Team

Eric Johnson (edj36), Pehuén Moure (ppm44), Kenta Takatsu (kt426), Eric Zhang (ez79)

System Description

Idea

Multi-player implementation of Scrabble

Key Features

- AI implementation that evaluates possible words and corresponding placements while calculating the score each combination generates, used to play against a person.
- Implementation of connection from one front-end client to back-end server.
- Implementation of connection from the back-end server to multiple front-end client instances at once.
- Server side logic to implement game-play and keeping score. The server side will take one user's input at a time and then process and display their input on the game board for all other players. If there is only one player then the server side logic will use the use the AI as the other player.
- Terminal based interface that displays the game board to the user. The front-end will accept inputs from the user when they are taking their turn.
- Word checking dictionary with random letter generator. This feature will evaluate player's moves and determine if they are valid. Valid moves will be placed on the game-board and all other player's boards will be updated. After each turn the game will randomly add a letter to the player's list.

Description

We intend to build a multi-player Scrabble game, that can be played with another person on the same computer or against an AI. The game will be played through either a terminal window with a simple GUI. We will have the option for a player to compete against friends in the game, or against a number of AI bots. The game will be able to handle up to four players at a time. Finally we will have a robust implementation of a data structure that allows us to effectively query a dictionary of all the words in the language defined by the game, Scrabble, for validating players moves and checking that the words the enter are valid in the "Scrabble" language.

Game play will be similar to as if you are playing Scrabble in person, but the application will automate some of the process for a player to move. When it is a player's turn to move, they will enter the word they want to play, the coordinates of the start letter, and the direction their playing. Once they have entered their word, the game will check that the word is valid in the Scrabble dictionary, and also that the word is possible to form given the current game state. For example, if the player has letters {`b, a, s, k, e, t`} in their tray, and they enter `base`, the game would allow that, but if they enter `ball` or `bsk`, the game would let the player know that neither word was valid, the first because the player does not have the letters to form `ball`, and the second because the word `bsk` is not a word in the Scrabble dictionary. After choosing a valid move, the player will submit the move to the game, and the game will update with that move. If the player cannot form a word, they will submit "Draw" (or a similar command) and can pass or swap letters on their board. Game play continues in this fashion until someone wins the game, someone quits the game, or the game cannot be continued.

Architecture

- **GUI** This component is the physical interface displaying the game state (with scores, tiles available, etc.) to the players of the game.
- **Repl** This component implements the Read-Eval-Print-Loop functionality for the app. This allows game-play to occur, with players inputting their respective moves during their respective turns via this module.
- **Player** This component will be an interface for Human and AI. Both modules need to be able to "execute moves" in the game, and the fact that they will be required to satisfy this interface guarantees that the game state is properly updated.
- **Tree** This will serve as the implementation of the Scrabble Dictionary. It will serve to efficiently check if a word is valid in the game.
- **Filter** This component will serve as a filter for the user inputs. It will be used to validate moves before passing them onto the server.
- **State** This component will be the "back-end" of the game. Whenever a valid request is made it will update the game-state (including game board).
- **Data** This component will store the data structures used throughout the game. For our purposes, it is cleaner to define the structures all in one file, than defining and re-defining them everywhere they are used.
- **Utils** This component will serve as a Library of sorts for commonly used helper functions. This includes specially tailored requests that we make in multiple different spots in the code for our game. Again, in order to unify our design as much as possible, we felt that having this central library of helper functions made sense, as opposed to implementing a multitude of functions a number of times.
- **Makefile** We plan to inlude a Makefile in order to standardize our build commands. We included files that have been provided to us in past assignments, and we have implemented the "make check" commands to check the Ocaml version numbers are correct.
- **Info** We store information about the game (help commands, error messages, etc.) in a JSON file. We chose to do it this way so that we could easily update strings we were displaying to the user without having to search our code to find where they were being called.

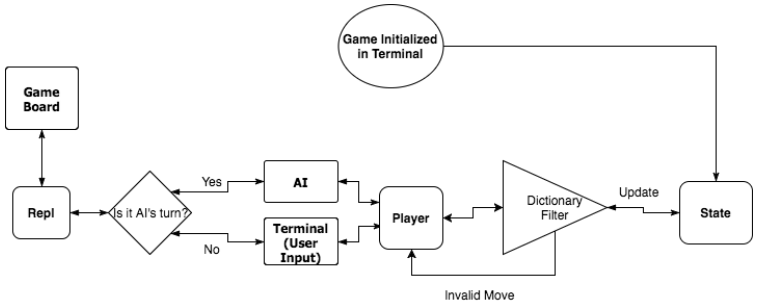

Figure 1: Components and Connectors

System Design

Important modules that will be implemented:

- **AI** This module is where the AI component of the game was implemented. This module implements the Player signature. The module evaluates possible moves and corresponding scores, and then pick a move to play. After choosing a move, AI "plays" the move in the game, and then await their next turn. The AI algorithm is genetic in nature. The algorithm functions by crawling until it finds a spot on the board that contains a tile. It then calculates the number of free space in all four compass directions (North, South, East, and West) from that coordinate. The number of free space is number of empty tiles between that tile and the boarder or a non empty tile. The AI then determines which direction to play a word from that spot prioritizing the direction with the most space. If the tile has words in any of the two of the adjacent compass directions then it will not play a word at that coordinate (as it would have to extend the existing word). If there is an equal number of words in either direction then it prioritizes the compass directions in this order: North, South, East, and West. After determining the direction it randomly selects that number of letters from its rack and finds all possible combinations of those letters. It then tests each of those words and sees if it is a valid move. If it is then it plays that word. If it does not then it repeats the process in the same direction but the number of spaces reduced by 1 until there is only 1 space left. If there is only 1 space left and no valid moves were found it crawls on to the next spot and repeats the entire process.
- **GUI** This module is the medium by which the user can see their progress in the game. It displays the game board, the user's score, and the tiles they have available for an upcoming move.
- **Human** This module represents a player playing the game. This is similar to AI, in that the rest of the game should not know whether a move was submitted by a human or by the computer, and both implement the Player signature, however, different from AI, the human will use the terminal to input the word for their next move.
- **State** This module serves as the "back-end" of the Scrabble game. This is where updating of the state will occur. The module will be given a valid move from Filter and will update the current game state, producing a new game state with correctly updated scores, etc. This includes a data structure representing the game, a mechanism to keep score for multiple players, and some functionality for updating the state as the game progresses.
- **Filter** This module will take input (from a user or from the AI) and evaluate it against the game rules, if the input passes, it will pass the move onto Server, otherwise, it will pass the input back to the "Player" (Human or AI) so that they can try another move.
- **Tree** This module will be implemented with a Radix tree, downloaded from OPAM. This allows us to efficiently query the "Scrabble Dictionary" to tell if words are valid for the board.
- **Utils** This module will be a Library for all of the helper functions that are used for more than one other module. This will help to organize and clean our code.
- **Game** This is the entry point for the game, it starts the REPL loop and GUI.
- **Repl** This is the Read-Eval-Print-Loop. It needs to carry out game-play after the game has been started.

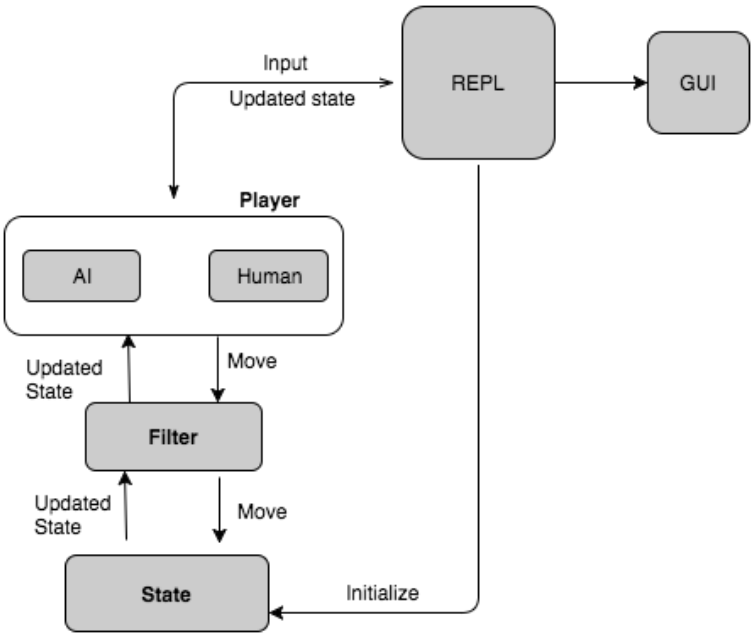

Figure 2: Module Dependency

Data

There is a large amount of data that needs to be kept tracked of as the Scrabble game progresses. Below are the structures and the proposed implementations for the components of our game.

- **Game State** We envisioned all the data associated with the current state of the game to be in a record, game state. This record will include:
  - Current Scrabble Board
  - Scoreboard
  - Rack of tiles for each players
  - Turn (whose turn it is)
  - Letter bag of tiles left for players to receive
- **Scrabble Board** is a 15 x 15 matrix of tiles. This will be implemented as a List of Lists of Tiles.
- **Tile** Each tile will have information about its location and if the tile offers bonus points. It will be implemented as a record with three fields: coordinates, bonus status, and if it is Occupied.
- **Bonus Status** is a variant that can be double letter, triple letter, double word, and triple word, the conditions for bonus points in the scrabble game.
- **Scrabble Dictionary** Dictionary of which potential moves can be compared to. If the word in the move is in the Scrabble dictionary, then it is a valid move. Since this dictionary will have a large amount of words, we need an efficient data structure to search through. We are going to use the built-in radixtree in OPAM to implement the dictionary.
- **Scoreboard** an association list that keeps track of the amount of points each player has. We chose to implement it this way because it is easy to update and it is simple to change the number of players.

- **Letterbag** will represent that letters that are left in play, and will be where the players draw letters into their racks. It will be represented as a list of letters. Each **Letter** is a record with two fields: character and points.
- **Player's Rack** Each player's rack will be represented as a list of letters that cannot exceed seven.

External Dependencies

- We used the ANSI Terminal Module to help render and update the game board. It was used exclusively in gui.mli.
- We used the Radix Tree Module provided to implement the Scrabble dictionary described above.
- We used Yojson for JSON parsing.

Testing Plan

We used testing as an instrumental part of the success of our project. We used both black-box testing as well as glass box testing because they both have their pros and cons, and using both methods will make our project more rigorous. We implemented tests on a constant basis for each function we write, ensuring that each previous function is correct before writing the subsequent one. Due to the nature of the project, a large portion of our testing will be play-testing the game in the terminal.

For each module, we will have different roles for the members of the team. There will be implementers and testers. The implementers will write the code for the module and make sure to include very well-defined specifications for all the functions. The tester will use the black-box method and will not know how each function is implemented, ensuring that the tests are not biased. These tests will focus on simple inputs as well as boundary cases. For example, in the case of testing the implementation of the Scrabble Board, the tester will first have simple letter input tests to see if it works, but also test to see if the proper error is raised if a number or other invalid input is used. One complication we might have with black-box testing is that, given each member has a general idea of how each component is implemented, it might be hard to keep unbiased; the tester will give his best effort to not let these biases influence the tests.

The glass-box testing will be done by the implementer of the code. Since the test has knowledge of the implementation, this way of testing can address errors that do not show up in the specification. With functions that use pattern matching, we will test using the path coverage method, creating test for each possible outcome of the function to test for correctness. For example, to test the Bonus Status variant, we will have test of each of its cases (double letter, double word, triple letter, triple word).

To make sure our testing plan does not fall through, we will have to hold each other accountable for creating quality tests. One way that we already hold each the team accountable is splitting roles during testing (tester vs. implementer), so the tester can check for correct code and specifications, while the implementer can check for quality test cases. Another way we will make sure that each member is committed to testing is having frequent meetings, where we will discuss the status of testing and constantly checking if each person is writing correct code.

Division of Labor

Kenta Takatsu

Kenta implemented a lot of the game logic, i.e., the scoring, the game board abstraction, and the moves among other things. Kenta was instrumental in the completion of the core game functionality. He was very helpful in making a lot of helper functions that the rest of the group used in their implementations as well. Specifically, Kenta worked on `state.ml`, `data.ml`, `gui.ml`, `filter.ml`, `game.ml`, `info.json`, `player.ml`,

and `utils.ml`.

Kenta worked approximately 25 hours on the project.

Eric Johnson

Eric worked on the game logic and the overall design of the system. Specifically, he was responsible for make sure the "pieces" of the game fit well together and that we had a good balance of modularity and interdependence within the code. Eric worked to make sure the flow of the game was easy to update and simple for multiple people to work on at once, and also planned/implemented the "filter" to make sure only valid moves were played in the game. Specifically, Eric worked on `data.ml`, `filter.ml`, `Makefile`, `utils.ml`

Eric worked approximately 20 hours on the project.

Eric Zhang

Eric worked on the Artificial Intelligence and the storage of the Scrabble Dictionary in the radix tree. In the beginning, he worked on the implementation of the original gameplay design and REPL loop. He later transitioned into implementing algorithms for the AI to play Scrabble, balancing its intelligence, efficency, and ease of implementation. Specifically, Eric worked on `data.ml`, `game.ml`, `tree.ml`, `player.ml`

Eric worked approximately 15 hours on the project.

Pehuén Moure

Pehuén took on specific roles involving the user interface and artificial intelligence portions of the application. For the user interface, he implemented and updated portions of our text-based GUI in order to make it more user-friendly, and to improve the overall user experience for our game. Additionally, he worked closely with Eric Z to implement the AI for our scrabble game. Specifically, Pehuén worked on `player.ml` and `gui.ml`.

Pehuén worked approximately 10 hours on the project.