CS4021-FP Assessed Exercise 2: Protect the λ

1 Aims and Objectives

This assessed exercise will test your ability to develop Haskell projects and use advanced Haskell programming idioms. In this assignment you will be asked to construct a two-player, turn-based, strategy board game as an interactive command line program using the Haskell language.

Please read the problem scenario presented in §2; and the functional requirements in §3. Further, make sure you read the submission guidance and information about the supplied project skeleton in §4.

To attempt this exercise you should have covered the MOOC in its entirety, and the two weeks of lectures delivered after the MOOC. Please review the lessons prior to attempting this exercise.

The main learning objectives of this exercise are:

1. to demonstrate understanding of how to express data structures and function interfaces using types.
2. to demonstrate understanding of how to structure programs using monads.
3. to develop substantial software applications including system interaction.
4. to construct, adapt, and analyse code using standard Haskell platform tools.

2 Game Rules

Tafl Games are a family of ancient Germanic and Celtic strategy board games not too dissimilar to Chess and Go[1,2]. The exact game variations and configurations have been lost to time or are only partially recorded. For this assignment you will be implementing an adaptation of the *Tablut* variant that has been designed specifically for this course called: *Protect the* λ.

The main objective of the game is to protect the λ from the *objects* using your λ-*guards*.

The game is played on a 9×9 board with 25 pieces—nine defending pieces and sixteen attacking pieces.

2.1 Piece Placement

The starting positions for each piece are as follows:

- The main λ starts on the central square, which no other piece may occupy.
- There are eight 'λ-guards' that defend the λ. Initially, the protectors are distributed equally along the cardinal directions (rows and columns) radiating out from the main λ.
- There are sixteen attackers—*objects*. The objects are organised in a 'T-Shape' at the end of the cardinal row and column of the λ.

Figure 1 illustrates the initial placement of the games pieces, where 'O' represents objects, 'λ' represents the λ, and 'G' represents the λ-guards.

[1] https://en.wikipedia.org/wiki/Tafl_games
[2] http://tafl.cyningstan.com/page/170/rules-for-tablut

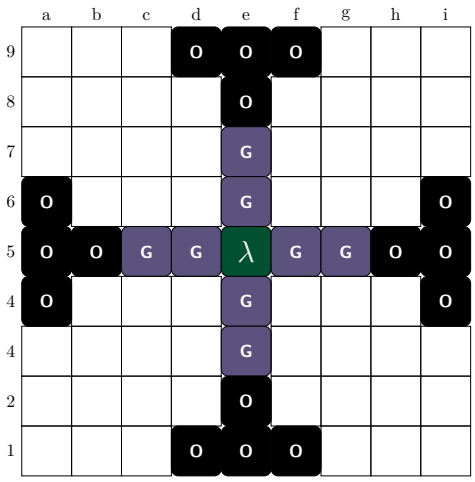

Figure 1: Initial Placement of Board Pieces

2.2 Movement

Movement is as follows:

- The attacking side (O team) makes the first move.
- Pieces can move any distance along their cardinal axes, i.e. along a row or column.
- Pieces cannot jump over another piece.
- A null move (from a square to the same square) is not permitted.
- No piece can move *onto* the central square. Pieces can move along a path that *crosses* the central square, so long as it is not occupied.
- Once the λ has left the central square it cannot move back onto the square.

Figure 2: Guard is captured by two Objects, one each side

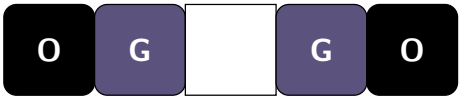

Figure 3: Two Guard pieces would be captured if an Object moves into the gap here

2.3 Attacking

When a piece is *captured* then it is removed from the board, like Chess. Pieces are captured as follows:

- *Custodial Capture* occurs when a piece is surrounded by two opposing pieces on two opposing sides, e.g. see Figure 2. A piece is *not* captured when it moves into a gap between two enemy pieces—i.e. you cannot be captured as a direct outcome of your own move.

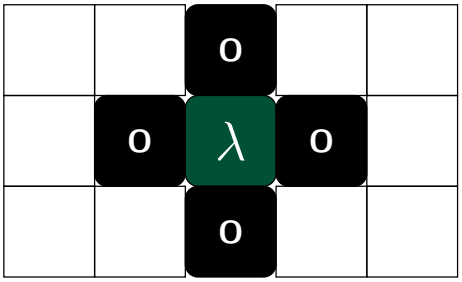

Figure 4: Lambda on centre square is captured when surrounded four Objects

- A piece can also be captured if it is between the empty central square and another opposing piece.
- A single move could result in multiple captures, if it causes two enemy pieces to both be surrounded, see Figure 3.
- If the λ is in the central square it can be captured if it is surrounded on all four sides by objects, see Figure 4.
- If the λ is adjacent to the central square then it can be captured if it is surrounded on the remaining (three) sides by objects.
- If the λ is on any other square, then it can be captured by enemies on two opposite sides, as with any other piece.

2.4 Winning/Losing

Games are won or lost as follows:

- If the λ has a clear path to the edge of the board, then the λ team has won. At the start of every λ team turn, the game should check whether the λ has a clear path to the edge—if it does then the victory is reported . . . the player does *not* need to make an explicit move.
- If the λ is successfully captured by objects then the objects have won.
- If a team loses all its pieces, then that team has lost.

3 Functional Requirements

Here we describe the game's functional requirements.

- The game is to be turn based. Each player takes a turn to make a single move.
- Players will interact with the game using a simple command prompt. The prompt will indicate whether a game is in progress, and when in a game, the player whose turn it is. Appendix A.2 describes the set of commands that should be supported, and how the prompt should appear. During game play the command prompt will display the current state of the board to users after each turn. However, if the game has been launched with `--test` the state of the board is *not* displayed between commands.
- Games can be saved and resumed. During game play the game state can be saved to an external file. Prior to starting a game players should have the option of loading an existing game. Appendix A.3 describes the serialization format for game states.
- The program will have a simple set of command line arguments to load an existing game state, or disable the display of the board between commands. Appendix A.1 illustrates the possible invocations.

4 Project Skeleton

You will be given a project skeleton that provides a common and minimal project setup. The project will contain a partial test suite to help guide development of your submission. You are required to implement the remainder of the game using the skeleton. The skeleton will provide a `README` file explaining how to compile the project and run the test suite.

We will use the project's provided infrastructure when assessing the project's functional correctness. Further, we will replace the partial test suite with a more exhaustive set of tests. Thus:

> **Please do not, under any circumstances, alter the supplied project skeleton unless instructed to.**

If you have any questions about the project's structure please get in touch, ideally using the moodle chat forum for this coursework.

5 Important Information

5.1 Submission Details

Please submit your projects to Moodle using the instructions below. Please make sure you consult Moodle for the submission deadline.

- You are to submit your project as a single **ZIP** archive.
- You must name the archive and the archived folder using your student number, e.g. `1234567a.zip` containing folder `1234567a`.
- You must ensure that your project can be executed using the provided infrastructure.
- Please do not submit executable files or any intermediary files generated by Haskell or other tooling.

> **If your submission is incorrectly archived, named, or fails to build using the supplied skeleton you will lose marks.**

5.2 Marking

The standard mark descriptors will be used and can be found in the student handbook for more information.

- **Clarity of Design (20%)**—Was the design of the project suitable for implementing the assignment? Did you make appropriate use of external libraries and internal modules?
- **Coding Style (20%)**—How well was the project presented? and did you make use of idiomatic features? Does the hlint analysis report significant warnings?
- **Documentation Coverage (10%)**—How complete and useful was the documentation coverage for the project? Were the provided code comments suitable?

- **Functional Correctness (40%)**—How sufficient was the implementation w.r.t. this specification document?
- **Submission Quality (10%)**—How well was the assignment attempt submitted? Was there a useful `status.txt` file explaining the code development progress, and known bugs? Does the Haskell code compile without errors? Is the student's ID specified correctly in the zip archive name?

5.3 Lateness Penalty

Any work submitted later than the advertised deadline on Moodle will be penalised using the standard penalties. Please consult the CS undergraduate class guide[3] for more information.

5.4 Plagiarism

Information about plagiarism and other examples of academic misconduct can be found in the class guide and on the University's website.

- https://www.youtube.com/watch?v=TR7uE81vhK8
- http://www.gla.ac.uk/plagiarism/

[3] https://moodle.gla.ac.uk/mod/resource/view.php?id=1022590

A Interaction Specifications

A.1 Command Line Arguments

Below are sample invocations of the game program.

```
$ ./game --test
$ ./game --state <statefile>
$ ./game --test --state <statefile>
```

A.2 Command Prompt Commands

Below we describe the minimum list of commands with their expected response. Please consult the provided test suite for a list of example interactions.

`:help`	Displays the list of available commands together with a description.
`:exit`	Exits the game prompt, displaying the text: `Good Bye!`
`:start`	Begins a game, displaying the text: `Starting Game.`
`:stop`	Ends a game, displaying the text: `Stopping Game.`
`:move src dst`	Move a piece from source position to destination position. Co-ordinates are specified in Algebraic Notation[4], i.e. from bottom left corner *a1* to top right corner *i9*. Vertical columns are labelled with consecutive letters a-i, horizontal rows are labelled with consecutive integers 1–9. Refer back to Figure 1 for more detail. The response is either: `Invalid Move!`; or `Move Successful`. If the move affects the outcome of the game the program will output either: `Lambdas Win` or `Objects Win`. The move command cannot be used when a game is not in session.
`:save fname`	Save the current game state to the provided file name. Responds with `State saved in <fname>`; or `Cannot save game`. The save command cannot be used when a game is not in session.
`:load fname`	Load the external game state from the provided file name, and resume playing the game only if the file has been successfully loaded. Responds with `State loaded from <fname>`; `Malformed Game State`; or `Cannot load saved game state`.

If any of the commands have been entered incorrectly i.e. are malformed, the program shall respond with:

```
The entered command was malformed.
```

If an unknown command has been entered, the program shall respond with:

```
The entered command was not recognised.
```

If a command has been entered when the program is not ready for the command, e.g. calling `:move` when a games has not be started, the program shall respond with:

```
The command cannot be used.
```

[4] https://en.wikipedia.org/wiki/Algebraic_notation_(chess)

The prompt shall take the following forms:

`tafl>`	when not in a game
`tafl game O>`	when in game and the turn of Team Object.
`tafl game L>`	when in game and the turn of Team λ

A.3 Game State Serialization Format

The Game State shall be serialized as a *Comma Separated Values* (CSV) file with ten lines. The first line will have the form: `O to play` or `G to play` to indicate whose turn it is to move on the next round. There will be 9 lines following this initial line. Each of these 9 lines has 9 columns. The cell at position (5) on the middle of these 9 lines will contain either an `L` to indicate that the λ is in the central position; or an `X` to indicate that the λ has left the central square. The remaining cells can contain no more than: eight guards indicated by `G`; and sixteen objects indicated by `O`. There can only be at most one λ on the board. If a cell is not occupied by a piece, then insert a space character. The first line should have exactly 9 characters on it, before the newline. The subsequent 9 lines should have exactly 17 characters on each line, before the newline. The final character in the file should be a newline, terminating the final line of the board. A `wc -c` command on a saved game csv file should always return 172 characters. A `wc -l` command on the same file should always return 10 lines. Any other CSV files should cause error messages when loaded. malformed

Figure 5 presents an example CSV file, indicating the starting state of the game. Please see the tests for more example game states.

```
O to play
 , , ,O,O,O, , ,
 , , , ,O, , , ,
 , , , ,G, , , ,
O, , , ,G, , , ,O
O,O,G,G,L,G,G,O,O
O, , , ,G, , , ,O
 , , , ,G, , , ,
 , , , ,O, , , ,
 , , ,O,O,O, , ,
```

Figure 5: Example `CSV` File showing Saved Game State.

A.4 Changelog

- 2018-11-07: Initial spec release