# FIT2102 Programming Paradigms 2019

## Assignment 2: Functional Programming in Haskell

**Due Date:** Friday 18th October, 11:55pm

**Weighting:** 20% of your final mark for the unit

**Task Description:**
You will create an autonomous player for the card game "Hearts." You will do this by filling in the undefined `playCard` function in the `Player.hs` file that you receive in the code bundle.

> *Hearts is an "evasion-type" trick-taking playing card game for four players, [...] players avoid winning certain penalty cards in tricks, usually by avoiding winning tricks altogether.*
> *-- Wikipedia*

Your goal in this assignment is to write a player for the Hearts game. As in the previous assignment, the focus will be on having *functional* code rather than implementing AlphaGo. However, marks will be awarded for advanced strategies.

**Submission Instructions:**

1. You need to submit you Player.hs, as a single file, to the uploader. It is a webform where you can submit your player file, to be run in a continuous tournament against all other submissions (see "The Tournament", below).
2. You need to submit your *code bundle* -- whatever extra players you may have, extra libraries you wrote -- to Moodle. To do so, zip up your whole code directory using the following format:
   **A .zip file named studentNo_name.zip which should extract to a folder named studentNo_name**

We will deduct up to ten (10) marks to people not respecting this convention.

The template Player code (which you have to implement) is in `staticgame/Player.hs`. All code for your player must be in this one file. You may include multiple players (each in their own files) if you think they are interesting, and you may have other code files (e. g., as per the "getting an HD section below"), but all the files that you have edited or added must be clearly identified in the top-level `ChangeLog.md`.

**NOTE:** The uploader is separate from and does not replace the Moodle submission.

# The Game

The game of Hearts is a trick-taking game where the goal to score as *few points* as possible. All cards in the Hearts are worth one point, and the Queen of Spade is worth 13. Your task will be to implement a simple AI to beat other players (students and staff) at this game.

There will be two versions of the game:

1. A two-player version, which will be used to assess the *strength* of your player. This version will use calibrated opponents and can be seen as a test.
2. An online tournament where your player will be pitted against other students' AIs in four-player matches - *Coming Soon...*

To play the game with your AI, you will need to implement a function called `playFunc` which will be given a number of parameters upon which you will need to act and return the card you believe will lead to victory. An important component of the game setup is that you will be allowed to have a form of *memory*.

Memory, in this case, will be entirely determined by you. The game will simply pass you information pertaining to the current round: your hand and other cards played; and the previous round: the complete set of cards played at the previous round and your memory.

The game you will play will follow most of the classic rules of Hearts. This game is very popular as it is easy to learn and play, thus many online versions are available if you want to train. Do note, though, that most of them usually vary a little, so take care to read the rules carefully.

At the core of the game of Hearts are the *point cards*: all cards in Hearts and the Queen of Spade. The goal of the game is to score as few points as possible. An alternative exists though: if a player manages to take all point cards, she gets 0 points and *every other player* gets 26 points (the maximum).

At the end of each *round*, when players have played all the cards in their hands, we tally the points in every trick taken by each player. That is, we sum the point cards they won.

The game goes on until at least one player has scored more than 100 points *and* there is exactly one player with the lowest score. The player with the lowest score is the winner.

## Scoring system

Counting points in Hearts is done as follows:

- **Point cards**: Each Hearts is worth 1 point, the Queen of Spades is worth 13 points.

- **Hand**: At the end of each hand (players have played all their cards), each player receives the amount of points in the tricks they won.
- **Game**: The game stops when a player has reached 100 points and exactly one other has the lowest score; the latter is the winner.

### Rules of play

The rules of Hearts we will use are as follows:

- **Reneging**: Each turn starts with a player playing a single card, the suit of which determines the suit of the trick. Other players must *follow suit* (play cards in the same suit) if they are able to; if they do not have any cards in that suit, they may discard any card of their choosing.
- **Bleeding**: No point card may be played during the first round unless the player has no other choice.
- **Leading**: The player with the Two of Clubs leads (starts) the first round with this card.
- **Breaking**: No player may start a trick with a Heart before any has been played, unless they have no other choice.
- **Shooting the Moon**: In case a player took all the point cards, she scores 0 points and the other players all score 26 points.

## Deliverable

**You will submit a .zip file named studentNo_name.zip which should extract to a folder named studentNo_name**

Your task is to write a `Player` for Hearts which will implement the following function:

```
playCard :: PlayFunc -- play a card from your hand during a trick
```

Look carefully at the type of `PlayFunc` defined in `src/Hearts/Types.hs`. Your player must pass the tests to be eligible to run in the tournament. Your implementation must respect the rules explained above. Also, your code must compile without warnings.

You will have two tasks during the assignment period:

1. Upload your player to the tournament so you can evaluate your player's performance.
2. Upload your `Player.hs` file to Moodle before TBD.

Before uploading your player, please check that the following run:

- `$ stack test`
  This will run the tests on your player, making sure your functions respect the playing

rules. If your code does not pass the tests, you will not be able to access the tournament.

The code provided uses the Safe pragma[1] to make sure the code you use is okay to run. It is also compiled with the `-Werror` flag which means that all warnings will throw errors and prevent your code from compiling. So do make sure you run the test suite before you upload your player.

## Memory

An important concept during a game is memory. In this implementation, you can decide what information you want to save between turns, the only condition is it needs to be converted to a `String`. This is shown in the type of the play function:

```
-- | Play function type.
--
-- Cards are added to each trick from the front.  Thus, the
-- first-played card in a trick is the last in its list of cards,
-- the lead suit is the suit of the first-played card in the
-- trick.  A play is only legal if it does not renege on the lead
-- suit.  That is, the player must play a card of the suit led if
-- they have it.  The winner is the highest trump card (if any
-- were legally played), or, if no trumps were played, the
-- highest card in the lead suit.
type PlayFunc
  =  PlayerId -- ^ this player's Id so they can identify
              -- themselves in the bids and tricks
  -> [Card]   -- ^ the player's cards
  -> [(Card, PlayerId)]
  -- ^ cards in the current trick, so far
  -> Maybe ([(Card, PlayerId)], String)
  -- ^ previous player's state
  -> (Card, String) -- ^ the player's chosen card and new state
```

The memory is the last parameter to the function, of type:

`Maybe` *(cards from the previous round, previous memory)*.

In the first round, this will be `Nothing`. Good use of the memory is one of the key features we will evaluate.

[1] More info at SafeHaskell, but this should not hinder your work.

### Type ambiguity

One thing to note, the game code uses a number of functions with the same name. While this is not an issue in most cases as the compiler can disambiguate using types, this can lead to spurious error messages if you are not careful. For example:

```
error:
      Ambiguous occurrence ‘tricks’
      It could refer to either the field ‘tricks’,
           imported from ‘Hearts.Types’ at ...
               (and originally defined at .../Hearts/Types.hs:52:3-8)
           or the field ‘tricks’, ...
```

To avoid this error, you need to explicitly define the type of either the variable or the function:

```
tricks (hr :: HandResult)
-- or
map (tricks :: HandResult -> [Trick]) results
```

### Multiple players

If you want to test multiple players, you can use the code at `app/Main.hs`. To do so, you will need to create a new file with the name of your player and provide the proper header:

E. g., file: `StrongPlayer.hs`

```
module StrongPlayer where
-- add your code
```

Remember to switch the module name back to `Player` for the one you decide to upload to the server.

## The Tournament

We will run a tournament online based on the code provided. Except the interface, this will be the same game.

When you submit your player, we will run the test suite on it (stack test). The tests on the server will be slightly longer, your player will compete in:

1. A 1v1 against a basic player.
2. A 1v1 against the minmax player.
3. A 1v1 against the advanced player.

4. A free-for-all, four player match against all the players above.

After the tests have run successfully, your player will join the tournament by immediately playing ten games against selected opponents. After that, it will be selected at random to play against newcomers.

The testing and running the tournament can be resource intensive and the server is a fairly slow machine. If you do not get immediate feedback, especially during the last days, no need to spam the Upload button, wait a bit.

**Reminder**: Your rank in the tournament will not have a direct impact on your mark. A high-performing player with spaghetti code will be graded lower than an average, well-written player.

The server for the course is at https://fit2102.monash with the following pages:

- The uploader: after logging in, this page will allow you to upload your code and compete in the tournament.
- The handout: this document.
- The ladder: this page will display the scores of the last tournament run.
- The docs: documentation about the assignment's code.

Once you upload your player, you will see two links on the page:

- `home.php`: shows your current ranking, last upload, and previous games played.
- `status.php`: shows the status of your current upload.

Furthermore, you can inspect your games by clicking on their number.

## Game AI

The goal of this assignment is not for you to develop an AI which can compete with openAI or AlphaGo. The emphasis should be on code quality and applying functional concepts. Below, you can find a list of standard AI algorithms, ranked roughly by implementation difficulty. It is possible to receive an HD with a well implemented heuristic player that meets the other HD criteria. However, students who research and successfully implement the latter strategies will be rewarded.

- **Naïve AI**: tries to play its best card given the current state of the game, you can start by implementing one to make sure you respect the game's rules.
- **Heuristic player**: will save additional information about the game being played to enhance its decision at each turn.

- **Probabilistic player**: will make use of probabilities to determine which cards have the highest chance of winning the game (not the trick).
- **MinMax**:[2] tries to minimise the maximum loss of a player by building a tree of possible moves and playing against itself. This method was developed for two-player, zero-sum game with perfect information.[3] In this context, it will be useful in the two-player version but will require modification in the four-player context.
- **Monte Carlo Tree Search**:[4] is the fusion between search algorithms such as minmax and using probabilities to determine the branching in the search tree. Will also make use of a *simulation* phase to explore deeper.

# Assessment

The assessment for this assignment will be in three parts:

1. Code quality, documentation and comments (30 marks)
2. Functional programming (40 marks)
3. Performance (30 marks)

## Code quality, documentation and comments

### Documentation and comments

You are required to provide a *file-level comment*, *function comments* and *line comments* where necessary. The most important part of commenting is to provide information, not to describe the code.

- **File-level doc**: You should provide an *overview* of your code and explain your rationale (how you decided to play) in a block comment at the top of the file. Think of this as a technical report: you want to summarise the workings of the code and highlight the interesting parts.
- **Function comments**: These comments should be situated above the type declaration of the function they refer to. They should give the *manual* of the function, explaining how it works -- and not simply describing what each line/function does.
- **Line comments**: In some cases, your code might be hard to read, or the action you do is counter-intuitive; in those cases, you can add a line comment to add information.

[2] https://en.wikipedia.org/wiki/Minimax
[3] Such as chess, checkers, or more exotic ones.
[4] Difficult, implementing such a strategy would qualify as an extension.

### Code quality

The quality of your code is a measure of how easy it is for us to understand it. This is a bit more than a simple style guide but you can use the following as guidelines:

1. Keep a reasonable length for lines of code.
2. Do not nest control structures (`if`, `where`, `case`, `let`, etc.).
3. Use point-free notation where appropriate.[5]
4. Write small reusable functions.
5. Have clear, informative variable and function names.

## Functional programming

Your use of FP concepts will be the main evaluation criterion for your assignment. Your code needs to use, efficiently, concepts from the course. You can think of this as a two-part marking scheme:

1. Apply concepts from the course. The important thing here is that you need to actually use them somewhere. For example, defining a new type and its Monad instance, but then never actually needing to use it will not give you marks.[6]
2. Have general code. Because of the tournament format, you should try to have code that can deal with two- and four-player games using the same functions instead of writing two separate blocks of code.

Finally, make good use of the memory, not only in what you save but in how you define and use it. In a way, this goes hand-in-hand with the two points above; you will have to have a non-trivial implementation and be able to use it in the two- and four-player contexts.

You can use the following as a non-exhaustive list of FP concepts, but beware not to affect your code quality:

1. Leverage higher-order functions.
2. Use function composition.
3. Have custom types and typeclasses.

## Extensions

Below is a non-exhaustive list of possible extensions. As with code quality, your implementation needs to be non-trivial and you need to use it to qualify.

- Using the tournament server logs to improve, or learn, your strategy.
- Using a form of `State` monad.

---

[5] Rule of thumb: if you start having double points, stop.

[6] Note: using bind (`>>=`) for the sake of "using the Monad" when it is not needed will not count as efficient usage.

- Creating your own RNG.[7]

## Performance

The important thing in this assignment is: ***your rank in the tournament will not influence your mark.*** The performance of your player will be evaluated automatically by the test suite. You get 10 marks for defeating each:

1. The basic player, only tries to play a valid move.
2. The minmax player, which uses a simple implementation of the min-max algorithm.
3. The advanced player, which will use a more advanced search strategy.

## Marking rubric

Below are the guidelines for the different grades for this assignment. This does not mean that you automatically get the grade if you fulfill them, rather, this is what a good implementation would achieve.

- **Pass**: The code compiles and your players respects the rules of the game.
- **Credit**: In addition to the criteria for a Pass, the game implements some strategy to choose cards that make it stronger than a random card selecting player.
- **Distinction**: The code is well structured, efficient, and uses some advanced concepts from the course (higher order functions, function composition, monadic operations, etc.). The code should use the memory string parameter of the playFunc to inform card choice based on what has already been played.
- **High Distinction**: The code does not contain any excess parts, the player can defeat all training opponents, it implements at least one extension, and the commenting supports the submission. The code uses advanced concepts like state monad and parser combinators (e.g. to parse the memory string).

You can expect a higher mark with an average level AI with elegant code that demonstrates functional concepts covered in lectures, rather than a high-performing AI with poorly structured spaghetti code.

Outstanding submissions, common mistakes and complete disasters may be discussed in class with your name removed.

## Plagiarism

We will be checking your code against the rest of the class, and the internet, using a plagiarism checker. Monash applies strict penalties to students who are found to have committed plagiarism.

[7] Hint: you can use the shuffled deck as a source of entropy.

Any plagiarism will lead to a 0 mark for the assignment and will be investigated by the staff. There is a zero-tolerance policy in place at Monash. So be careful and report any collaboration with other students or other sources.