{5 + 10 + 5} * 12 + 15 = 24$$ for the implementation section, **not** 27.	20**

As mentioned above, **if you submit code that does not compile, you will get 0 for the implementation component of the assessment**.

SUBMISSION

Your work must be submitted via the appropriate links on Blackboard before the deadline. Standard penalties apply for late submission:
https://sites.google.com/sheffield.ac.uk/comughandbook/your-study/assessment/late-submission

The work you submit must be **your own individual work**. Serious penalties apply to the use of unfair means. Please refer to the student handbook if you need a reminder:
https://sites.google.com/sheffield.ac.uk/comughandbook/your-study/assessment/unfair-means