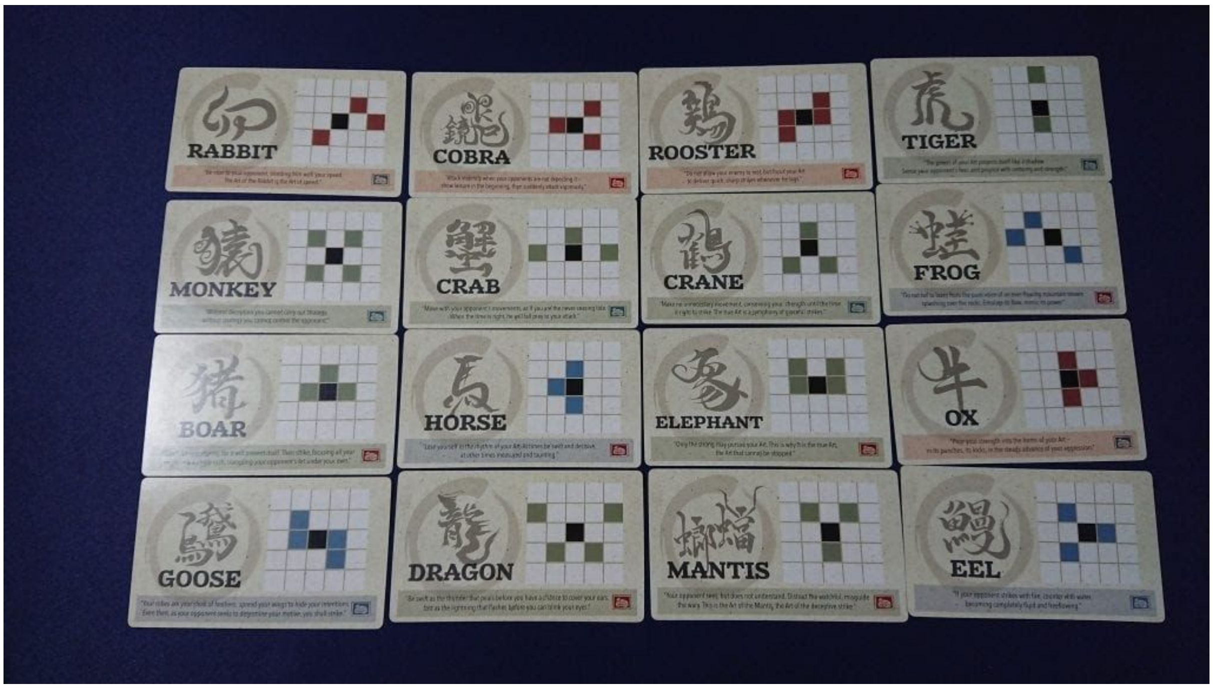
Onitama

Onitama is a strategic board game for two created in 2014 by Japanese. The game is played on a 5x5 board. Each player receives five pieces, one masterpiece placed in the middle back row, known as the Shrine, and four pupils, 2 placed on each side of the master. 16 movement cards are provided to dictate the movement of the pieces during the game. Five of the 16 movement cards are utilized per game.

The goal of the game is to capture the opponent's masterpiece, known as Way of the Stone, or move your masterpiece into your opponent's Shrine, known as the Way of the Stream. Players shuffle the 16 movement cards and then give two movement cards to each player and one movement card is placed on the side of the board for a total of 5 movement cards in play. Each piece can be captured and captures other pieces the same way which is based on the movement provided by the movement cards (i.e., if it lands on an opponent's piece, the opponent's piece is captured and removed from the board). After one player moves, he or she must replace the movement card at the side with the used movement card, allowing both players rotating access to all 5 movement cards in play.

The list of all the available cards is the following.



The project

You are requested to model in Haskell the game of Onitama performing the following tasks:

1. Generate valid random games
2. Validate a sequence of moves (i.e., if all the moves follow the rules game)
3. Generate unit test to cover all the statements of the program
4. (Optional) Given a number n and a game state, determine if there is a winning strategy for one of the players in less than n moves.

The project is individual.

Game External Representation

For checking the input and output of the functions required by this project, a game is represented using textual files following the following format.

The first line of the file contains a tuple (Cards, PiecesA, PiecesB) that represents the initial state of the game.

- Cards is a list of 5 strings representing the cards available for the game. In particular, the first two cards belong to the first player, the following two cards belong to the second player, and the last fifth card is the one on the side. The card name should be a string starting with a capital letter, followed by lowercase letters.
  As an example ["Cobra","Rabbit","Rooster", Tiger","Monkey"] represent an initial state where the first player has the cards Cobra and Rabbit, the second player has the cards Rooster and Tiger, and the card Monkey is on the side. The order in which the cards of a player are presented should follow a lexicographical order (e.g., ["Rabbit", "Cobra","Rooster","Tiger","Monkey"] is not correct since "Cobra" is lexicographically smaller than "Rabbit")
- PiecesA and PiecesB represent the position of the pieces for the first player and the second player respectively. PiecesA and PiecesB is a list of pairs of natural numbers. The position of a piece is represented with the pair of coordinates, i.e., (X,Y) represents the piece on row X and column Y (row and column numbers start from 0). The first element on a Pieces list represents the coordinates of the masterpiece. The remaining pieces are listed according to their coordinates sorted in lexicographic order. As an example, the initial position for the first player at the beginning of the game can be represented as [(0,2),(0,0),(0,1),(0,3),(0,4)]

From the second line on, lines represent moves (one move per line).
A move is represented with a tuple (Start, End, Card) where Start represents the coordinates of the piece that has been moved, End represents the coordinates of the piece after the move, and Card is the card used.

For example, the move that takes the masterpiece in the initial situation and applies the card Rabbit can be encoded as follows ((0,2),(1,3),"Rabbit")

To summarize, the file containing the following two lines is a valid file representing a game where the first player does the Rabbit move.

(["Cobra","Rabbit","Rooster","Tiger","Monkey"],[(0,2),(0,0),(0,1),(0,3),(0,4)],[(4,2),(4,0),(4,1),(4,3),(4,4)])\n((0,2),(1,3),"Rabbit")

Functions to implement

The following functions should be implemented.

- generateRandom :: Integer → Integer → IO (String)
  This function takes an integer seed, an integer n, and outputs a string representing a game with n moves computed randomly where the random generator has been initialized with seed. Assume that the only cards available are Rabbit, Cobra, Rooster, Tiger, Monkey. The moves requested can be less than n when the last move is a winning move for one of the two players.
- isValid :: String -> IO (String)
  Assume that the only cards available are Rabbit, Cobra, Rooster, Tiger, Monkey. This function takes a file that is supposed to encode a game and outputs "NonValid X" where X is the first move that is not valid when the game is not valid, "ParsingError" if the file does not represent a game. In case the game is valid the function should instead return the representation of the final state according to the format used to represent the initial game (i.e., the first line of a game file - see section before).
  Note that in the case a player wins the game, the list representing the pieces of the opponent should be the empty list. Remember that the state representation coordinates depend on the next player to play (i.e., a (0.0) coordinate from a player A is a (4,4) coordinate for the second player).
  Note that the function should report "ParsingError" if the game state given in input is not formatted correctly (e.g., when a string is found instead of an integer) or is wrong (e.g., pieces overlap, a piece has wrong coordinates like (4,5), cards of the player are not sorted lexicographically, a player has more than 5 pieces).

Optional (small) additional complexity: implement the previous two functions considering all the 16 possible cards

Optional exercise: implement the following function for obtaining more points
hasWinningStrategy:: Integer → String → IO (String)

This function is supposed to take an integer n, a file name s representing a game. It should parse the first line of s representing the initial state of the game (note that it is possible to consider initial games that start with fewer pieces located also in non-standard positions) and return "FirstPlayer" if the first player has a winning strategy with less than n moves, "SecondPlayer" if the second player has a winning strategy with less than n moves, "None" if no players have a winning strategy with less than n moves, "ParsingError" if the first line of the file does not represent a valid game state. Note that having a winning strategy with less than n moves, means that the player can secure the win in less than n moves against any possible moves by the opponent.

How to submit the project

The project must be developed using Stack (https://docs.haskellstack.org/en/stable/README/). Stack allows you to focus on the code and provide facilities for deploying the code and testing it.

For detailed instructions on how to use Stack, please read the documentation and in particular the Quick Start guide available at https://docs.haskellstack.org/en/stable/README/#quick-start-guide.

You will receive a zip file (my-project.zip) containing a folder created by using stack.
The zip file contains among other files the app/Main.hs, src/Lib.hs, and test/Spec.hs files that are all Haskell source files that compose the actual functionality of the project. The app/Main.hs and test/Spec.hs are already defined. The src/Lib.hs is the file that you should modify in order to implement the functionalities of the project. In case you need to use additional modules, the dependencies are automatically handled by Stack if you add them into the file package.yaml.

Lib.hs should contain the definition of the generateRandom, isValid, and hasWinningStrategy functions (if implemented).

app/Main.hs is the provided files that contains the code to execute the application from the command line. The application should be invoked from the my-project directory with

```
stack my-project-exe -- [-f|--file FILEPATH] COMMAND
```

The command can be a choice between generateRandom, isValid, hasWinningStrategy followed by the arguments of the corresponding functions. The command then calls the function passed as the first argument and prints its output in standard output. In the case the option -f is passed, the output is saved in the file given as optional argument.

For more information run the command

```
stack my-project-exe -- --help
```

Please note the usage of "--" before the sequence of flags and arguments to pass to the haskell program!

test/Spec.hs is the program to run the tests. The main function triggers the tests for all the generateRandom, isValid, and hasWinningStrategy functions. In the my-project folder you can find a folder called "unit_tests". In this folder, there are three subdirectories called generateRandom, isValid, and hasWinningStrategy. These are the folders where the test should be stored.

For the generateRandom function, the Xth test is captured by the presence of two files: X.out and X.param in the folder generateRandom. The file X.param should have two lines, each of them containing an integer that represents the two integer arguments of the function generateRandom. The X.out should contain the output of the "generateRandom Y Z" function where Y and Z are the integers respectively in the first and second line of the X.param file.
The test will pass if X.out contains the output of "generateRandom Y Z" and if this output is valid (i.e., if you run the function isValid on the output of "generateRandom Y Z" its results should be different from "NotValid" and "ParsingError").
Note: in the folder generateRandom of the folder unit_tst the files 1.out and 1.param have been added as an example. These files should be removed and replaced with the appropriate ones when the function generateRandom is properly developed.

For the isValid function, the Xth test is captured by the presence of two files: X.in and X.out in the folder isValid. X.in is the input file for the function isValid while X.out is the file containing the string that the function computes. A test passes if the application of "isValid X.in" generates the string saved in X.out.

For the hasWinningStrategy function, the Xth test is captured by the presence of three files: X.in, X.param, and X.out in the folder hasWinningStrategy. Similarly to what is done for the function generateRandom, the file X.param specifies the integer parameter of the function hasWinningStrategy. In this case the X.param should have only one line containing one integer. A test passes if the application "hasWinningStrategy Y X.in" generates the string saved in X.out where Y is the integer saved in the first line of the X.param file.
Note that the folder hasWinningStrategy should be empty if the hasWinningStrategy is not implemented.

Invoking the main function in test/Spec.hs runs all the tests in all the folders sequentially. The tests should be used to cover all the possible paths of the program. To run the test and check the coverage it is possible to use the following command from the my-project folder.

```
stack test --coverage
```

This command runs the tests checking the coverage information. It is important to reach a 100% coverage for the expressions, local declaration, and the alternatives used. The output of the previous command should, therefore, include the following three lines.

```
100% expressions used ...
100% alternatives used ...
100% local declarations used ...
```

If the 100% coverage is not possible, then the README.md file present in the main directory of the project should contain an explanation of why the 100% coverage was not possible (note that there is no need to reach the 100% for the top-level declarations) .

For more information about the test coverage please have a look at:

- https://docs.haskellstack.org/en/stable/coverage/
- https://wiki.haskell.org/Haskell_program_coverage

By using the Haskell Program Coverage utility launched with stack it is for example possible even to inspect visually the paths that are not covered yet. This may be extremely useful to understand what test should be added to cover all possible paths.

The README.md file of the project should describe briefly how a game is represented internally, the main challenges for the development of the program and how these have been overcome (max 3000 chars including spaces and punctuation). If the coverage of the test is not 100% the readme should contain a discussion on why this was not achieved.

To test your application, before submission, you should submit it for a test using the following URL: https://dm552.sdu.dk/. You have to submit your my-project.zip file and tests will be run to check if there are simple mistakes that you have to address. When submitting you will receive a code that you can use later to check if your submission has passed the checks. When the application passes the checks, it can be submitted in Blackboard. The same zip file submitted to https://dm552.sdu.dk should be uploaded.
Note that you can submit your application multiple times to https://dm552.sdu.dk . Due to the nature of the test that can take up to 5 minutes there may be delays in giving back your result in case all of you submitted at once. So please do not wait the last minute to test your application and have a bit of patience when checking the results. Note also that passing the tests does not imply automatically that you have fulfilled the assignment. The tests are only some simple automatic tests performed to help you avoid some mistakes. The assignments will be checked also manually after the final submission.
In case of problems with the testing platform feel free to contact us (Henrik, Jonas, Jacopo).

The first version of the code, after tests, should be submitted via Blackboard before May 24th, 23.59. Then you will have 1 week of time to share your tests (only the test) with the other students to spot and fix mistakes, if any. The final code (my-project.zip) should be submitted within the 31st of May using blackboard (SDU Assignment).

Strict Deadline: 31st of May, 23.59 CET Time.

Final Requirements

Minimal requirements:

- There should be at least 5 different tests for all the implemented functions. If all the 16 cards are used (additional difficulty) one test should include the card Eel.
- The code should build (i.e., running the command "stack build") and the execution of "stack test --coverage" should run.
- All the main functions defined should have a type signature.

- The source code should be commented and documented (e.g., all the major functions/data types introduced should have a textual description to summarize what they do)
- 100% coverage of the code (if not, the explanation why it was not achieved should be given in the README.md)

Failing to do any single of these points can cause the project to not be taken into account for the exam grade.

Preferences:

- Conciseness of the code would be favored
- Minimality of number and execution time of the tests to reach the 100% coverage should be favored (test should not run for more than 5 minutes on commodity hardware).

While the project is individual, you are welcome to share the tests for the isValid and hasWinningStrategy functions with the other participants of the course in the interest to cross-validate the correctness of your solution. Note that there is no point in sharing instead the generateRandom tests since the random choice of the moves may depend on the internal representation and different from project to project.