FIT2102 Programming Paradigms 2018

Assignment 2: Functional Programming in Haskell

Due Date: Friday 19th October 2018 at 11:55pm

Weighting: 20% of your final mark for the unit

Task Description:
You will create an autonomous player for the card game "Oh Hell".
You will do this by filling in the undefined `playCard` and `makeBid` functions in the Player.hs file that you receive in the code bundle.

Oh Hell is a trick-taking card game in which the object is to take exactly the number of tricks bid: taking more tricks than bid is a loss.[1]

Your goal in this assignment is to write a player for the Oh Hell 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:
Zip up your whole code directory and submit to moodle. 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 files that you have edited or added must be clearly identified in the top-level README.md.

NOTE: There is also a webform where you can submit your player file, to be run in a continuous tournament against all other submissions (see "The Tournament, below). This is separate from, and does not replace, the Moodle submission.

The Game

Games of Oh Hell involve four players, playing a sequence of hands. Each player is dealt a hand whose size ranges from 3 to 12 cards. After dealing the hands, a "trump card" is taken from the deck and revealed to all players. This card determines the trump suit. Trumps are cards that win over all other suits.

[1] Wikipedia

Then, each player in turn will make a bid on how many tricks they think they can win. A trick involves each player playing one card from her hand. Players must follow suit, if they cannot, they may play a trump card. Cards are ordered in rank, except for the `Ace` being the highest.

A hand finishes when all cards have been played. A game comprises nineteen hands, from three cards to twelve then back down to three. In case they fulfill their contract (win precisely as many as they bid), players score ten (10) points plus one (1) point for each trick they took. The winner is the player with the most points.

Deliverable

Your task is to write a `Player` for Oh Hell which will implement the following two functions:

1. `makeBid :: BidFunc --` announce how many tricks you plan to take.
2. `playCard :: PlayFunc --` play a card from your hand during a trick.

Look carefully at the types `BidFunc` and `PlayFunc` defined in src/OhTypes.hs.
Your player must pass the tests to be eligible to run in the tournament. Both functions must respect rules explained below. 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 Friday 19th October, 11:55pm.

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

- `$ stack exec staticgame`
  This will run a single game with four instances of your player.
- `$ stack test`
  This will run the tests on your player, making sure your functions respect the bidding and 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[2] 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.
The tournaments will be run with four players at a time. One game (within a tournament) involves dealing a number of cards (between 3 and 12) to each player, then another card is

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

turned over to be the “Trump Card”. The suit of the “Trump Card” is called the “Trump Suit”. Cards from the “Trump Suit” are called “Trumps”. A Trump will always beat a non-Trump. There are then two phases:

Bidding: Before a game starts, you have to announce how many tricks you plan to take (win). To help in this decision, you will have knowledge of the current trump card, your hand, and the bids of the previous players.

Bidding must respect the following three rules:

1. Cannot bid less than zero tricks.
2. Cannot bid more than the cards in hand.
3. *Hook rule*: the total of the bids must not equal the number of cards in hand.

Rule number 3 means that, in case you are the last player to bid, your bid cannot bring the total number of bids to the size of your hand -- so there is a bit of fun.

Playing Tricks: The core of your work will reside in the playing function: `playCard`. Using this function, you will choose the card you wish to play from your hand. To do so, you will have knowledge of the bids made at the start of the game, the current trump card, and the cards played before your turn.

Playing a card must respect the following rules:

1. *Reneging*: a player must follow the suit in the current trick, given by the first card played.
2. If a player does not have a card in the current suit, she may play any card.

Additionally, to allow us to run a tournament consistently your code must complete within one second (1s). If you time out, this will count as a play error.[3]

Scoring: for each game you get 10 points if you win precisely the number of tricks that you bid, plus the number of tricks that you won. You get 0 points if you do not win the number of tricks that you bid.

The Tournament

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

[3] This should not be an issue for most players, only advanced, non-deterministic strategies should worry about this.

Important: 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 .

We run a server for the course 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.

Marking

Minimum: all of these requirements must be reasonably executed to achieve a passing grade (detailed marking rubric below):

- Implement the functions above correctly so that they follow the rules of OhHell.
- Ensure all the tests pass and a game can successfully be run with your Player implementation.
- The player must make some attempt to win.

If all of the above are implemented in good functional style up to a D can be achieved. To get a higher grade you have to use a little creativity and add some functionality of your own choice - suggestions below.

Ideas for getting an HD. Any one or more of the following (or something of your own devising with a similar degree of complexity) done well (on top of the basic functionality described above) will earn you an HD provided it is implemented using the functional programming ideas we have covered in lectures and tutes:

- Use the parser combinators from the Week 10-11 tutes to parse the log files from tournaments to compute statistics that inform your strategy
- Find good uses for typeclasses we studied in the lectures, such as Functor, Applicative, Monad, etc.
- Your own ideas!

Some of the above may require a little independent research. That's what computer science is all about.

Tips for getting started:

- Read the course notes and complete the tutorial exercises.
- **Start as soon as possible**. Don't leave it until it's too late. There will be no extensions unless you qualify for special consideration.

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.

Marking Rubric:
It is important to realise that (as stated above):

- If you implement the **Minimum** requirements above demonstrating application of functional programming ideas from our lectures and tutes you will achieve a pass grade.
- You can receive up to a D for perfectly implementing the Minimum requirements demonstrating a thorough understanding of how to use the features of Haskell to write clean, clear functional UI code.
- To achieve HD and up you will need to do the above plus some aspect of additional functionality as described above.
- Rubric table next page:

	Mark%	P	C	D	HD
OhHell Game	30	Tests pass and player bids and plays legally		The player employs a reasonable strategy to attempt to win.	The player implements a strong strategy. Implemented ideas beyond the minimal requirements such as (but not limited to) the suggestions above
Functional Programming and general coding style	30	Attempted	The code is clear and commented	The code is composed of functionally independent blocks	Demonstrated understanding of advanced concepts from the lectures
Type correctness	20	d			Functions use types that are as generic as possible. Types are used effectively to ensure correctness of program.
Well	20	Attempted			There are clear comm

documented code					ents at the start of all files that have been modified by the student outlining their changes to the given code. The design of their functionality and the way it uses functional programming principles from the lectures and course notes should be clearly explained.