Summary of the Workings of the Code

The code makes use of observables to run the Frogger game by making use of the scan function. Streams of user moves and moving object ids are passed into reduceState in scan to make gameState objects. gameState objects represent the state of the game and the returned gameState in scan is used by updateView in the subscribe function to update the HTML according to the state. The frog is a green circle that moves left, right, up and down using a keyboard. Moving objects are represented by rectangles. Frogs and moving objects are moved by manipulating the x and y coordinates. Speed of moving objects are manipulated by their moveSize attribute in the movingObj objects and the value used for the interval in the observable containing their ids. Moving objects are wrapped around the svg. For turtles, they are hidden and based on their diveRange to produce a diving effect. The crocodiles have an extra rectangle which represents the mouth which Frog cannot enter. Score is awarded if the frog makes a valid up move that does not collide with cars or enter the river.

High Level Overview of Design Decisions and Justification

Model-View-Controller (MVC) is used to run the game. User inputs and moving object ids are fed in as inputs to manipulate state that updates the view for users. This design has allowed for a single flow of data of the whole game making it easier to manipulate the game. New states are created if a moving object moves and when the user moves the frog. Using the states generated by these events the game is updated to reflect the changes. To achieve state management, a gameState object was created and an observable was used together with scan and reduceState to update the state. A pure class was used to represent the frog's attributes and methods. This has reduced code repetition because a lot of the game manipulation involves the frog and by making a class all the methods for frog can be directly applied on the Frog object by method chaining such as frog.move(). Moving objects were represented using objects and there are functions to manipulate them. To make the moving objects move, intervals were used to manipulate the x coordinate at intervals based on the moveSize of a moving object. The x value for moving objects were mod with svg width - moving object width to wrap it around the svg. This avoids the moving objects from going out of the svg. To increase difficulty every round, the moveSizes of moving objects are increased so that it moves more at every interval producing higher speed and game difficulty. For turtles, they are hidden and based on their diveRange to produce a diving effect. Crocodile mouths are represented with another white rectangle on the green rectangle for crocodiles and if the Frog collides with it, it dies.

How the State is Managed While Maintaining Purity

The state of the game is represented by an object called gameState. The game is started with a readonly object of gameState that represents the initial state of the game. To maintain purity while managing the state, the attributes in the state are not altered by any function or code applied to it. The gameState object is managed in the scan function in the observables using the reduceState function. This function is pure because it does not alter the gameState attributes and it applies functions to return a new gameState object. To manipulate the frog

attribute a new gameState is returned with a new Frog object in the frog attribute, purity is maintained by using the methods provided by the Frog class. The Frog class protects purity by not altering its attributes, if any alteration is needed based on user move the methods will make use of the attributes of the frog and return a new Frog object with different attributes. The cars, planks, crocs and turtles attributes have lists of moving objects. These attributes are not altered as well, instead a new gameState is created. If the state requires a change in distinctTargetArea attribute, a new gameState object is returned with distinctTargetArea containing a new list of distinctTargetArea objects that are created using functions that do not alter the list but return a new list. To update the score and highScore attributes, a new gameState object is returned. In this returned gameState object the score and highScore are calculated using methods that do not alter the gameState object given to the reduceState method. The restart and init attributes are not altered in the gameState object given to the reduceState method. Instead the complete method returns a new gameState object with different values for restart and init. Purity is protected because the gameState passed to the reduceState and all the methods it calls are pure. The reuduceState method and methods called by it do not alter the HTML to protect purity. All impure code contained in the subscribe function as all the functions manipulating HTML are called from the subscribe function.

How the Code Follows FRP style

To achieve small and granular functions, functions with long code were broken down into smaller functions and this function would call these smaller functions to achieve the same behaviour. For instance, the isInPlank function in Frog class applies the reduce function on the list given to it and in the reduce function it calls a small and granular function called isIn that checks if the Frog is in that moving object.

Reusable functions were created reducing repetitive code. For instance, to move a car, plank and crocodile across the screen the same code was required, function moveCarPlankCroc accepts car, plank and croc as input and calls another function to calculate their x attribute to return a new car, plank or croc object. This reduced code repetition because code for the same process was coded once as a function.

Every function except for the functions that manipulate HTML were coded with purity and referential transparency. Functions were made pure by ensuring they do not alter the input directly or any global constants and return a new value that is not an altered input value or return nothing. They are referentially transparent because the places where they are called can be replaced with a value without affecting functionality. For instance, seaNoFrogMoveState function is pure as it takes a gameState object as input and its return value is a new gameState object with some of its attributes having a copy of the attributes of the input gameState object and the others computed using functions that are pure and the moveDown function in Frog class reads the constant SVG_HEIGHT_WIDTH without altering its value.

Fluent interfaces and a fluent coding style was used. This was achieved by using declarative code and avoiding imperative code like loops. Frog and FrogMove classes do not have methods allowing to alter their value, their functions only return a new object of their class or

return another value without altering their attributes. For example, the moveLeft function in Frog class returns a new Frog object with a different x value instead of altering the x value or it returns a copy of the object and not the object itself.

Different complex types and generic types were used appropriately. For instance, every type of moving object has a corresponding object type such as car, plank, croc and turtle. By doing so specific functions could be coded for specific types for a certain logic or manipulation. For instance, the function moveTurtle that accepts turtle and moveNotTurtle that accept car, plank and croc, turtle need the hidden attribute to be changed when returning a new turtle object when turtle has to move on screen but car, plank and croc don't need it. By doing so these types can be manipulated with functions that cater to their type.

Higher order functions (HOF) were used throughout the code. For instance, to calculate the score when a frog moves in a river, the calculateScoreRiver function returns a function. For example, createMovingObjView function takes a function as an input to determine what colour to use when displaying a moving object in HTML. These are HOF because a HOF accepts as input or return a function

Next, Curried functions were used throughout the code. Curried functions are functions of several arguments rewritten into a function that takes one argument and returns another function to take the next argument until all arguments are taken then it does computation. For instance, in roadFrogMoveState function, calculateScoreRoad returned a function that takes in another input to compute the final return value, it was saved as currentScore const and called again with updateScore(s.score) in the next line.

Functions were composed and chained throughout the code to produce more readable code and reduce code length. For instance, the moveNotTurtles function takes in a list of car, plank or croc and chains a map function to it with a function passed into the map function and by doing so the function can return a new list with the moving object's x value updated. The Frog class objects use chaining to return new Frog objects based on the game state. For instance, riverFrogMoveState function the const frog is created with the input game state frog with move method chained to it to return a new Frog object that corresponds to user move. Functions were also composed throughout the code where a return value of a function is passed into another function. For instance, in the roadFrogMoveState function the score const has a value which is assigned by calling updateScore on s.score and calling currentScore on the returned value.

Usage of Observables Beyond Simple Input

A stream of the string "start" using of was created so that in the reduceState function if the string "start" is input, the state returned would be the initial state that is needed to start the game. Streams of user moves, move$ is created to manipulate state based on user movement. Streams of ids of moving objects were created using intervals with different times to make the moving objects move at different speeds. concatMap was used to shorten the code and multiple observables representing a list of moving object ids will be emitted at the same time interval by interval function so that moving objects with these ids will have the desired speed. These streams were merged together using the merge and scan function to call reduceState function on these values. delayWhen is used to make a 100ms delay when

a round is over to increase game smoothness when restarting. Return values of the scan function in pipe function is subscribed by the updateView function in the subscribe function. The returned gameState objects from scan are used by updateView to update the Frog and moving objects on the HTML.