Welcome to my pong game.

My program begins by running the pong() function after the window loads. On this function an observable stream is created using the interval() method. Each interval is 1ms for the smoothest game play.

The program's code uses a method of continuosly passing a state through different functions, changing it each time based on its current parameters and then returning a fresh State type with new values. Every function that the reduceState calls and that that calls and so on is 100% pure.
This means that no matter what the state of the system is, each function will return the exact same value for any input given to it, and a brand new object as well to avoid memory leaks and allow for future possibility of storing previous states (for use in an action replay etc). I have even gone a step further and curried methods (updateScorePure, moveObjPure, nextStatePure) as to pass them the parameters that are constant with the game to create new functions with these parameters in mind. What this allows for the ability to run unit tests on each function individually, knowing that no matter what the state of the system they will give the same answer. This also means that calling functions produces no side effects to the current game state. The functions are continuosly passing each other states, looking at what needs to change, and then handing off their new state to the next function. This also happens with Body types, where Body's are copied with new values based on the funcitons input parameters.

The subscription runs throughout the entire gameplay, this is to maintain pureness within the functions of the program by eliminating side effects, it is never unsubscribed from from within any method which maintains the codes purity. The asteroids example had updateView call to stop the subscription, which meant that the updateView changed both the state of the program as well as the DOM. My updateView function will only ever interact with the DOM and pausing and resetting the game is achieved through continously passing new states that are the exact replica of the state before hand, or passing in the level(0). This means that updateView will only ever have side effects with the DOM. A useful feature for splitting debugging the debugging of state calculation and DOM manipulation into two mostly independent tasks .

The game is interacted with through the keyboard, the observable streams define both the key types as well as the key events that should be listened for. The up and down arrow keydown events tell the program to calculate its next state with the direction of the arrow press in mind. The velocity of the paddle remains the same until the player releases the arrow, which is also counted as a MovePlayer action. Space will pause the game and the left and right arrow will increment levels (as I don't expect you to spend the time trying to get past each one)

The observable stream is filtered the keys for non repetitions, this is due to the key registering as continuously pressed after about half a second of pressing. Each press passes a class to the reduceState function through scan, which looks at the previous state and continuously updates what the next one will be. If no event is observed from the user, the

game will ‘tick’ which continuously passes the state through a series of calculations to determine what comes next.

Tick will first check if the game is paused or reset and if it is will laziliy return the current state (won’t calculate the next state) or initial state respecitively. If not paused the next state is calcualted by running filters, maps, reductions and and concatinations over the cons lists of bots and players with different methods to calculate the Scalable Vector Graphics. I have used cons lists as they are a great example of how church encoding can create fully functional programs. This also allows me to implement my own methods for the cons list such as ‘get’. In the future the game’s functional style can become tighter and tighter through adding methods with Generic types to the Cons list. Methods such as ‘pipe’ could come in handy to tighten up the code (less code). These methods can be applied to any list of new kinds of objects due to the use of generic types which greatly inreases the parametric polymorphism.

I have curried functions such as reboundVector and bodiesCollided to increase reusability of code. As there are not a lot of different types of objects in this game is doesn’t come in extremely handy, but it keeps the code expandable for the future.
I have used currying and higher order functions within the levels function, as you can see, I can define methods that are applicable to bodies such as chaseBall and reboundVector. I this saves a lot of repeated code. Creating new functions such as l3 (.3 extra speed level) from speedChange(x)(y) which again came from another function ‘change’, a function which takes as paremters a function, two number values and Body and then returns a new Body based on that function and two numbers. It’s a mouthful but it basically allows for cleaner mapping of functions which would have previously contained huge chains. You can see this used as well for the obsevervable methods defined in the program. There are also many implementations of higher order functions in cons as to allow for recursion over impoerative loops whilst still applying a funciton for each recursive call.

My UpdateScore function only handles the changes of each round, such as little encouragement messages which were originally going to be random, however that wouldn’t be pure as it would give different results each time (possibly in future could make a pseudorandom method based off of the state). The score, the level, and the round are calculated here.
It also generates the score/canvas message for the next or previous level.
updateScore uses multiple anonymous functions, as they are only used once within the program they are defined using arrow notation and not stored in a reference.

I have added an implementation for iterating through a HTMLCollection. This is due to converting it to an array would not maintain referential integrity. The forEachDOM method allows you to apply a function. I have also rewritten the code from asteroids that applies attriubute to use forEach instead of a for loop. As a for loop can result in errors due to incorrect initialisation, condition, final expression, and statement.

I have used type checking throughout my program as to not only make it easier for future programmers, but also make it easier to spot bugs.

For my extra feature I have implemented a few things. To start, I have a level-system. This means that players progress through the game and encounter different challenges each time. From multiple bot paddles to faster balls, fatter bot paddles, and so on.
Each level is calculated in updateScore, and the plauer can decrease a level if the bot wins from a best of 5 set. After finishing all 10 levels the player gets the message ‘you win’.

I have changed the number of winning rounds needed to 3 from 7. This progresses the player to the next level faster and can be changed within constants. I have a function which stores a cons list of levels. Each level a little bit different from the one previous. The levels function is one of my favourite as it demonstrates how higher order functions, cons lists, and currying can create sleek and easily readable code.

I have also implemented a ‘waiting screen’ which tells the user how to play and also has a slow and asteady battle between the paddles which can be taken over at any time by the user.

Although implementing levels may not seem like much, there is huge potential for expandibility with the way I have designed the code in terms of currying and using List’s for storing bodies. With more time there is room for developments such as Split Player paddles, angled objects to bounce off of, multiple balls, etc. I’ve designed my functions to be easily repeated with currying and adding levels is as easy as adding a new item to the cons list.

The game is also, in my opinion, quite aesthetically pleasing.

There is a much more in depth explanation within the comments of the code about some of the design choices I have made, each decision is explained line by line.