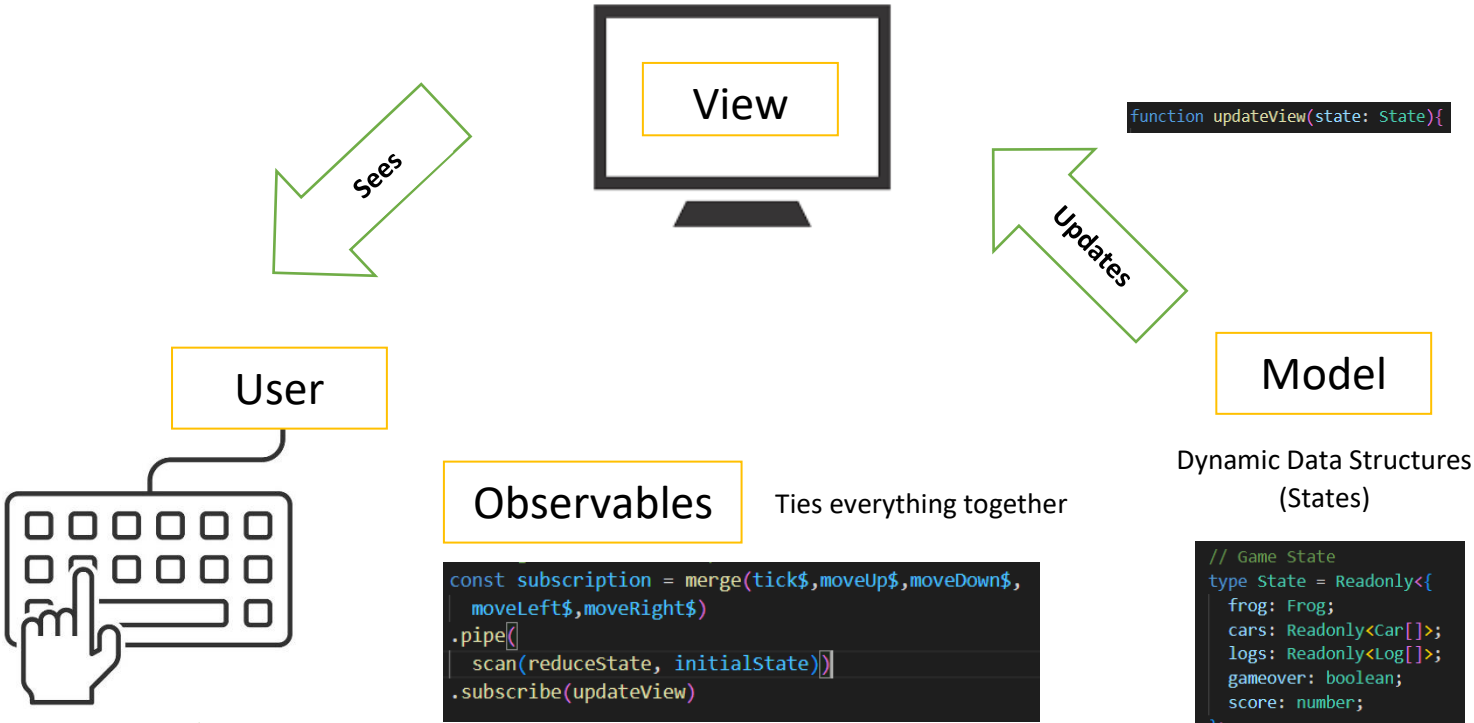
# Frogger Code Report

## MVC Architecture:

The code for my frogger game was designed using the Model-View-Controller (MVC) Architecture. The diagram below shows how I have utilized the Functional Reactive Programming (FRP) style by separating my objects into multiple states while using various inputs to manipulate them before ultimately tying everything together through the use of Observables.




```
function updateView(state: State){
```

```
const subscription = merge(tick$,moveUp$,moveDown$,
  moveLeft$,moveRight$)
.pipe(
  scan(reduceState, initialState))
.subscribe(updateView)
```

```
// Game State
type State = Readonly<{
  frog: Frog;
  cars: Readonly<Car[]>;
  logs: Readonly<Log[]>;
  gameover: boolean;
  score: number;
}>;
```

```
const reduceState = (currentState: State, event: Move | Tick): State => {
```

```
class Tick { // In-game tick
  constructor(public readonly elapsed: number) {}
}

class Move { // Up Down Left Right Movement depending on Keyboard Input (x-input / y-input)
  constructor(public readonly x_dist: number, public readonly y_dist: number) {}
}
```

## Model

From the MVC diagram above, we can see how each sector works with each other to make up the entire game of Frogger. Starting with the model, you can see how I utilized a dynamic data structure to make up the states of various objects and entities in the game. Moreover, the read-only typing of the properties prevents the user from directly modifying the properties of the objects in their most recent states, which definitely helps maintain purity throughout the code. While creating this model, I realized that I would eventually have to create objects using code of the similar structures. With that idea in mind, I decided to utilize tail recursion to keep creating the required numbers of objects with the same properties. For example, since all the cars had similar properties – such as id, x-coordinate, y-coordinate, and speed – I took advantage of the inner function parameters to recursively create multiple cars in multiple rows and columns using the same function while, at the same time, making each of their speed and color vary depending on their row number to spice up the difficulty of the game. Doing this saved me the trouble of having to create multiple of the same functions to add the rows of cars on the road. Aside from that, to make it easier to traverse through and understand my code, I utilized constants to replace the various values that would otherwise make it extremely difficult to understand their purpose. Most importantly, the functions I used to create the objects are pure as their properties would only change if their input parameters were modified. Otherwise, they would remain the same. With the “chess pieces” of the game set in place, we can now move on to talk about how the various objects in the game move around the canvas.

## User and Controller

In order for the objects to move around the game, we will first need a controller that handles all the inputs in the game. For this particular version of the game, the Move and Tick class provides constructors for the various functions we will mention later on and works separately from each other. While the move class will be particularly dealing with the key inputs of the users, the tick class is essentially used to create the game clock that handles all artificial movements of the objects throughout the state of the game. In the user movement inputs, I noticed that the code for each movement key would be similar in structure. Thus, I decided to use the keyObservable() function, which picks out a particular key and event for that particular movement. This function utilizes other functions from Observables such as filter to isolate the specific keys and events and map to set the action upon each input. With that set up, any objects that are set instances of the Move class would move a certain set distance in a particular direction depending on their respective observable stream as long as the game continues to run. Moving on to the Tick class, it was used in my code to provide the constructor for the main game clock. In my game, I purposely set interval for the observable

stream, tick, as 10 milliseconds to ensure the smoothness of the animations in the game. Essentially, the lower the tick interval, the more often, the game state will be updated. With all these functions and observable streams created, we can now work on the visuals of the game.

## Update View

Having created all the main functions, we can now define the initial game state of the various objects in the game such as the frog, cars, and logs. In order to make them move around the canvas, we will need to create a state reducer that will be referred to by the scan function in the main game stream to constantly update the game state. In this state reducer, the frog, which is controlled by the user, will be mainly considered as an instance of the Move class as it moves depending on the inputs of the user. On the other hand, the other objects that move constantly throughout the screen will be set as instances of the Tick class, where their properties would be updated every 10ms. Similarly, the collision handler would also be placed in the instance of Tick as collision between the frog and other objects would need to be checked constantly. Naturally, the frog would also need to be considered as an instance of the Tick class. In this part of the code, the x coordinates for the logs and car would constantly be checked as we would have to loop them pack around from the side of the car that their moving away from with each time they leave the screen.

As for the collision with the frogs and other objects, I utilized the concept of higher order functions. For example, the frogCarCollision function returns the helper function, checkCollisions, where it takes in the properties of the car and frog as the input parameters and returns a Boolean result. This way, we can keep reusing the function and make the code cleaner. This is a pure function as it will always return the same result as long as the input parameters do not change. Furthermore, I later on used filter to constant return the collision state of the frog and cars. Finally, I merged all the situation where the frog dies to a single Boolean variable that would constantly be checked in the reduce state section. In this part of the code, I purposely made it so that the user would have to manually stay on the planks to stay alive. This not only makes the game more challenging but also more fun. While these slices of code keeps on running in the background, the update view section of the code – otherwise called the view handler, where the only impure functions are being called – would actually be the one visualizing everything that's taking place on the main game canvas. In the impure view handler, the code will keep updating the visuals of the game and animations of the objects based on the most recent state of the objects in the state reducer. It is also in the part of the code where the game over status will be constantly checked. If so, all the observable streams will immediately be unsubscribed, marking the end of the game.

## Main Game Stream

Finally, in the main game stream, all the main observables are merged and piped into a scan function where the initial value is the initial game state, and for each action, it would transform and reduce the state of the objects in order to update them while subscribing to the updateView. Basically, the scan section constantly updates the properties of the objects while the update view changes the visuals accordingly. This is essentially where all the previously mentioned sections of the code were tied up as per in the diagram mentioned in the first page.