

COM2108 FUNCTIONAL PROGRAMMING

PROGRAMMING ASSIGNMENT

DEADLINE: 3PM, MONDAY 14TH DECEMBER 2020.

GRADING ASSESSMENT, 40% OF MODULE TOTAL

*This programming assignment forms part of the grading assessment for this module. You should already have passed the threshold assessment: speak to the module leader, Emma Norling, if not. This assignment accounts for 40% of your mark for the module, or 2/3 of the grading assessment. It builds upon the “Programming Assignment Warm Up” that you should have been working on from week 4 onwards. Solutions to those warm up tasks have been made available, but if you have not completed the tasks yourself, you are likely to find this programming assignment very challenging. It is **meant** to be challenging – the threshold tests will have ensured that you already have a pass mark for this module.*

INTRODUCTION

In this assignment you will implement 5s-and-3s dominoes players and test them against each other by playing simulated matches.

In section 5 are some hints about good play. **You should experiment with your dominoes 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.

5S-AND-3S DOMINOES MATCHES

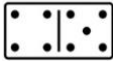
A 2-player 5s-and-3s dominoes match is organised as follows:

- A match consists of N games, typically 3. **The winner is the player who has won most games.** With our tireless computer players, N can be much greater.
- Each **game** involves a sequence of **rounds** (as defined in the programming assignment warm up). 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 61**.
- If at the end of a round neither player has reached 61, a new round starts and the scores obtained within this round are added on to the scores after the last round.
- **As with all good pub games, you must finish exactly**, i.e. if your score is 59 you need to play a domino which scores 2 to win the game. If you score more (‘going bust’) your score remains the same (i.e. if your score is 59 and you play a domino which scores 4 your score remains at 59).

RULES

The rules are a very slight variant on those provided in Stage 2 of the pre-assignment tasks:

- Each player starts with a hand of **9** dominoes. The remaining dominoes take no part in this round – they are ‘sleeping’.
- Players add a domino to the board (‘dropping’ a domino) in turn and accumulate the 5s-and-3s scores of their ‘drops’.

- The player who drops the first domino gets the 5s-and-3s score of its total spots (e.g. if the first drop is  the player who dropped it scores 3).
- Play continues until neither player can play (either because s/he has run out of dominos or is knocking).

MATCH-PLAYING SOFTWARE

A Haskell module **DomsMatch** is provided for you. Its top-level function looks like so:

```
domsMatch :: DomsPlayer -> DomsPlayer -> Int -> Int -> (Int,Int)
```

Here the third argument is the number of games to be played and the fourth argument is an initial seed for the random number generator. **domsMatch** returns the number of games won by the first player (P1) and the second (P2).

To use **domsMatch** your data structures must be compatible. This is what **domsMatch** uses:

```
data DominoBoard = InitBoard | Board Domino Domino History
                        deriving (Eq, Show)
```

- **InitBoard** is the initial state of a round, with nothing dropped
- In **Board**, the two dominoes are those at the left and right ends, with pips lined up, e.g if the board is [(1,5),(5,5),(5,6)] the left end is (1,5) and the right is (5,6).
- The **History** allows the course of the round to be reconstructed (skilled dominoes players work things out from this):

```
type History = [(Domino, Player, MoveNum)]
```

History is a left-to-right list of triples representing the dominoes on the board, who played each domino and the play number, starting at 1. So for example, if the board was [(1,5),(5,5),(5,6)] as above, the history might be [((1,5), P1, 1),((5,5),P2,2),((5,6),P2,3)] – indicating that player 1 played (1,5) first, then player 2 played (5,5), then player 2 played (5,6) – P1 must have not had any domino to play against 1 or 5 (because otherwise P1 would have played turn 3).

```
data Player = P1 | P2 deriving (Eq, Show)
```

```
data End = L | R deriving (Eq, Show)
```

```
type Scores = (Int, Int)
```

```
type MoveNum = Int
```

```
type Hand = [Domino]
```

```
type Domino = (Int, Int)
```

(continued over)

```
type DomsPlayer = Hand -> DominoBoard -> Player -> Scores -> (Domino, End)
```

i.e. a **DomsPlayer** is a function which takes

- The player's **Hand**
- The DominoBoard
- The player's identity (P1 or P2)
- The current **Scores**

and returns the **Domino** to play and the **End** to play it. **Note that this is different to what was used in the pre-assignment tasks!**

EXAMPLE OF A MATCH

One **DomsPlayer** is provided for you (in **DomsMatch.hs**): **randomPlayer** which simply chooses a random play from its hand.

So

```
> domsMatch randomPlayer randomPlayer 100 42  
(52, 48)
```

plays a match of 100 games between two random players, with random seed 42. The first player wins 52 of these games, the second, 48.

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, e.g. playing (6,6) if there are no more 6s in your hand.
- The History tells you what dominoes remain and therefore what to guard against.
- You can use the History to work out what dominoes your opponent may have (e.g. if the ends are 5 & 5 & the opponent doesn't play (5,5) s/he doesn't have it.
- 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 have to 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 provided you start with, and 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)
- Each hint you use should be coded separately into a Tactic
- A Tactic will be associated with a game situation
- You need to be able to easily change your Tactics
- There is a natural order to some Tactics
- A player can be defined by which Tactics it uses, in which order
- You should design a framework along these lines before you code any players.

You must implement at least **two** different domino players. 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 (i.e. the comments you wrote first).

WHAT TO HAND IN

Hand in 3 documents, in a single zip archive:

1. Your **design**, preferably as a diagram (see the examples given in mini-lecture 17c) and explanatory notes.
2. Your **implementation**: commented code as a .hs file, ready to run.
3. Your experimental **results**: you should try to demonstrate that changes you make to the domino players code give them an advantage. This should be a separate, clearly-named document.

AND AS AN EXTRA CHALLENGE...

Stay tuned – there will be an extra challenge (which is optional, and *not* for assessment) announced shortly.

MARKING SCHEME

Component	Details	Marks
Design	Does the design provide a framework for implementing a number of different players?	10
	Are the players themselves well-designed: is it clear from the design how the players decide on their moves?	10
Implementation	Is the code functional , able to play a game of dominoes? (Even if it doesn't play it well, so long as there are at least two players that differ from each other and the provided players.)	20
	Is the framework well-coded , using good functional style?	10
	Is the code well-presented , using clear and consistent layout, with good choice of function and parameter names?	10
	Is the code well-documented , with comments <i>where necessary</i> ? Remember, you should aim for code that is largely self-explanatory. Do not write comments that are not necessary, but <i>do</i> comment code that is not self-explanatory.	10
Results	Are there comparative results for the players?	10
	Are there clear differences in the results, with a clear explanation?	10
	How smart are the players? How much knowledge about the game do they capture?	10
Total		100

SUBMISSION

Your work must be submitted via the appropriate link on Blackboard before the deadline. Standard penalties apply for late submission – see <https://sites.google.com/sheffield.ac.uk/comughandbook/general-information/assessment/late-submission> for details.

This work is intended to be 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/general-information/assessment/unfair-means>