DM552 Programming Languages
Project Assignment

March 9, 2022

Contents

1 Onitama

Onitama is a two-player strategy board game created in 2014 by Japanese game designer Shimpei Sato. The game is played on a 5x5 board. Each player controls 5 pawns – 1 **master pawn** and 4 **student pawns**. 5 out of the total 16 **move cards** dictate the movement of the pawns during the game.

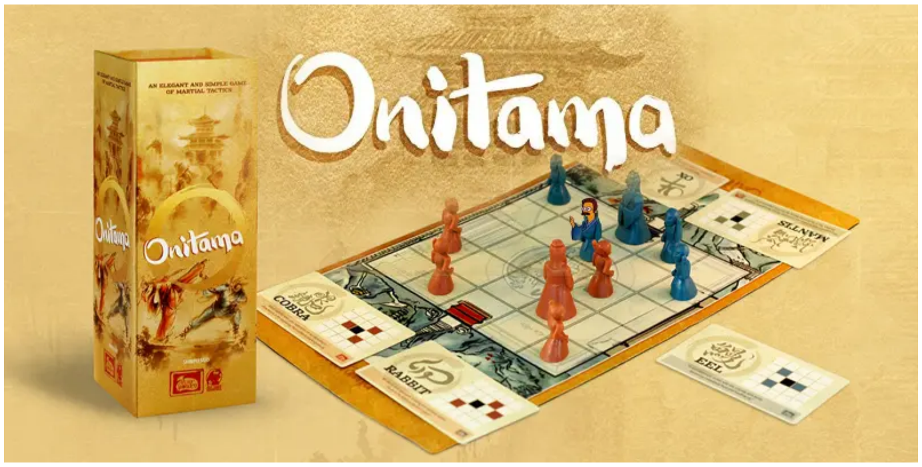

Figure 1: Onitama board game

Initially, each player's master pawn is placed in the middle square of their closest row (known as the **shrine** or **temple**), with the 4 students beside it. The 16 cards (see Figure 2) are shuffled and 5 are chosen randomly, with each player receiving 2 cards and 1 being set aside by the board. The remaining cards are not used. Both players place their 2 cards face up in front of them, while the stamp (red or blue) of the 5th card determines which player goes first.

Each player's turn consists of using one of the 2 available move cards to move any of their pawns. The move is applied relative to the **player's perspective** and the position of the pawn, with the dark colored square on the card representing the pawn's position. Landing on an opponent's pawn captures the pawn and removes it from the game. Moving out of the board or onto one of your own pawns is forbidden. After playing a move, the played card is swapped with the 5th card and the player's turn ends.

A player can win in two ways – by capturing the opponent's master pawn, known as the **Way of the Stone**, or by moving their own master pawn into the opponent's temple, known as the **Way of the Stream**.

Take a look at the provided rulebook for more details. Additional resources can be found on Wikipedia and BoardGameGeek.

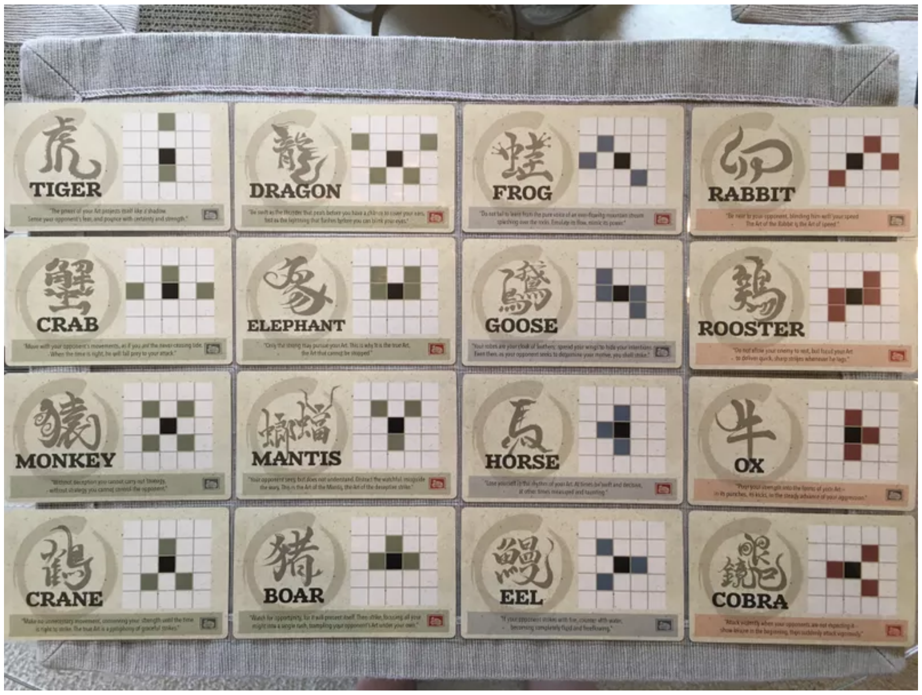

Figure 2: Onitama move cards

2 Assignment

Your assignment is to implement the core of a simulator for Onitama in Haskell. For simplicity's sake we will **ignore the ability to pass a turn**, which is allowed in the case when there are no valid moves to take (see page 8 in the rulebook). We will also ignore the move cards' **stamps**.

Additionally, along with the standard Onitama rules described in Section 1, we also introduce the "**super move**". When playing a turn, instead of playing a move corresponding to one of the 2 available move cards, the player can execute the super move. The super move allows the player to move a pawn onto any of its adjacent squares, both orthogonally and diagonally (i.e. onto any square 1 step away in any of the 8 possible directions). The player then chooses 1 of the 2 move cards to swap with the 5th card, as usual. Each player can use the super move **only once per game**.

Your code will have to implement the game's rules (with the addition of the super move) and provide functionality for simulating a game, generating random games and counting the number of possible games. Additionally, you will have to write unit tests and ensure good test coverage of your implementation. The details follow in Section 3.

The project is **individual** and cannot be done in groups, but sharing unit tests is allowed.

3 Implementation

3.1 Format

In order to give you as much flexibility as possible in the design of your internal representation while still maintaining a standard interface, your code will have to serialize and deserialize its game data into a common string format.

The string represents a single game. The format is line-based and contains one or more lines (separated by a single line feed character, commonly represented as `'\n'`). Empty lines are **ignored** and whitespace is **not significant** outside of double quotes.

The **first line** is of the form `(<cards>,<pawns1>,<pawns2>,<turn>,<sm1>,<sm2>)` and represents the **initial state**, with the 6 variables being placeholders for the following:

- `<cards>` represents a list of 5 cards used in the game. It has to be of the form `["<card1>",...,"<card5>"]`, where `<card1>` through `<card5>` are valid card names such as `Monkey`, `Rooster`, etc. The first 2 cards belong to the first player, the next 2 to the second player, and the last card is the one beside the board. The 2 pairs of cards have to be **lexicographically** sorted.
- `<pawns1>` and `<pawns2>` represent the positions of the pawns for the first and second player, respectively. Each is a list of points of the form `[(<x1>,<y1>),...,(<x5>,<y5>)]`, containing, in order, the master pawn and a **lexicographically** sorted sequence of student pawns. If one of the players won, the other player's pawn list should be **empty**.

  The **coordinate system** used is from the **perspective of the first player**. The first player's bottom-left square (leftmost in their closest row) is the origin $(0, 0)$. For a point (x, y), x specifies the row and increases toward the second player, while y specifies the column and increases to the right.
- `<turn>` represents the current player, i.e. whose turn it is. It should be `0` for the first, and `1` for the second player.
- `<sm1>` and `<sm2>` represent whether each of the players have used their super move. The format of each is either `True` or `False`.

Any **remaining lines** are of the form `((<x1>,<y1>),(<x2>,<y2>),"<card>")` and represent the **moves** that the players have played, starting with the current player and alternating between the two. The placeholders are similar as above:

- `(<x1>,<y1>)` and `(<x2>,<y2>)` specify the starting and the final position of a pawn that has been moved.
- `<card>` must be a valid card name and among the 5 cards used in the game. Additionally, it can be prefixed with `Super␣` to denote the card has been used as the **super move**.

As an example, consider the initial state and the sequence of moves specified by the following snippet (the symbol ↪ represents a wrapped line due to typesetting and not an actual line feed character, and the symbol ␣ represents a space character):

```
(["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)],0,True,False)
((0,2),(1,3),"Rabbit")
((4,2),(3,3),"Super␣Tiger")
```

The initial state specifies that the first player has the cards Cobra and Rabbit, the second player has the cards Rooster and Tiger, and the 5th card is Monkey. The pawns are in their standard positions (which **does not** have to be the case in general) and it is the first player's turn.

Two moves are played. The first player plays his Rabbit card and moves his master pawn from $(0, 2)$ to $(1, 3)$. After that, the second player plays his Tiger card but decides to turn it into a super move in order to move his master pawn from $(4, 2)$ to $(3, 3)$.

The final state ends up being `(["Cobra","Monkey","Rabbit","Rooster","Tiger"],[(1,3)` `↪,(0,0),(0,1),(0,3),(0,4)],[(3,3),(4,0),(4,1),(4,3),(4,4)],0,True,True)`.

3.2 Interface

The interface your code has to implement consists of 3 Haskell functions implementing the following functionality:

- `simulateGame :: FilePath -> IO String`

  The function takes a path to a file and **simulates** the game specified by the file (following the common string format from Section 3.1), checking if it follows the described rules. The function should return:

  - the string `InvalidFormat` if the file is not readable (for whatever reason) or its content does not follow the common string format (missing initial state, wrong syntax, etc.),
  - the string `InvalidState` if any of the parameters of the initial state are invalid (invalid cards, overlapping pawns, etc.),
  - the string `InvalidMove␣<move>` if a particular move is invalid (invalid card, coordinates out of bounds, a player already won, etc.), where `<move>` is the invalid move, represented as usual as `((<x1>,<y1>),(<x2>,<y2>),"<card>")`,
  - a string representing the final game state (the same as the first line of the common string format) if both the initial state and the sequence of moves are valid and follow the game's rules.

- `generateGame :: Integer -> Integer -> String`

  The function takes two integers s and n (in that order). It has to generate a random **valid** game with **at most** n moves (or less if the game ends in a win or there are no remaining valid moves), seeded with the random seed s. The function should return the game as a string (following the common string format from Section 3.1).

  You can generate the game in whichever way you want, as long as the generated game is valid. Note the absence of `IO` in the return type which makes the generation completely reproducible, i.e. **deterministic** with respect to the random seed s.

  You can assume $n \geq 0$ or raise an error otherwise.

- `countGames :: Integer -> FilePath -> IO String`

  The function takes an integer n and a path to a file that specifies a game (following the common string format from Section 3.1). However, only the initial state is significant and any specified moves are **not important**.

  The function has to count and categorize the number of **possible valid games** that can be played if starting from the initial state and using at most n moves (or less if the game ends in a win or there are no remaining valid moves).

  In other words, the function has to, at every turn, consider all of the possible moves a player can make and count all of the **possible sequences of moves** that could potentially take place.

  The function should return:

  – the string `InvalidFormat` or `InvalidState` in case of invalid input, as before,

  – a tuple `(<c>, <c1>, <c2>)`, where `<c>` is the total number of possible games, while `<c1>` and `<c2>` are the number of games that end up in a win for the first or the second player, respectively.

  You can assume $n \geq 0$ or raise an error otherwise.

3.3 Development

To aid development we will make use of a widely-used Haskell build tool called Stack. Using Stack will allow you to easily build and test your project.

For a quick introduction to Stack please read the Quick Start Guide and get familiar with a couple of its commands. Essential commands to know about are **`stack build`** and **`stack test`**. `stack exec` and `stack repl` (equivalently `stack ghci`) are also useful to know about.

For detailed instructions feel free to explore the documentation further.

Instead of manually creating a new Stack project, you will receive a ZIP archive containing the project template. Some of the important files are:

- `package.yaml`

  This file contains the project's metadata and a list of all of the required dependencies. If for some reason you want to use an additional Haskell package as your dependency, you can add it under the appropriate list in this file, but make sure **not to remove** any of the existing dependencies.

- `stack.yaml`

  This file contains some additional Stack-specific configuration which you **should not modify**.

- `src/Lib.hs`

  This is the file where you will write your implementation of the 3 required Haskell functions. The file will already contain dummy definitions which you will fill out over time. For ease of testing and evaluating the assignment, we require that the functions are defined and exported from **this particular file**. You're of course allowed to write your own internal utilities and helper functions, but **don't modify** the export list of this file.

- `app/Main.hs`

  This file implements a small command-line utility that allows you to invoke one of your 3 functions with the parameters specified over the command-line. This can come in handy when you want to quickly test your function from the command-line.

  Use `stack exec my-project-exe -- [-f <filepath>] <command> <args>` to invoke a function, where:

  - `<command>` is one of `simulateGame`, `generateGame` or `countGames`, and
  - `<args>` is a space-separated list of arguments to pass to the corresponding function.

  The function's result will be output to standard output. If you'd like to save the output to a file, you can use a feature called "output redirection" if your shell supports it, or optionally pass the `<filepath>` argument (using the `-f` flag) naming the file.

  You can see the full help by running `stack exec my-project-exe -- --help`.

  Please note the usage of `--` before the sequence of flags and arguments that are passed to the Haskell program.

- `test/Spec.hs`

  This file implements a small "test runner" which is invoked by Stack in order to test your project when you execute `stack test`. You **should not modify** this file.

  Details regarding the structure of the tests are given in Section 3.4.

A useful command to also keep in mind during development is `stack repl`. This command will drop you into a Haskell REPL with your project loaded. From there you can experiment with and manually test your Haskell functions and enter arbitrary Haskell code.

3.4 Unit Tests

Executing `stack test` will run the "test runner" in `test/Spec.hs` which will search the `unit_tests` directory and execute all of the tests it finds. The unit tests are stored as **plain text files** and specify the input given to each of the functions and the expected output the functions should produce. If the function doesn't produce the correct output, an error is reported.

`unit_tests` contains 3 subdirectories, one for each of the functions you have to implement. The structure of each of the tests is the following:

- `unit_tests/simulateGame`

  Each test consists of two files: `<test>.in` and `<test>.out`.

  `<test>.in` should specify a game according to the common string format. Its path is passed to `simulateGame` and the resulting value is then compared to the content of `<test>.out`.

- `unit_tests/generateGame`

  Each test consists of two files: `<test>.param` and `<test>.out`.

  `<test>.param` should contain a line-separated list of 2 integers that are the parameters to `generateGame`. The generated game is then compared to the content of `<test>.out`.

- `unit_tests/countGames`

  Each test consists of three files: `<test>.param`, `<test>.in` and `<test>.out`.

  `<test>.param` should contain a single integer parameter that is passed to `countGames` along with the path to `<test>.in`. The resulting value is then compared to the content of `<test>.out`.

3.5 Code Coverage

Running `stack test --coverage` will make Stack report how well your unit tests **cover** all of the possible execution paths of the code you've written. Stack does this by using the Haskell Program Coverage (`hpc`) tool. For more information about coverage please have a look at Stack's documentation and the Haskell Wiki.

It will be important for you to write as many unit tests as necessary until you reach 100% coverage in the following categories reported by the coverage tool: **expressions**, **alternatives** and **local declarations**. The other categories (top-level declarations, etc.) you can safely ignore.

A thing to note is that `stack test --coverage` outputs the path to an HTML report generated by the coverage tool. The report gives you a **very useful** visualization of your source code annotated with colors. Yellow denotes what hasn't been covered so far, while green and red denote boolean expressions which have been observed to always be either true or false, respectively.

3.6 Autograder

While developing your implementation, along with building and testing locally, you will also make use of **DM552's online autograder**. You submit a ZIP archive of your project and the autograder compiles it, runs an additional suite of tests and gives you back a report. This will help you catch common mistakes and give you some feedback while you're working on the project, but also make it easier for us to evaluate your assignment.

You can submit your project for evaluation as many times as you want. However, depending on the server load and the number of people submitting their projects, it might take a couple of minutes for your results to come in. Therefore, don't wait until the last minute to test your implementation and be patient when checking the results. Also be aware that the **maximum allowed evaluation time is around 5 minutes**, so make sure your code terminates in a reasonable amount of time.

Passing the autograder test suite **does not guarantee** you've completed the assignment, as it will still be checked manually after the final deadline. In case of problems with the autograder, feel free to contact us.

4 Hand-in Deadlines

You will have to develop and hand in your implementation incrementally. Assignments are handed in via Itslearning and consist of submitting a ZIP archive of your project, **and a hash of your autograder submission as a comment** (see Section 3.6).

Each hand-in deadline has a set of associated requirements that you should read **carefully**:

- **6 April (`simulateGameAux`)**

  An internal representation of the game's state and moves should be implemented. As a first step toward `simulateGame`, there should also exist an implementation of a function with a simpler interface:

  ```
  simulateGameAux :: <S> -> [<M>] -> String.
  ```

  Instead of parsing a file which specifies a game, `simulateGameAux` works directly on the internal representation that you chose. `<S>` and `<M>` are whatever types you implemented to represent the game's state and moves.

  The function receives the initial state and the sequence of moves and returns the same thing as `simulateGame` would. Unit tests can be written but are not yet required, and the autograder tests don't have to pass for now.

- **25 April (`simulateGame`)**

  The function `simulateGame` should now be implemented (possibly by reusing `simulateGameAux`) from before. All of the `simulateGame` tests from the autograder suite should be passed.

- **2 May (`simulateGame` + 100% coverage)**

  All of the requirements from the previous deadline apply. The code coverage should reach 100% as described in Section 3.5. If it doesn't, use the `README.md` file to explain why.

- **11 May (all functions)**

  All of the functions should be implemented and pass all of the autograder tests.

- **18 May (all functions + 100% coverage)**

  All of the requirements from the previous deadline apply. The code coverage should reach 100% as described in Section 3.5. If it doesn't, use the `README.md` file to explain why.

  All of the main functions should have type signatures and source code should be documented with comments in places where necessary. In general, strive to write clean and concise code which is clear even without comments.

  The `README.md` file should briefly describe the internal representation, the main challenges encountered and how they were overcome (max. 3000 characters).

The intermediate hand-ins will mostly rely on the automatic autograder tests, but **the final hand-ins will be checked and graded manually**.

You are welcome to submit your project to the autograder at any time, even long before the upcoming deadline. If you've already implemented the functionality for a future deadline other than the upcoming one, it is perfectly fine to hand in the same assignment multiple times.

You are allowed to miss a single intermediate deadline without any explanation. Missing two intermediate deadlines requires you to contact the professor and explain why you didn't manage to complete the work on time. **Missing the final deadline results in failing the project** (and therefore not being admitted to the exam).

5 FAQ

- **Why read data from files instead of just strings?**

  The proposed interfaces could as well avoid working with files and just use plain strings, however, one of the goals of the project is to get you to use and understand Haskell's IO facilities.

- **Do I need to reach 100% boolean coverage?**

  No, you don't need to have 100% boolean coverage. The Haskell coverage tool can report less than 100% boolean coverage when you have guards or cases in which the last condition always evaluates to true. This is normal and not a problem. You can inspect this visually by opening the HTML file generated by the coverage tool and looking at the expressions marked with green, indicating that they always evaluate to true.

- **Why does the log produced by the autograder contain backslashes?**

  The autograder uses the Haskell `show` function before outputting any Haskell values to the log. In particular, this is done when printing a test's input, parameters, expected output, etc. Since all of these values are of type `String`, the `show` function will introduce a layer of quotation when representing them, and might use backslashes in order to escape any double quote characters that are present in the strings themselves.

  If you're seeing double or triple backslash sequences you probably have an extra call to `show` somewhere in your code.

- **The autograder is telling me that `Building the project failed or it took more than 300 seconds.`**

  We've seen this error with a few students where the `stack.yaml` file was somehow updated such that it's incompatible with the Stack version we run on the server.

Try restoring the original version of the file by copying it from the project template. If the error persists, feel free to contact us.

- **My coverage report tells me that some number literals haven't been covered (e.g. number literals `1` in `-1` in yellow).**

  This outcome is a current limitation of the coverage tool. If you encounter this problem, please mention this in your `README.md`. If only literals are reported not covered, this is a valid reason for not reaching the 100 % coverage.

- **My code is failing autograder's `normal-4` test for `countGames`.**

  `countGames` might be tricky to debug in general. If it helps, here are the 6 move sequences that cause the first player to win:

  ```
  ((0,2),(1,3),"Cobra"),((4,2),(2,2),"Tiger"),((1,3),(2,2),"Monkey")
  ((0,0),(1,1),"Cobra"),((4,2),(2,2),"Tiger"),((1,1),(2,2),"Monkey")
  ((0,0),(1,1),"Cobra"),((4,2),(2,2),"Tiger"),((1,1),(2,2),"Rabbit")
  ((0,2),(1,3),"Rabbit"),((4,2),(2,2),"Tiger"),((1,3),(2,2),"Monkey")
  ((0,0),(1,1),"Rabbit"),((4,2),(2,2),"Tiger"),((1,1),(2,2),"Cobra")
  ((0,0),(1,1),"Rabbit"),((4,2),(2,2),"Tiger"),((1,1),(2,2),"Monkey")
  ```