Programming task: card game

Obtain the source code according to the paper *Functional Software Architecture*. Change to the directory `java-magazin-2019` and build the program according to the `README.md`.

You need to submit two files where `0000000` in a file name stands for your matriculation number.

- A zip file containing just the `src` directory created using the command `zip -r m0000000 src`. It should contain the code after all subtasks, but not the bonus questions.
- A plain text file (markdown ok) `m0000000.txt` that provides an explanation of what you implemented for each of the following subtasks (which modules did you change, which functions, data structures, type classes, etc are affected).

The code in the zip file must build and run using stack as described in the original `README`. Do not submit binaries as they will be ignored. Do not change the underlying architecture as it is explained in the paper.

We will test the strategies you implemented using our implementation of the tournament, which will make it easy to compare the implementations.

1. (10 points) Modify the pretty printing so that playing cards are displayed using the Unicode characters for suits as shown in the Wikipedia article for Hearts (do not worry about the colors). Hands should be shown as a comma-separated list of cards. In one place, printing of cards is not affected by changing the pretty printer. Fix this place so that cards are always shown using the pretty printer.
2. (10 points) There is some dead code in modules `Gameplay` and `Game`. Find it and remove it.
3. (10 points) The calculation of final points is incorrect. Find the problem and fix it.
4. (10 points) On startup, the game erroneously prints the `HandsDealt` event to the console. Suppress this printout.
5. (10 points) Another way of winning the game is to collect all points. Change the end game message such that the player who collected all points is declared winner (*shooting the moon*). Otherwise, the player with the most points is declared loser.
6. (10 points) Improve the strategy of the interactive player `playInteractive` such that it only offers legal cards for the next move. Moreover, it automatically plays a legal card if there is only one possible move.
7. (20 points) Implement a **simple** robo player `shootTheMoonStrategy` that attempts to collect all scoring cards. This strategy must be named

`shootTheMoonStrategy :: PlayerStrategy` and placed in a separate file (module) named `M0000000.hs`. (Do not use the history component implemented in the next item as doing so makes the task more difficult.)

To do so cleanly, move `start` and the definitions of the players to the `Main` module. This way, your strategy can import the module `Gameplay` and it can be imported into the `Main` module as follows:

```
import qualified M0000000
-- ...
player0000000 = G.makePlayer "0000000" M0000000.shootTheMoonStrategy
```

8. (10 points) The supplied robo player `playAlongStrategy` plays very badly, partly because it does not have enough information. To amend this lack of information, define a type `PlayerHistory` as `[(PlayerName, Trick)]` and extend the `PlayerState` with a history component `playerHistory` of type `PlayerHistory` to collect all tricks so far along with the player who took them and implement trick collection. Add a boolean flag `playerShoots` to indicate whether the robo player goes for shooting the moon. Take care that the new components are properly initialized.

9. (50 points) Devise a strategy (based on the extended `PlayerState`) that examines the history to determine the next move. This strategy must be named `strategy :: PlayerStrategy` and placed in a separate file (module) named `M0000000.hs` so that different strategies can play against one another. It should win consistently against `playAlongStrategy` and there will be bonus points for strategies that win against other students' strategies.

To do so cleanly, move `start` and the definitions of the players to the `Main` module. This way, your strategy can import the module `Gameplay` and it can be imported into the `Main` module as follows:

```
import qualified M0000000
-- ...
player0000000 = G.makePlayer "0000000" M0000000.strategy
```

10. (10 points) For testing and also to fairly compare different strategies, it would be good to provide the card dealing process as a configuration parameter for `Gameplay.runGame`. Change `runGame` so that it takes a first parameter of type `IO [Card]` that returns a shuffled card deck. Modify `start` so that the original behavior is preserved.

11. (30 points) Use the library `optparse-applicative` to build a command line interface for the interactive game. The interface should work as follows:

```
% stack run -- -h
hearts - run the game of Hearts

Usage: hearts ([-p|--playAlong PLAYER] | [-s|--shootTheMoon PLAYER] |
  [-m|--myStrategy PLAYER] | [-i|--interactivePlayer])
  Run the game of Hearts in interactive mode

Available options:
  -p,--playAlong PLAYER         Player name
  -s,--shootTheMoon PLAYER      Select ShootTheMoon strategy
  -m,--myStrategy PLAYER        Select my strategy
  -i,--interactivePlayer PLAYER Select interactive strategy
  -h,--help                     Show this help text
```

To run the interactive game with all available strategies, it can be invoked as follows:

```
% stack run -- -m Mike -i Peter -p Annette -s Nicole
```

12. (50 points) Implement a framework to conduct a tournament for an arbitrary number of players of type `Player` in a separate new module `Tournament`. The framework should form groups of four players and have them play for a configurable number of rounds. One round means to execute the game for all permutations of the players on the same card shuffle. In each stage, groups with less than four real players are filled up with copies of the `playAlongPlayer`. You need to devise a means to make `runGame` report the outcome of a game, which is currently only printed, without compromising the architecture. Moreover, you must refactor `runGame` according to the previous item.

The entrypoint to the tournament should be

```haskell
type ScoreMap = Map PlayerName Int

start :: [Player] -> Int -> IO ScoreMap
start players nRounds = ...
```

where result associates each player with its score. For a win by shooting the moon, each player gets the points as scored (winner 26 points, all others 0 points). For an ordinary game, each player gets the difference of 26 and the points collected.

Use the library `optparse-applicative` to extend the command line interface so that you can run the tournament to evaluate your strategy against `playAlongPlayer` and `shootTheMoonStrategy`. The interface should work as follows:

```
% stack run -- -h
hearts - run the game of Hearts

Usage: hearts [-t|--tournament] [-n|--rounds ROUNDS] ([-p|--playAlong PLAYER] |
              [-s|--shootTheMoon PLAYER] | [-m|--myStrategy PLAYER] |
              [-i|--interactivePlayer])
  Run the game of Hearts interactively or in tournament mode
```

```
Available options:
  -t,--tournament               Switch to tournament mode
  -n,--rounds ROUNDS            # rounds for tournament (default: 1)
  -p,--playAlong PLAYER         Player name
  -s,--shootTheMoon PLAYER      Select ShootTheMoon strategy
  -m,--myStrategy PLAYER        Select my strategy
  -i,--interactivePlayer PLAYER Select interactive strategy
  -h,--help                     Show this help text
```

The `-i` flag does not make sense in tournament mode; this should be flagged as an error. If the `-t` flag is absent, all other flags are ignored and the game runs interactively.

To evaluate the three strategies we can write

```
% stack run -- -t -s Anton -m Berta -n 10
```

which will run 10 rounds of the tournament for players Anton using ShootTheMoon and Berta using MyStrategy along with two Robo players using the PlayAlong strategy. The final output should look roughly like this:

```
("Anton",3536),("Berta",5248),("Robo-1",4760),("Robo-2",4760)
```

which demonstrates that MyStrategy is superior to the other strategies. Of course, your numbers will be different.

Bonus questions

Bonus questions are submitted in separate files called `bonus-0.zip` and `bonus-0.txt` where `0` is replaced by the number of the bonus question. The code can build directly on the baseline from the repository. The `.txt` file should contain your explanation as outlined above.

13. (30 points) Consider the paper's footnote about the `HandsDealt` event. Design an event data structure that supports broadcast events (for all players) as well as targeted events for single players. Implement a suitable event delivery function.

14. (50 points) We uniformly extended the `PlayerState` type so that it works with all intended scenarios. This approach is not the best possible. Ideally, each implementation of a player should be able to choose their own type for `PlayerState`. Suggest how to adapt the code in this way without disrupting the overall architecture. Demonstrate that your suggestions works with a prototype implementation that uses different types for the player state in the strategies `playAlongStrategy` and `playInteractive`.