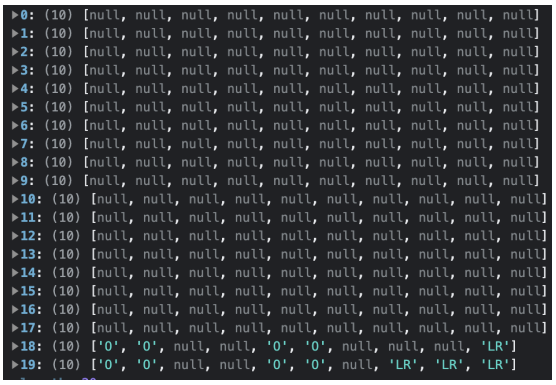
FIT2102 Assignment 1
Design Decision Report
By Chloe Ting, 33095108

Game Instructions

[w]: Cycle through rotations.
[a]: Move left.
[s]: Move down.
[d]: Move right.
[r]: Restart game.

Summary of Code

- The game board is represented by a 2-dimensional array for easy visualisation and future debugging, as shown below, where each string represents a block on the board. It is a custom generic Grid type to accommodate for various game configurations.
- The shape of each tetromino is visually represented by 1s among an array of 0s. This representation simplifies the process of mapping tetrominos onto the 2D game board when a collision is detected.
- A coordinate function is used to find the coordinate of each non-null cell (an active tetromino cube) in the game board.
    - This function plays a central role in collision detection and updating the game board during each tick, ensuring that the positions of active tetromino cubes are accurately tracked and managed.
- The code's use of an Action interface and the reduceState function allows for functional transformations of the game state.
- Rotations are handled by cycling through indices in the Tetromino constant. Rotations follow the Super Rotation System, as it seemed the easiest to implement due to its central axis.
    - With no wall kicks implemented, blocks simply do not rotate if at risk of collision, enhancing gameplay clarity and avoiding unintended rotations that could lead to collisions.



Design Justification

- **FP/FRP adherence**
    - Importantly, the code is highly immutable. The game state is managed as an immutable Readonly data structure, and new state objects are created when changes are needed. The code frequently creates new objects and copies data rather than mutating existing objects.
        - For instance, in functions like apply within various Action classes, the code creates new state objects with updated properties instead of modifying the original state in place.
        - This is also observed in the use of recursion in the Restart class, which avoids mutable state changes and instead creates a new state object each time.
    - Several small, granular functions are used to achieve high readability, reusability and modularity. For example, I define several helper functions that get called in clearRows. This aids in maintaining an adequate separation of concerns and ensuring the code is extendable for any future additional features.
    - Due to the code's high immutability, and because functions in the code are granular, functions that update the game state maintain purity. There exist very minimal side-effects after the use of functional update patterns.

- **The usage of Observables**
    - Various game interactions are represented as streams of events, making it easier to handle user inputs, animations, and game logic in a highly organised and modular manner.
        - I employed the interval function to create a regular time-based tick observable, facilitating the automatic downward movement of blocks in the game.
    - The code efficiently updates the game state in response to events using Observables.
        - For example, the scan operator accumulates state changes over time, and the subscribe method is used to react to changes in the state and update the game's visuals accordingly.

Identifying Issues

- The use of Date.now() as a pseudo-random seed is impure. I did define a global constant seed using Math.random(), but could not get it to work the way I intended.
- There could have been a more efficient use of HOFs and generic types. The Action classes could have been HOFs, making the code more extendable and flexible.

- Rotations for each tetromino are hard-coded. It's desirable to generalise operations to work with any data, which might involve a more abstract or algorithmic approach to rotations. There is limited flexibility with my current approach.