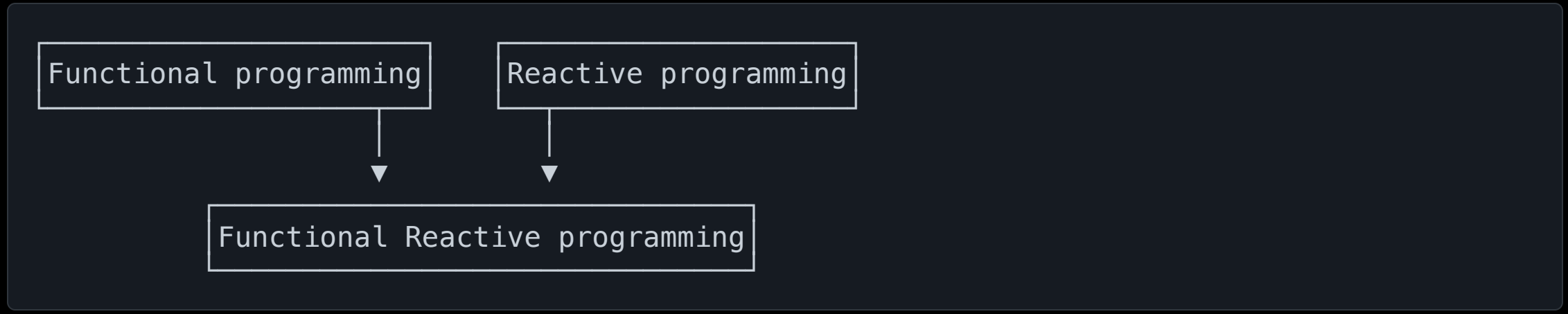
Functional Reactive Programming

> using RxJS (event though it is not purely frp 😬)

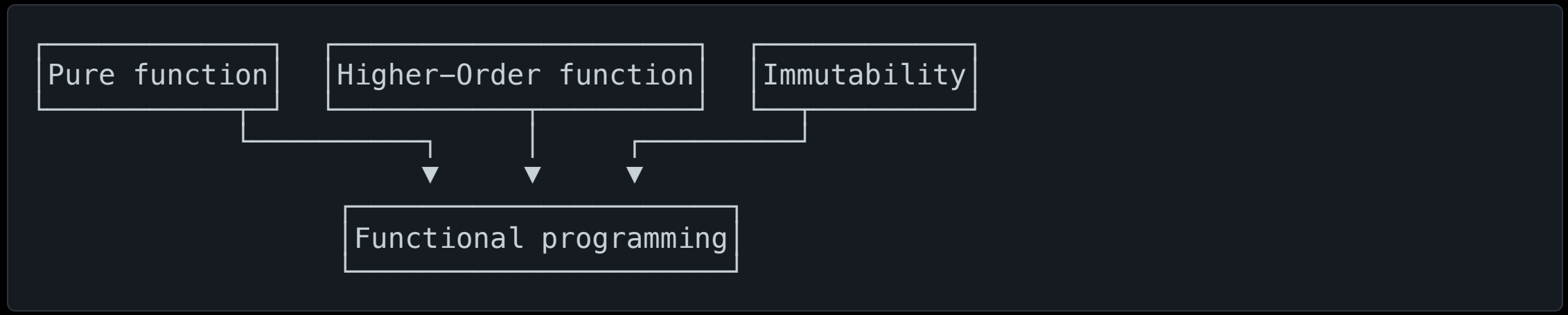
But let's begin with the problem

https://codesandbox.io/s/tripple-click-5p4t0z?file=/src/index.ts

Functional Reactive Programming



Functional Programming



Pure functions

> - Return values are identical for identical arguments
> - No side effects

```
const clamp = (min, max, num) => min(max, max(min, num))

// referential transparency
clamp(1, 10, 7) === min(10, max(1, 7))
clamp(1, 10, 7) === min(10, 7)
clamp(1, 10, 7) === 7
```

vs

```
const clamp = (min, max, num) => {
    console.log(min, max, num);

    return min(max, max(min, num))
}
```

Immutability

```
dice.value = 3

console.log(dice.value) // 5, because some timer callback just mutated it
```

vs

```
newDice = { ...dice, value: 3 }

console.log(newDice.value) // will always be 3

findDiff(dice, newDice) // because the old dice is still available
```

Higher order function

```
// currying
const sum = (a: number) => (b: number) => a + b;

const array = [1, 2, 3, 4];

const withTenAdded = array.map(sum(10)); // [11, 12, 13, 14]
```

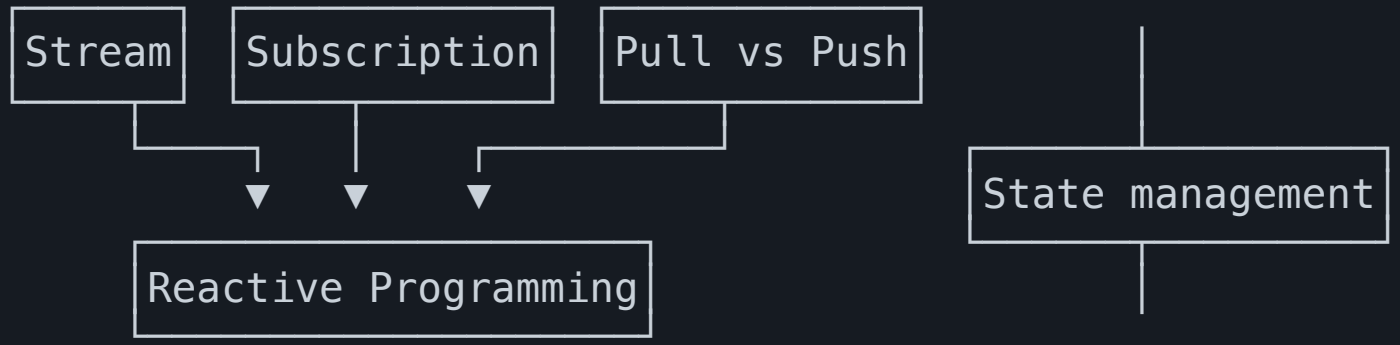
works like

```
const sum = (a: number, b: number) => a + b;

const array = [1, 2, 3, 4];

for (let i = 0; i < array.length; i++) {
    array[i] = sum(a, b);
}
```

Reactive programming




```
var b = 1
var c = 2
var a $= b + c
b = 10
console.log(a) // 12
```

Stream

> A sequence of data elements made available over time

Like a lazy array of events

```
  click   click error  complete
  |       |     |      |
--a-------b-----X------|->
```

in contrast to

```
a-|
b |--------|
c          |--------|
d                   |------
```

Where we always have a current state

Subscription

Represents the execution of a stream, is primarily useful for cancelling the execution.

```
// Only since now the stream starts running
const subscription = stream$.subscribe(renderData);

// The way to stop execution
subscription.unsubscribe();
```

Pull vs Push

In **Pull** system, the **Consumer determines when** it receives data from the data **Producer**. The Producer itself is unaware of when the data will be delivered to the Consumer.

In **Push** systems the **Producer determines when** to send data to the **Consumer**. The Consumer is unaware of when it will receive that data.

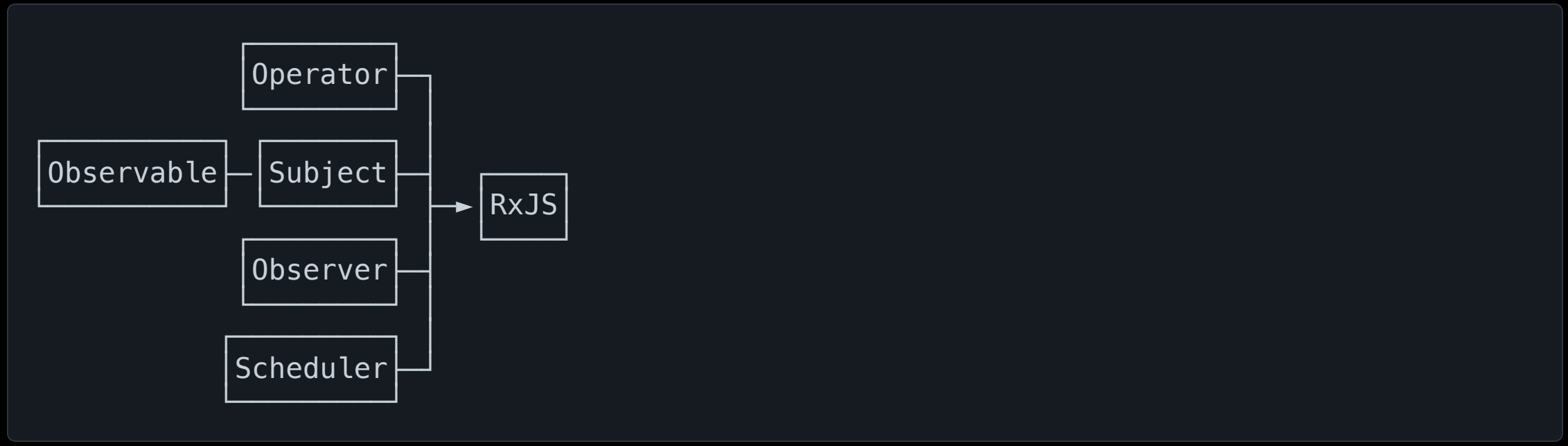
```
// Pull system
const pulledStep = getCurrentStep();
updateSomething(pulledStep);

// Push system
const counter$ = step$.onChange(step => updateSomething(step))

//  (Proactive)⟶   (Passive)
//  (Listanable)  ⟶(Reactive)
```

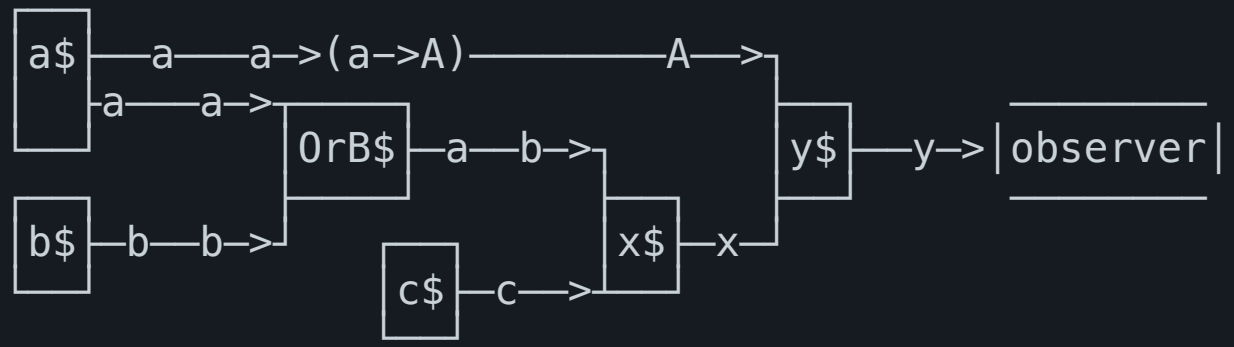
RxJS

> Reactive Extensions Library for JavaScript



Observable (stream)

> Observables are lazy Push collections of zero or more values




```
legend:
 a$    = observable
-a--   = event over time
(a->A) = modification of an observable
```

Operator

> Operators are the essential pieces that allow complex asynchronous code to be easily composed in a declarative manner.

Creation Operators

https://codesandbox.io/s/observable-tm0yfx?file=/src/index.ts

> Are the other kind of operator, which can be called as standalone functions to create a new Observable

```
const array$ = from([1, 32, 151, 2332, 12, 33]);

const everySecond$ = interval(1000);

const click$ = fromEvent(document, "click");

const fromPromise$ = from(makePromise(true));
```

Pipeable Operators

https://codesandbox.io/s/pipable-operators-xnxgt2

> Are the kind that can be piped to Observables using the syntax `a$.pipe(/* operators */)`

```
const complexLogix = pipe(
     map(modifyValue),
     scan(accumulateValues, initialValue),
);

interval(1000)
     .pipe(filter(isOdd))
     .pipe(complexLogic)
     .subscribe(console.log);
```

Join Creation Operators

https://codesandbox.io/s/join-creation-operators-3mg9vo?file=/src/index.ts

> Composition of operators

```
const timer1$ = interval(1000);
const timer2$ = interval(100);

const result = concat(timer1$, timer2$);
```

Subject

https://codesandbox.io/s/subject-skyx28?file=/src/index.ts

> A Subject is like an Observable, but can multicast to many Observers. Subjects are like EventEmitters: they maintain a registry of many listeners.

Regular observer looks like:

```
--a-------b-----c------d->
```

while subject is like:

```
a-|
b |--------|
c          |--------|
d                   |------>
```

where we always have a current state

Observer and Subscription

https://codesandbox.io/s/subscription-and-multicasting-q83jq5?file=/src/index.ts

> An **Observer** is a consumer of values delivered by an Observable

> A **Subscription** is an object that represents a disposable resource, usually the execution of an Observable

```
const observer = {
     next: nextValue => { ... }, /* Called on every next value */
     error: err => { ... }, /* Called if error will happen */
     complete: () => { ... }, /* Called on complete */
};

// Subscription = Observable + Observer
const subscription = observable$.subscribe(observer);

subscription.unsubscribe();
```

Scheduler

> A **Scheduler** controls when a subscription starts and when notifications are delivered. It consists of three components.

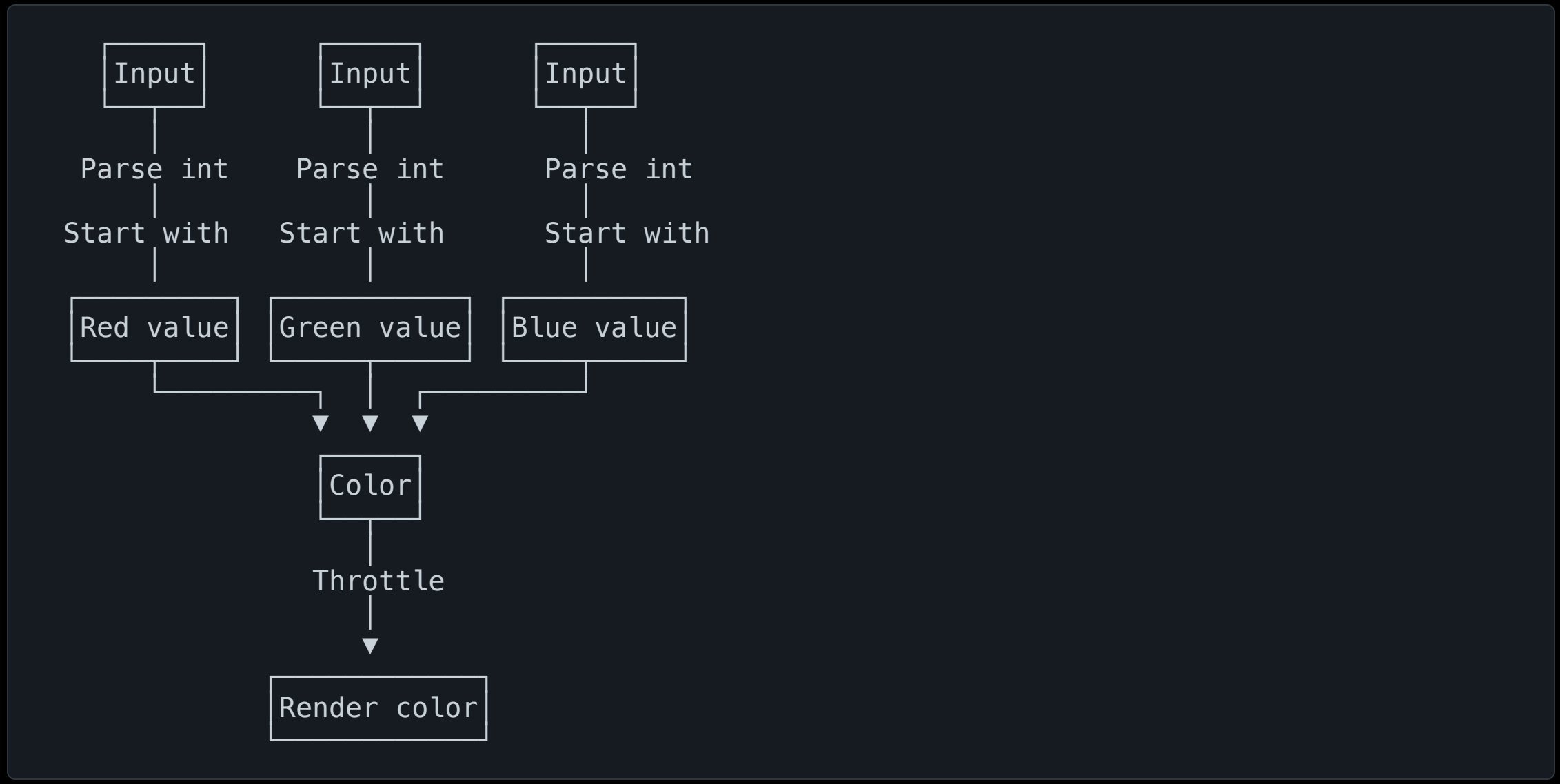
Random use cases:

- Make a synchronous steam an asynchronous
- Fit execution into `requestAnimationFrame`
- Fit execution into the micro task queue, which is the same queue used for promises

Let us play

https://codesandbox.io/s/color-picker-srf76n

Color picker



Way to go

Official RxJS docs

https://rxjs.dev

Alternative learning resource

https://www.learnrxjs.io

RxJS marbles

https://rxmarbles.com

Awesome FRP JS

https://github.com/stoeffel/awesome-frp-js

Task

> Make a timer

Minimum requirements

- If the timer is not running, a **Start** button should start the timer
- If the timer is running, a **Stop** button should stop the timer
- Every time you start the timer, it should count from 00:00

Base requirements

- While the timer is running, a **Lap** button should add the time to the list
- Each time you start the timer, the list should be cleared