Quick summary on workings of code and learning outcomes

After reading the specs, I started with just bombarding ideas into the code, mainly used many filter () and subscribe () to get the wanted result. Much research made, however after consulting tutors and discussion in forum, although was trying to make use of observables while staying "pure", I began to understand what FRP really means. Did not really bother that much while reading the notes, however implementing the code is a different thing.

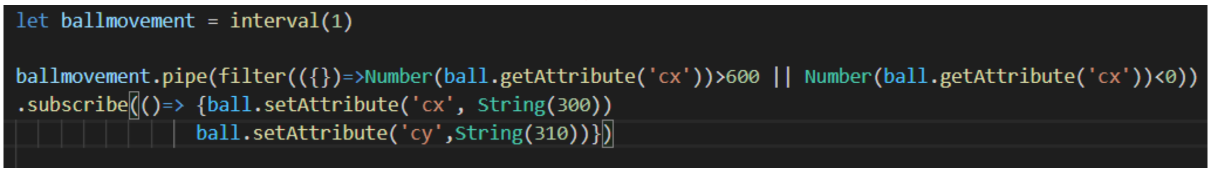
Process of how I understand FRP

I guess my main learning outcomes are:

1. pure functions will have the same output whenever the same specific input is given.
2. Functions that has any access to variables outside the scope and have possibility to change it, eg. The use of getAttribute / setAttribute outside subscribe (), will have side effects.

With these 2 main learning outcomes: I began to realise my old codes

Filter (({})=> xxx.getAttribute * r)



```
let ballmovement = interval(1)

ballmovement.pipe(filter(({})=>Number(ball.getAttribute('cx'))>600 || Number(ball.getAttribute('cx'))<0))
.subscribe(()=> {ball.setAttribute('cx', String(300))
                 ball.setAttribute('cy',String(310))})
```

- for example , is not pure at all!

3. Also, I realise FRP isn't all about having 0 side effects completely because that's impossible.

From the notes in Functional JavaScript written by Dr Tim, one of the side effect stated is outputting to console :

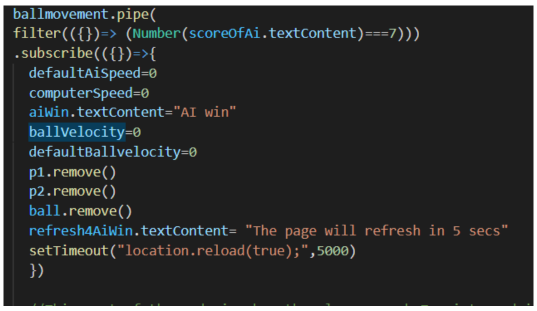
Console.log (smth)

By knowing this, I was confused → does this mean FRP don't allow outputting?

However, throughout the assignment, I realise that FRP's main goal **is** to **contain** the side effect and achieve **referential transparency** (which we learnt in lecture) and not just purely can't use console.log at all. Easier to understand this theory, through implementation of code.

By understanding this, I understood why setAttribute() can only be written within the subscribe function since that's the only output source. Using it elsewhere, will only introduce side effect and unable to maintain pure codes. Throughout the assignment, in attempt to achieve pure code, I also realise the benefit of FRP which can make the code much more bug free and also don't have to worry bout side effect much.

While I was using observable but in an impure way the code was messy and really confusing at times:



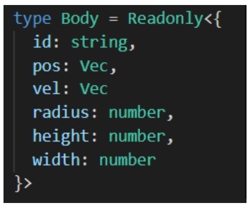
```
ballmovement.pipe(
filter(({})=> (Number(scoreOfAi.textContent)===7)))
.subscribe(({})=>{
  defaultAiSpeed=0
  computerSpeed=0
  aiWin.textContent="AI win"
  ballVelocity=0
  defaultBallvelocity=0
  p1.remove()
  p2.remove()
  ball.remove()
  refresh4AiWin.textContent= "The page will refresh in 5 secs"
  setTimeout("location.reload(true);",5000)
  })
```

A brief example (a small fraction) of what kind of messy code I was making.

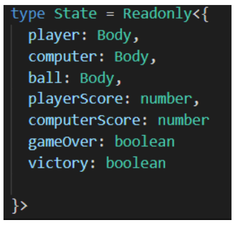
Resulting in many bugs and output issues as there were many mutable variables as well. (not declared readonly)

Overview of my design decision and justification

From the above summary, my old code is impure and to achieve pure functional code, I need State object which keep tracks on each state and update with a whole new state instance. To do this, I refer much to Dr Tim's asteroid code. With it being the perfect example of a pure Functional FRP style, I decided to go ahead with his design. First, I must understand the code, here is my understanding and how I used it for my game Pong:



```
type Body = Readonly<{
  id: string,
  pos: Vec,
  vel: Vec
  radius: number,
  height: number,
  width: number
}>
```



```
type State = Readonly<{
  player: Body,
  computer: Body,
  ball: Body,
  playerScore: number,
  computerScore: number
  gameOver: boolean
  victory: boolean

}>
```

It started with using **type** for stuff which can be categorise as one type (data type), most importantly are the **Body** and **State** type which I've also used in my implementation as well.

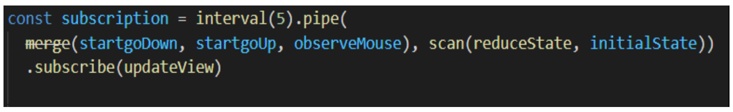
The Body keep track of the elements that we are manipulating while the State keep track of all those bodies in that state instance.

I accustomed it to use it in my own way of writing Pong game, I included the **Vector class** because it is also suitable for Pong, where position and velocity are concerned as well. I updated Vector class with more function that can be used for calculation but at the same time creating new Vec () each time so to ensure purity and FRP style in place.

These implementations are great for keeping code pure functional, as it is defined **Readonly** as well which will show an error (red line) to remind us whenever we are trying to change its property value.

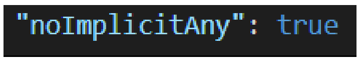
With this type implementation: my state throughout the game can be manage in this way, where I create function which accepts a state as parameter and since the state will be constant throughout the function, any reference to the state and its property will still remain pure since it will still obey the same output for same input (as state remain constant **inside** that function) as discussed in the forum : https://edstem.org/courses/4439/discussion/297210

Also, with the use of reduceState(), where we scan it with the initialState, we are able to return an output with desirable changes but also note that every return will be a deepcopy of the state and hence a "new State". We used **{…state }** as since State is Readonly → both of this ensures us to not ever have a "mutable variable" and also not mutate variable outside the scope which are impure implication.



```
const subscription = interval(5).pipe(
  merge(startgoDown, startgoUp, observeMouse), scan(reduceState, initialState))
  .subscribe(updateView)
```

Similar to Dr Tim's code, i then put all my setAttribute and getElementbyId in the **updateView** function which will then be pass into a **subscribe(updateview)**.
This ensures that whatever output we have are all inside the subscribe (keeping every risk of side effect inside a container) which then achieve FRP style of code.

Most of it are similar to asteroid code, however as we have 

```
"noImplicitAny": true
```

which means the attr () function to setAttribute conveniently cant be use as it involves object(can be any) hence I set them manually.

Extension and Game logic

I did a minor extension where players can choose their level of difficulties, with a more accurate AI and faster ball being the hardest level. With this implication, I need to reset the game when player want to choose a different level hence the extra feature of a refresh button and the need to use the setTimeOut

However, with that being said, I realise Observable is more abstract way (more pure) of a setTimeOut function hence I used another observable with the interval (5000) for the refreshing state. All this are implement inside the subscribe() to keep it pure.

The game logic I implemented with HTML and CSS is that player can choose either use mouse or up and down arrow keys to move the mouse for the left paddle, and :

1. They click on the level they want to play
2. Every time they score, their scoreboard above will increment.
3. Once hit 7, either side wins with a statement indicating the winner in the middle of the canvas.
4. Once game finish, the page will refresh in 5 seconds. Or the player can choose to refresh now with the refresh button.