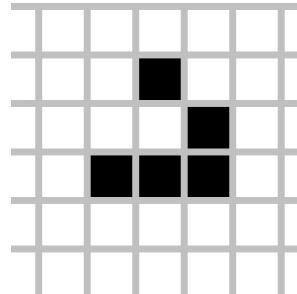
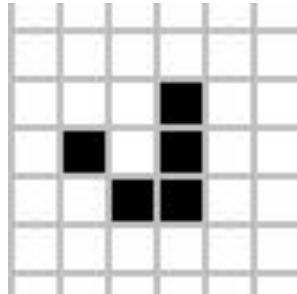


Functional Reactive Programming

CSC302H1, Winter 2018

Conway's Game of Life

- Grid of “alive” and “dead” cells
- In each iteration of the game (a “tick”), cells become dead or alive based on the previous state of the grid:
 - [underpopulation] Any live cell with <2 live neighbours dies.
 - [overpopulation] Any live cell with >3 live neighbours dies.
 - [reproduction] Any dead cell with exactly three live neighbours becomes a live cell.



Implementing the Game of Life

We need:

- an **initial state** of the world
- **game logic** (based on the pre-defined rules)
- a **timer** to produce “ticks”

We also want:

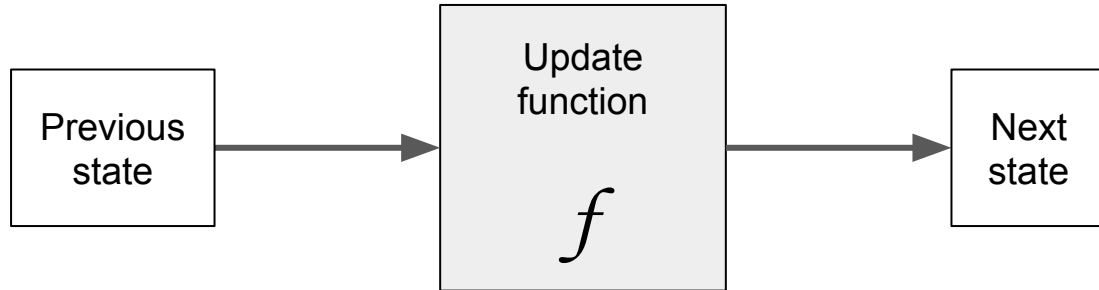
- a **visual representation** of the world, which gets **updated** when the state of the world changes
- a way to **interact** with the world (change the state of cells)
- a way to **pause/unpause** the game

} I/O

Implementing the Game of Life: **Game logic**

We would like to implement the game logic as a **function**, which takes a **previous game state** and produces the **next game state**.

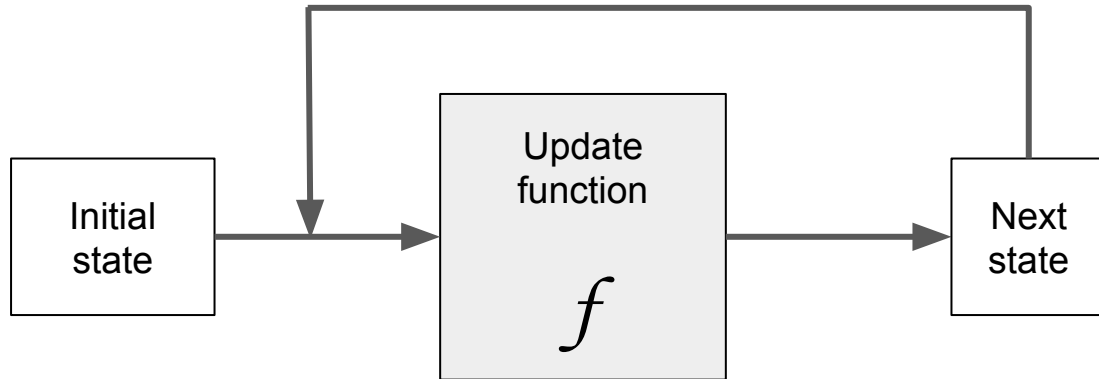
Moreover, we would like to treat the function as a **black box**: it doesn't matter how it is implemented; we can trust that it will correctly produce the next state if we give it the previous state.



Implementing the Game of Life: **Game logic**

This means that we can produce new states forever if we

- start with the initial state, then
- keep feeding the next state back into the function



Functional Game Logic

What's functional about this model?

It has no **side effects** (interaction with anything outside the model):

- To produce the next state, the game logic doesn't use anything other than the input it's given (the previous state)
- The game logic doesn't *modify* a global state (or any other external data); it produces a **new** version of the state

no side effects ↔ *stateless* ↔ *pure function*

What is the advantage of having a functional model?

→ **Testability**

What's the problem with this code?

```
function crazyDouble(x) {  
    if (new Date.getDay() == 4)    // if today is Wednesday  
        return x*3;  
    else  
        return x*2;  
}
```

We'd like to apply crazyDouble to all the elements of a list...

```
var doubled = []  
for (var elem of [1, 2, 3])  
    doubled.append(crazyDouble(elem))
```

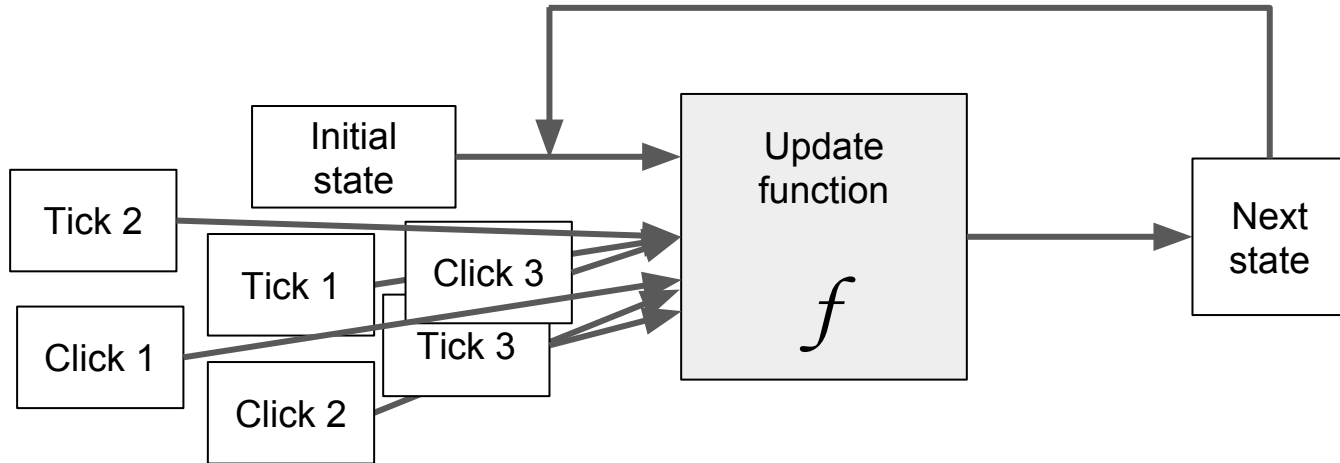
How do we test `crazyDouble`?

- Side effects make testing much harder - why?
- Are there cases where we need side effects?
(Pure functional languages don't allow them...)
- Notice that the loop order matters here:
If it is Wednesday when we start looping, and it stops being Wednesday *while* we are looping, then some elements will be doubled and some will be tripled - and the result would be different if we looped in a different order!
- So we can't apply `crazyDouble` to the elements in parallel, or in a different order, for optimization...

Implementing the Game of Life: Triggering updates

We want to trigger an update (i.e. call our update function)...

- on each tick
- on each click
- (maybe even when other things occur, if we add other functionality?)



Implementing the Game of Life: **Triggering updates**

Can we treat clicks and ticks as **events**, and implement **event handlers**?

→ What are the challenges of this implementation?

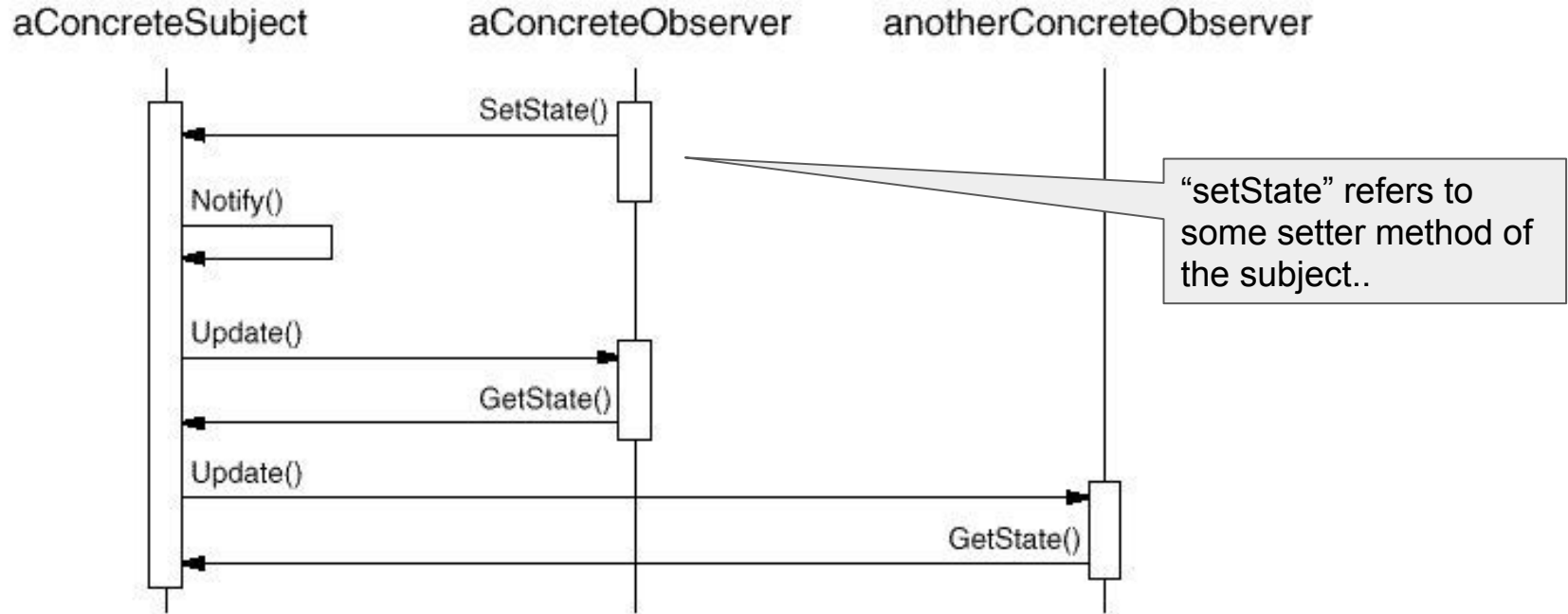
→ How would we test this implementation?

Can we create an **observable** object that holds clicks and ticks, **subscribe** to it, and trigger an update when a click or tick occurs?

(What is the difference between event handling and observer/observable?)

Observer/Observable Design Pattern (301 Review)

- Common design pattern
- Appeared in GoF
- A few names:
 - Observer-Observable
 - Listener
 - Publish-Subscribe
- When something happens to object A, object B gets notified and takes an action
- The two objects care about interfaces (eg. observer and observable), not concrete implementations



Observer/Observable - Why?

- **Simple way to decouple modules**
 - An observable doesn't need to know much about its observers
 - As long as the observers implement the observer interface (which is usually very simple), they will get notified whenever something interesting happens
- **Fundamental building block in event-driven architecture**
 - eg. GUI where a user's mouse click raises an event, which triggers various listeners
 - This is a standard way of decoupling GUI (presentation layer) from business logic

What is reactive programming?

[Wikipedia:](#)

“...a declarative programming paradigm concerned with data streams and the propagation of change”

[The Reactive Manifesto:](#)

“Reactive Systems are:

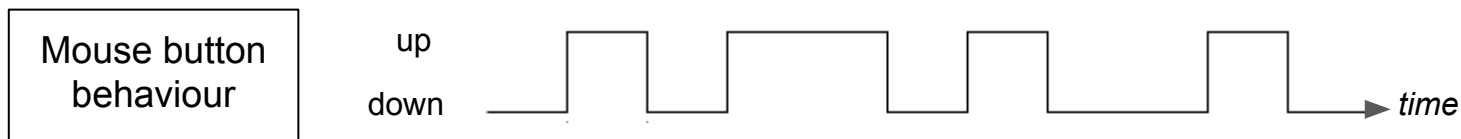
- Responsive: The system responds in a timely manner if at all possible.
- Resilient: The system stays responsive in the face of failure.
- Elastic: The system stays responsive under varying workload.
- Message Driven: Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation and location transparency.”

[Introduction to Reactive Programming:](#)

“Reactive programming is programming with asynchronous data streams.”

Modeling data as a stream

At the core of any system are **values**, which exist for some continuous period (eg. variables, pixels, mouse position). We'll refer to these as “**behaviours**”.



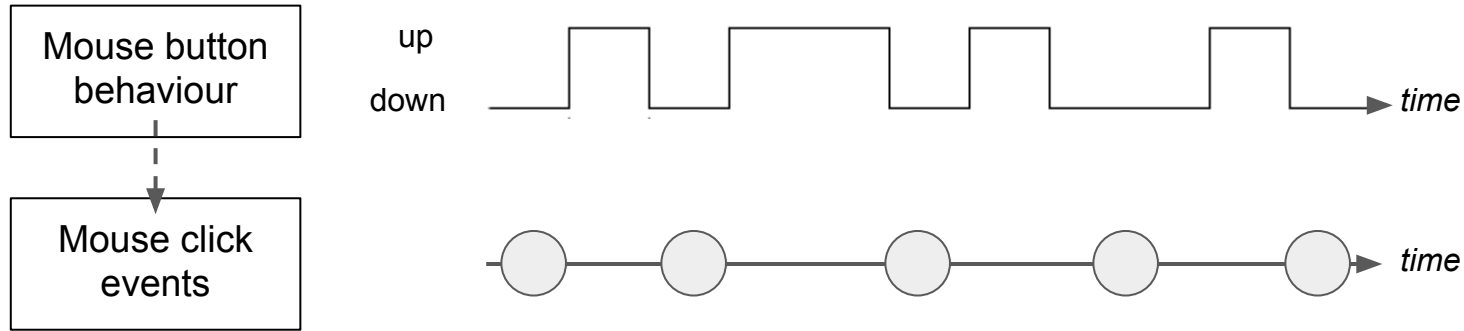
But computers don't operate continuously (in real time).

And we're not necessarily interested in behaviours at every point in time (eg. do we always care when the user moves the mouse?).

So we want to examine certain behaviours, in a way that doesn't depend on time.

Modeling data as a stream

We can describe specific conditions on a behaviour, which we'll call “**events**”.

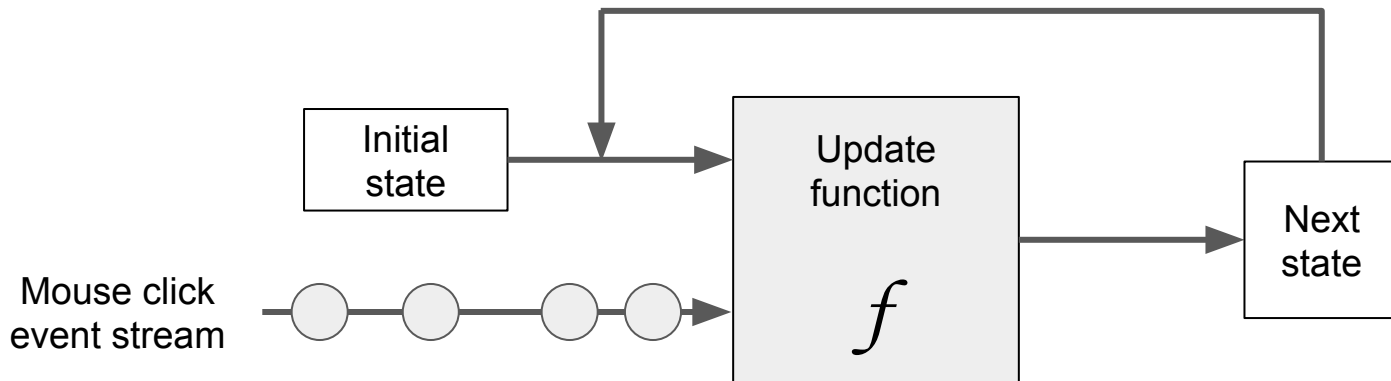


Events are discrete (they only occur at specific points in time).

We need a way to model events and have our system handle them...

Modeling data as a stream

If we can connect our system to an **input stream**, then we can **add events** to the stream as they occur, and our system can **react** to them as it receives them.



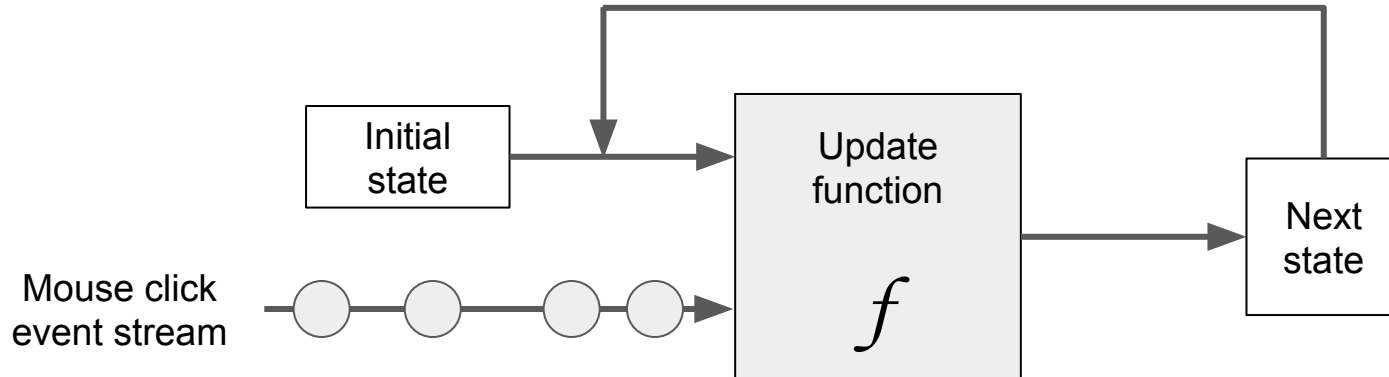
(Notice that our system still has no side effects!)

Modeling data as a stream

We don't model time, because the timing of an event doesn't matter to our system!

We are only interested in:

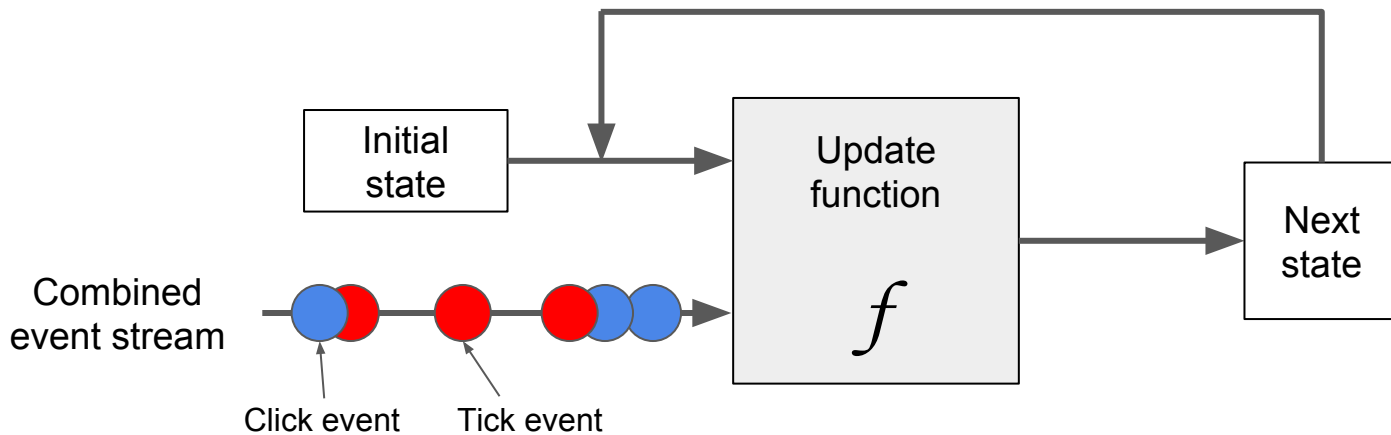
- the relative **order** of events, and
- the **state** of the system when the event occurs (which itself is only dependent on the initial state and the previous events that have occurred)



Modeling data as a stream

What if we have multiple kinds of events (eg. clicks and ticks)?

We can create streams of each of these events - and then we can **combine** them (or **create** new kinds of events) using functional patterns.



Can we make our **output** functional too?

Let's use React.js to create our output visualization...

Aside: A Quick Intro to React

- React is a JavaScript library that allows us to create HTML (and insert it into the DOM) using JS
- Part of the motivation for React was that the manipulation of DOM nodes is very costly
- React maintains a “shadow DOM” which is much cheaper to manipulate, and only makes changes to the webpage DOM when needed
- React *looks* a lot like HTML, but it can contain JS code that *produces* HTML
- Also, unlike HTML tags, React components can have custom properties and state

Can we make our **output** functional too?

React components can be functions, with properties as input and HTML as output:

```
const Grid = ({ world }) => (  
  <table>  
    <tbody>  
      {world.map(row => (  
        <tr>  
          {row.map(cell => (  
            <td style={{ background: cell ? 'black' : 'white' }} />  
          ))}  
        </tr>  
      )}  
    </tbody>  
  </table>  
);
```

Grid is a function that takes a 2D array `world` and returns an HTML `<table>`

Rows from `world` are mapped to table rows `<tr>`

Cells in each row are mapped to table cells `<td>`

If the cell is “true” (alive), the corresponding `<td>` is coloured black; otherwise, it is coloured white

Can we make our **output** functional too?

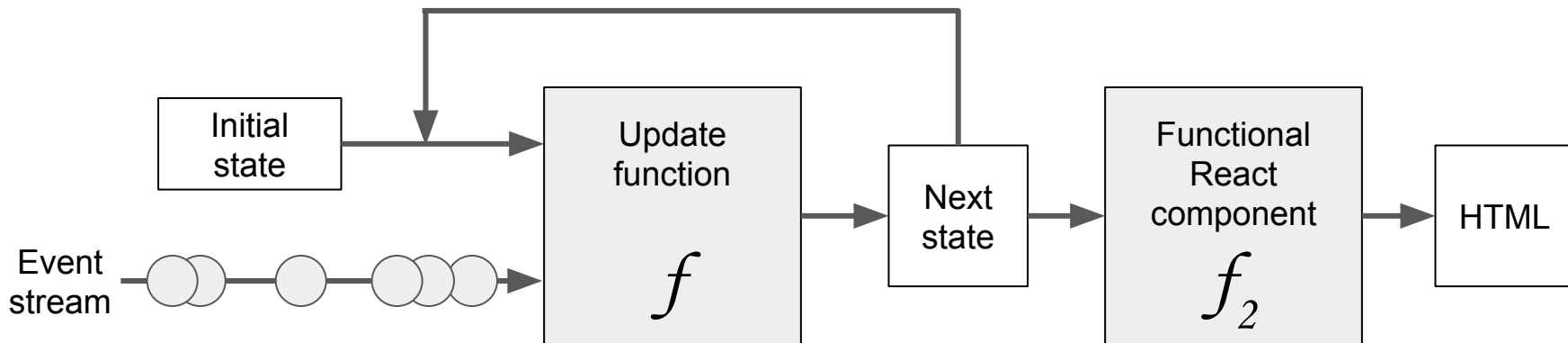
For `exampleWorld = [[false, true], [true, false]]`

we expect `<Grid world={exampleWorld} />` to produce:

```
<table>
  <tbody>
    <tr>
      <td style='background: white' /> <td style='background: black' />
    </tr>
    <tr>
      <td style='background: black' /> <td style='background: white' />
    </tr>
  </tbody>
</table>
```

Can we make our **output** functional too?

- This gives us the same **testability** as making our game logic functional
- We can easily create test input (a world array), for which we know exactly what output (in HTML) to expect



Commonly-used FP patterns/functions

map

`[a, b, c, ...].map(f) → [f(a), f(b), f(c), ...]`

```
>> [1, 2, 3].map(function (x) { return x*2; })  
[2, 4, 6]
```

filter

`[a, b, c, ...].filter(f) → [x where f(x) is true]`

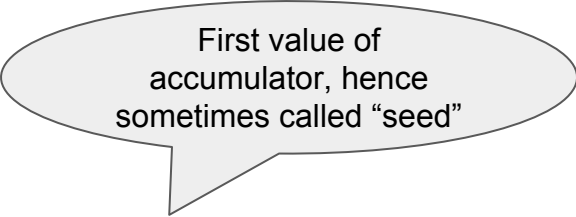
```
>> [1, 2, 3, 4].filter(function even(x) { return x % 2 == 0; })  
[2, 4]
```


Commonly-used FP patterns/functions

reduce

`[a, b, ..., z].reduce(f, acc) → f(z, f(... f(b, f(a, acc))))`

```
>> [1, 2, 3].reduce(function sum(x, y) { return x + y; }, 0)
// sum(3, sum(2, sum(1, 0)))
// sum(3, sum(2, 1))
// sum(3, 3)
6
```



First value of
accumulator, hence
sometimes called “seed”

```
>> [2, 3, 4].reduce(function prod(x, y) { return x * y; }, 1)
// prod(4, prod(3, prod(2, 1)))
// prod(4, prod(3, 2))
// prod(4, 6)
24
```

Commonly-used FP patterns/functions

lambda function

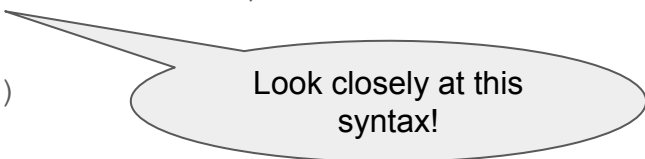
- Anonymous function (not bound to a name), created at runtime
- Useful for functions that will only be referred to once, eg. as a function input

```
(param1, param2, ...) => returnExpr  
(param1, param2, ...) => { funcBody...  
                           return expr; }
```

JavaScript

```
>> var add3 = (a, b, c) => a + b + c;
```

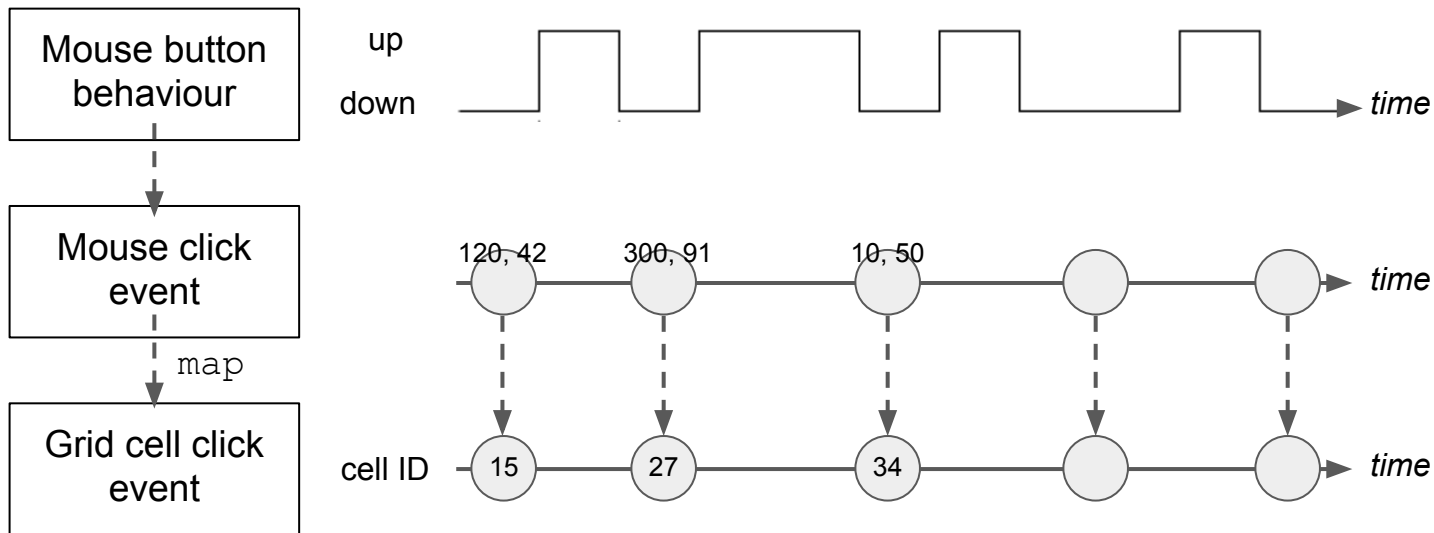
```
>> [1, 2, 3].map(x => x*2)  
[2, 4, 6]
```



Look closely at this
syntax!

Using FP patterns with event streams

Recall that behaviour can be abstracted as streams of events. We can use FP patterns to turn these event streams into new streams of new events.



This is a DOM event object, containing (x, y) position information we want...



Bacon.js

Bacon.js is an FRP library for Javascript.

Using Bacon, we can create a stream from DOM click events:

```
Bacon.fromEvent(root, 'click')
```

Then, we can use `map` to produce a stream of function *closures* containing information about the location of the click on the grid:

```
Bacon.fromEvent(root, 'click')  
  .map(event =>
```

```
    world => myToggleCell(world,  
                          event.target.x,  
                          event.target.y)
```

```
);
```

This function closure takes a world and produces a new world, with the state of cell (x,y) toggled

One more FP pattern: scan

Recall that `reduce` accumulates a list of values into a single, final value by repeatedly applying a function.

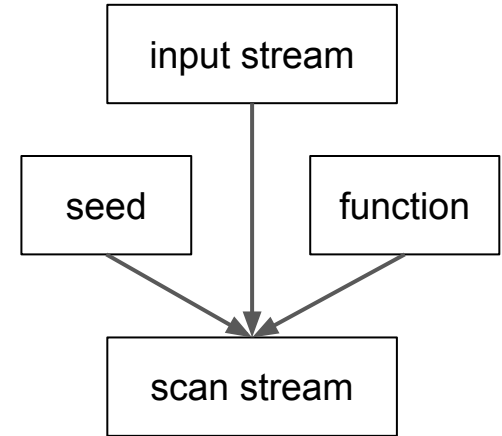
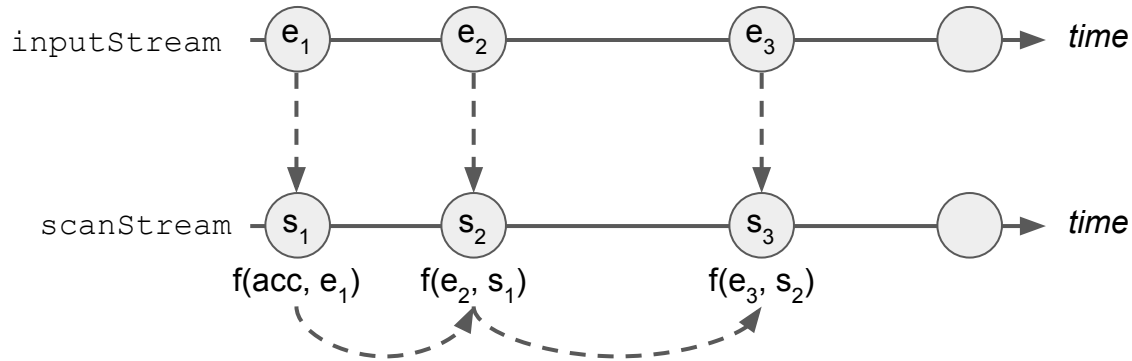
`scan` does the same, but it outputs each intermediate accumulated value as well as the final value.

```
[a, b, ..., z].scan(acc, f)
→ [f(a, acc), f(b, f(a, acc)), ..., f(z, f(... f(a, acc)))]
```

```
>> [1, 2, 3].scan(0, function sum(x, y) { return x + y; })
// [sum(1, 0), sum(2, sum(1, 0)), sum(3, sum(2, sum(1, 0)))]
// [1, sum(2, 1), sum(3, sum(2, 1))]
// [1, 3, sum(3, 3)]
[1, 3, 6]
```

scan with event streams

`inputStream.scan(f, acc) →`





Bacon/scan

Using Bacon, we can create a stream from pause button click events:

```
Bacon.fromEvent(pauseButton, 'click')
```

Then, we can scan the click events to produce a stream of booleans that indicates whether our game is currently “active” (whether ticks are occurring):

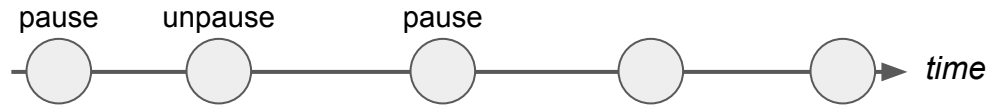
```
Bacon.fromEvent(pauseButton, 'click')  
  .scan(true, (prevState, _) => !prevState);
```

Each time the pause button is clicked, we change from “active” to “inactive” or vice-versa.

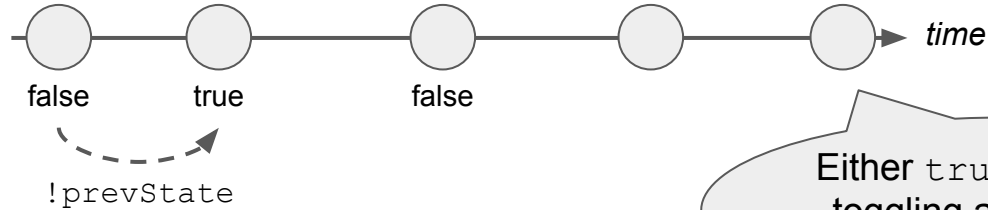
We start out active, so the scan seed is true.

scan: Pause button clicks become “active” events

Pause button
click event



activeStream

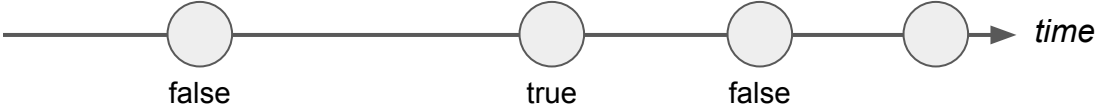


Either true or false,
toggling accumulator
from preceding event

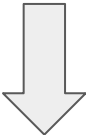
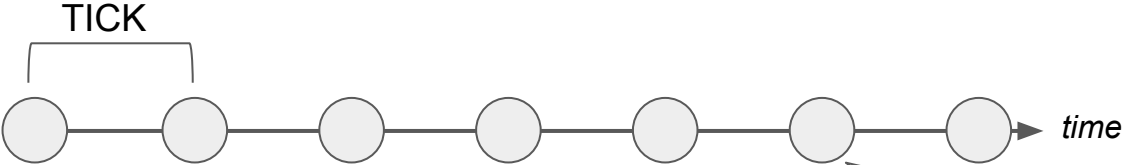
Now, we can use this to filter other streams, to remove events that should *only* occur when the game is active.

activeStream applied to ticks using filter()

activeStream



tickStream



Active ticks
(game not paused)



Ticks filtered out when the activeStream value is false.

```
ReactDOM.render(<Grid world={initialWorld} ... />, root);

let clickStream = Bacon.fromEvent(root, 'click').map(event =>
  world => myToggleCell(world, event.target.x, event.target.y)
);

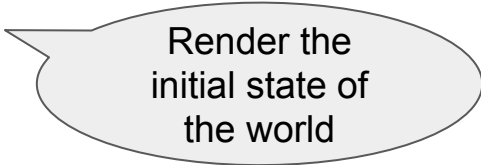
let activeStream = Bacon.fromEvent(pauseButton, 'click').scan(
  true,
  (prevState, _) => !prevState
);

let tickStream = Bacon.interval(TICK, myUpdateWorld);
tickStream = tickStream.filter(activeStream);

let eventStream = Bacon.mergeAll(clickStream, tickStream);

eventStream
  .scan(initialWorld, (oldWorld, updateWorldFunc) => updateWorldFunc(oldWorld))
  .onValue(world => ReactDOM.render(<Grid world={world} ... />, root));
```

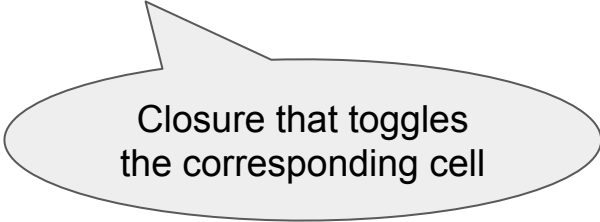
```
ReactDOM.render(<Grid world={initialWorld} ... />, root);
```



**Render the
initial state of
the world**

```
ReactDOM.render(<Grid world={initialWorld} ... />, root);

let clickStream = Bacon.fromEvent(root, 'click').map(event =>
  world => myToggleCell(world, event.target.x, event.target.y)
);
```



Closure that toggles
the corresponding cell

```
ReactDOM.render(<Grid world={initialWorld} ... />, root);
```

```
let clickStream = Bacon.fromEvent(root, 'click').map(
  world => myToggleCell(world, event.target.x, event.target.y)
);
```

```
let activeStream = Bacon.fromEvent(pauseButton, 'click').scan(
  true,
  (prevState, _) => !prevState
);
```

```
let tickStream = Bacon.interval(TICK, myUpdateWorld);
tickStream = tickStream.filter(activeStream);
```

**What is activeStream?
What does
filter(activeStream)
create?**

**Closure that
produces the next
state of the world**

```
ReactDOM.render(<Grid world={initialWorld} ... />, root);

let clickStream = Bacon.fromEvent(root, 'click').map(event =>
  world => myToggleCell(world, event.target.x, event.target.y)
);

let activeStream = Bacon.fromEvent(pauseButton, 'click').scan(
  true,
  (prevState, _) => !prevState
);

let tickStream = Bacon.interval(TICK, myUpdateWorld);
tickStream = tickStream.filter(activeStream);

let eventStream = Bacon.mergeAll(clickStream, tickStream);

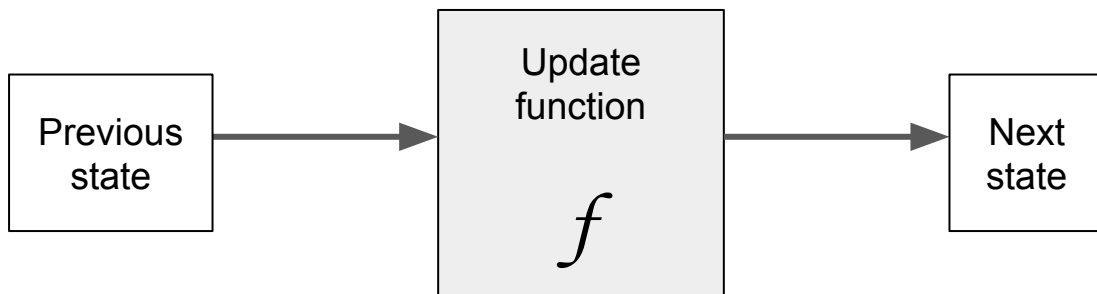
eventStream
  .scan(initialWorld, (oldWorld, updateWorldFunc) => updateWorldFunc(oldWorld))
  .onValue(world => ReactDOM.render(<Grid world={world} ... />, root));
```



What does this do?

Notice that:

1. We have 2 types of events that trigger a change to our world: clicks and ticks.
2. We map each of those events to a function closure:
Click: `world => myToggleCell(world, event.target.x, event.target.y)`
Tick: `myUpdateWorld`
3. The function closures have the same form, so that we can pass them to `scan`, which applies them cumulatively.



(Look familiar?)

```
ReactDOM.render(<Grid world={initialWorld} ... />, root);
```

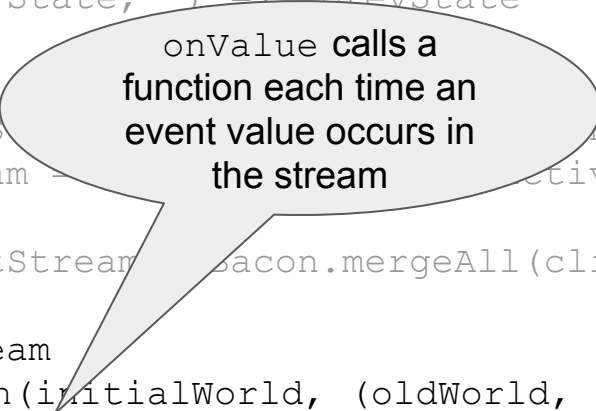
```
let clickStream = Bacon.fromEvent(root, 'click').map(event =>  
  world => myToggleCell(world, event.target.x, event.target.y)  
);
```

```
let activeStream = Bacon.fromEvent(pauseButton, 'click').scan(  
  true,  
  (prevState, ) => !prevState  
);
```

```
let tickStream = Bacon.fromEvent(K, myUpdateWorld);  
tickStream = Bacon.mergeAll(activeStream);
```

```
let eventStream = Bacon.mergeAll(clickStream, tickStream);
```

```
eventStream  
  .scan(initialWorld, (oldWorld, updateWorldFunc) => updateWorldFunc(oldWorld))  
  .onValue(world => ReactDOM.render(<Grid world={world} ... />, root));
```



**onValue calls a
function each time an
event value occurs in
the stream**


```
ReactDOM.render(<Grid world={initialWorld} ... />, root);

let clickStream = Bacon.fromEvent(root, 'click').map(event =>
  world => myToggleCell(world, event.target.x, event.target.y)
);

let activeStream = Bacon.fromEvent(pauseButton, 'click').scan(
  true,
  (prevState, _) => !prevState
);

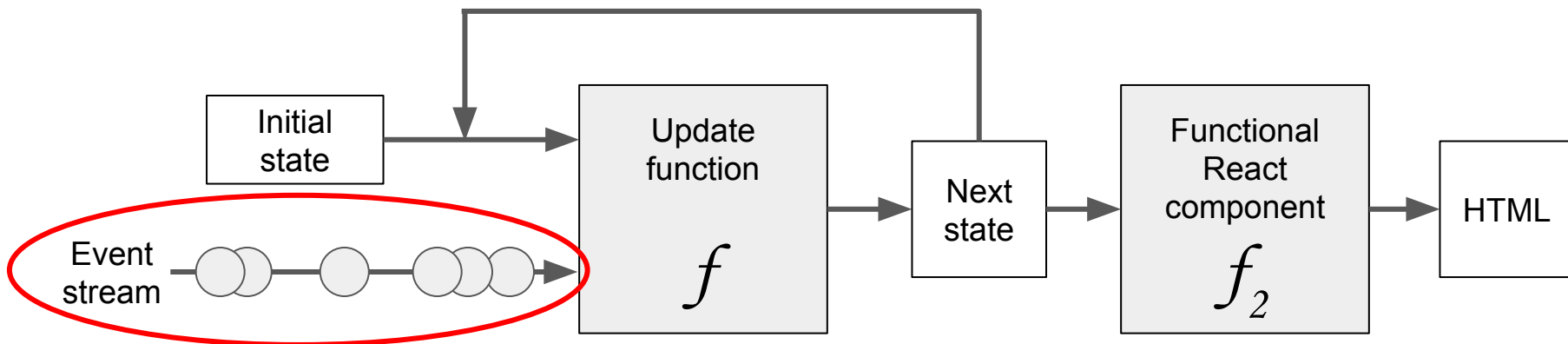
let tickStream = Bacon.interval(TICK, myUpdateWorld);
tickStream = tickStream.filter(activeStream);

let eventStream = Bacon.mergeAll(clickStream, tickStream);

eventStream
  .scan(initialWorld, (oldWorld, updateWorldFunc) => updateWorldFunc(oldWorld))
  .onValue(world => ReactDOM.render(<Grid world={world} ... />, root));
```

Testing our implementation

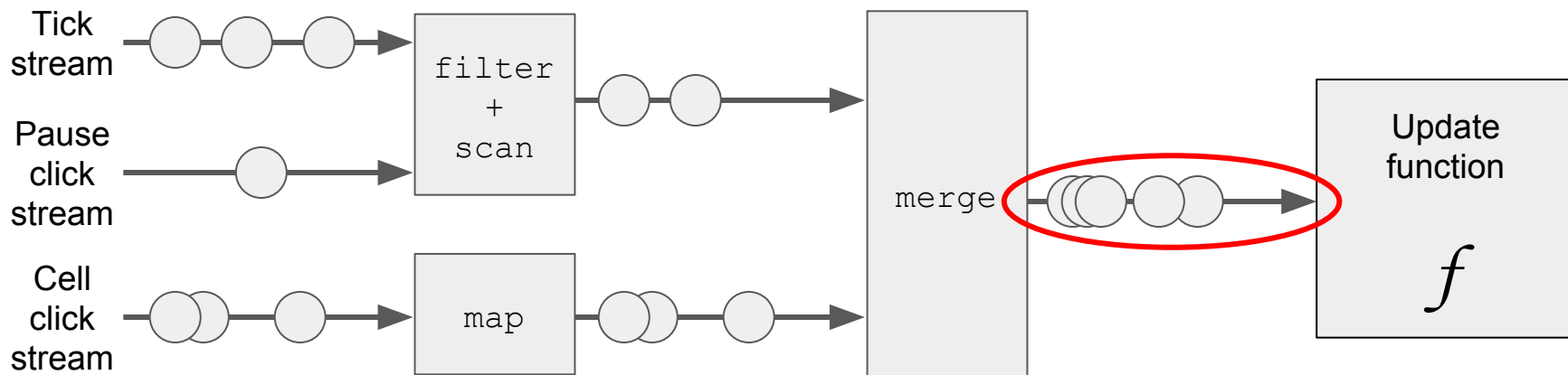
For testing our implementation, we'd like to create fake event streams:



Remember: our update function has no side effects, so if we give it a single event stream of ticks, clicks, etc. as input (and an initial state), we know exactly what output to expect.

Testing our implementation

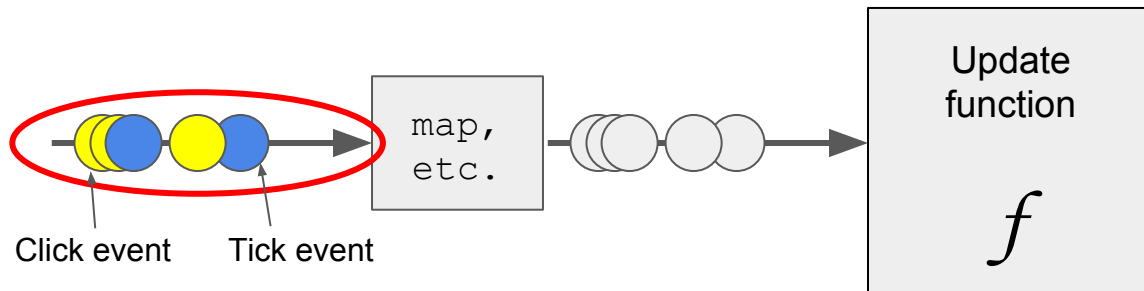
However, our implementation produces a combined input event stream by processing separate event streams:



We'd like to mock the combined event stream and *then* apply processing like `map`.

Testing our implementation

We'd like a structure like this:



so that we can easily control the relative ordering of different kinds of events.

Let's re-write some of our code...

Testing our implementation

First, let's put the "update function" into a separate function, which takes an initial state and an event stream:

```
function main(initialWorld, eventStream) {
  eventStream
    .scan(initialWorld, (oldWorld, updateWorldFunc) =>
      updateWorldFunc(oldWorld)
    )
    .onValue(world => ReactDOM.render(<Grid world={world} ... />, root));
}
```

Testing our implementation

We'd like `eventStream` to be a stream of combined events, but we still need to map those events to update functions.

Let's assume that each of our events contains a property called `myType`, which is a predefined constant. We can rely on `myType` to tell us which update function we should map the event to.

```
function main(initialWorld, eventStream) {
  eventStream.map(event => {
    switch (event.myType) {
      case TICK_EVENT:
        return myUpdateWorld;
      case CELL_CLICK_EVENT:
        return world =>
          myToggleCell(world, event.target.x, event.target.y);
      case PAUSE_CLICK_EVENT:
        return ...;
    }) ...
}
```

Testing our implementation

What should we map `PAUSE_CLICK_EVENTS` to?

→ `PAUSE_CLICK_EVENTS` are used for filtering `TICK_EVENTS`

→ They don't correspond to updates...

```
function main(initialWorld, eventStream) {
  eventStream
    .scan({ active: true }, (prevEvent, event) =>
      Object.assign(event, {
        active: event.myType == PAUSE_CLICK_EVENT ?
          !prevEvent.active : prevEvent.active
      }))
    .filter(event =>
      event.myType == CELL_CLICK_EVENT ||
      (event.myType == TICK_EVENT && event.active)
    ) ...
}
```

```

function main(initialWorld, eventStream) {
  ReactDOM.render(<Grid world={initialWorld} ... />, root);

  eventStream
    .scan({ active: true }, (prevEvent, event) =>
      Object.assign(event, {
        active: event.myType == PAUSE_CLICK_EVENT ?
          !prevEvent.active : prevEvent.active
      }))
    .filter(event => event.myType == CELL_CLICK_EVENT ||
      (event.myType == TICK_EVENT && event.active))
    .map(event => {
      switch (event.myType) {
        case TICK_EVENT:
          return myUpdateWorld;
        case CELL_CLICK_EVENT:
          return world =>
            myToggleCell(world, event.target.x, event.target.y);
      }})
    .scan(initialWorld, (oldWorld, updateWorldFunc) =>
      updateWorldFunc(oldWorld))
    .onValue(world => ReactDOM.render(<Grid world={world} ... />, root));
}

```


Creating a “real” event stream

```
let clickStream = Bacon.fromEvent(root, 'click').map(event =>
  Object.assign(event, { myType: CELL_CLICK_EVENT }));

let activeStream = Bacon.fromEvent(pauseButton, 'click').map(event =>
  Object.assign(event, { myType: PAUSE_CLICK_EVENT }));

let tickStream = Bacon.interval(TICK, { myType: TICK_EVENT });

let eventStream = Bacon.mergeAll(clickStream, activeStream, tickStream);

main(..., eventStream);
```

Creating a “fake” event stream

```
/* create an array of 10 fake tick events */
let eventArray = Array(10).fill({ myType: TICK_EVENT });

/* insert some fake pause events between the ticks */
eventArray.splice(5, { myType: PAUSE_CLICK_EVENT });
eventArray.splice(1, { myType: PAUSE_CLICK_EVENT });

/* insert some fake click events between the other events */
eventArray.splice(1, { myType: CELL_CLICK_EVENT, target: { x: 10, y: 1 } });
eventArray.splice(4, { myType: CELL_CLICK_EVENT, target: { x: 1, y: 13 } });
eventArray.splice(8, { myType: CELL_CLICK_EVENT, target: { x: 40, y: 9 } });

let eventStream = Bacon.fromArray(eventArray);

main(..., eventStream);
```

What if we don't use FRP?

Instead of a stream of events...

```
let activeStream = Bacon.fromEvent(pauseButton, 'click').scan(  
  true,  
  (prevState, _) => !prevState  
);
```

let's store a program **state** (could be stored in a state container) and create an event listener to update it:

```
let activeState = true;  
pauseButton.addEventListener('click', () => (activeState = !activeState));
```

What if we don't use FRP?

Again, instead of a stream of events...

```
let clickStream = Bacon.fromEvent(root, 'click').map(event =>
  world => myToggleCell(world, event.target.x, event.target.y)
);
```

let's store a program **state** and create an event listener to update it:

```
root.addEventListener('click', event => {
  worldState = myToggleCell(worldState, event.target.x, event.target.y);
  ReactDOM.render(<Grid world={worldState} ... />, root);
});
```

What if we don't use FRP?

Instead of a stream of (regular) tick events containing an update function closure, filtered by a pause event stream...

```
let tickStream = Bacon.interval(TICK, myUpdateWorld);
tickStream = tickStream.filter(activeStream);
```

let's call an update function at a **regular interval**, which checks whether the **state is active**, and **updates the state**:

```
setInterval(() => {
  if (activeState) {
    worldState = myUpdateWorld(worldState);
    ReactDOM.render(<Grid world={worldState} ... />, root);
  }
}, TICK);
```

What if we don't use FRP?

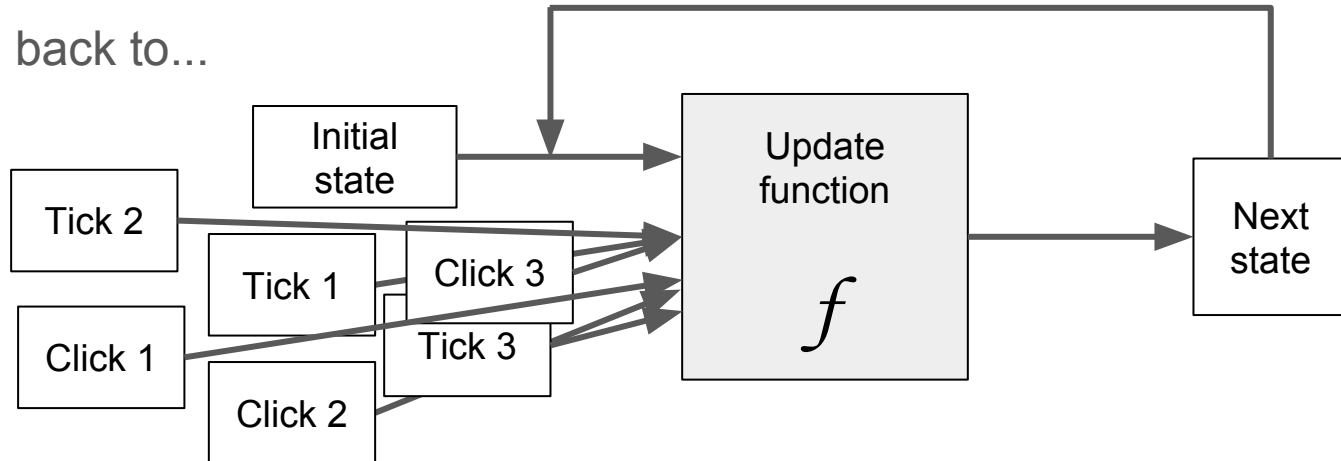
```
root.addEventListener('click', event => {
  worldState = myToggleCell(worldState, event.target.x, event.target.y);
  ReactDOM.render(<Grid world={worldState} ... />, root);
});
```

```
setInterval(() => {
  if (activeState) {
    worldState = myUpdateWorld(worldState);
    ReactDOM.render(<Grid world={worldState} ... />, root);
  }
}, TICK);
```

Why do we trigger re-renders here?

What are the challenges of this architecture?

- How do we create and control test inputs?
- With event handlers, event bubbling, etc. etc., how do we know when events will be processed? How do we know their relative ordering?
- Ticks are no longer modeled the same way as clicks
- Are we back to...

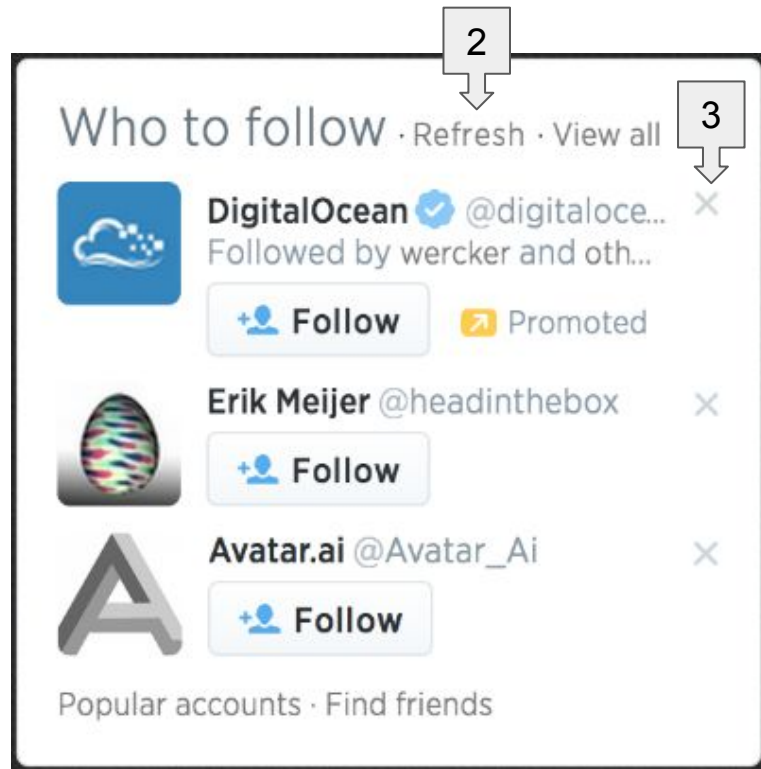


Let's look at a different example...

On Twitter, there is a UI element that suggests other accounts you could follow.

We can implement its core functions using FRP:

1. On startup, load accounts data from the API and display 3 suggestions
2. On clicking "Refresh", load 3 other account suggestions into the 3 rows
3. On clicking the 'x' button on an account row, clear only that current account and display another



“Who to follow” suggestions box implementation

First, let's setup event streams for clicking the 'x' on each account row, and for clicking “Refresh”:

```
var refreshClickStream = Bacon.fromEvent(refreshButton, 'click');  
var close1ClickStream = Bacon.fromEvent(closeButton1, 'click');
```

Now suppose that `makeRequest(suggestionApiUrl)` makes a asynchronous call to the Twitter API, requesting a single account suggestion.

Then we want to:

1. Request 3 suggestions on startup
2. Request 3 suggestions on clicking “Refresh”
3. Request 1 suggestion on clicking the 'x' button on an account row

“Who to follow” suggestions box implementation

```
/* create a click stream for the refresh button */
var refreshClickStream = Bacon.fromEvent(refreshButton, 'click');

/* create a click stream for each 'x' button */
var close1ClickStream = Bacon.fromEvent(closeButton1, 'click');

/* create 3 suggestion streams, adding a fake initial click for startup;
   makeRequest(suggestionApiUrl) makes an API request for 1 suggestion */
var suggestion1Stream = Bacon.mergeAll(refreshClickStream, close1ClickStream)
  .startWith('startup click')
  .flatMap(() => Bacon.fromPromise(makeRequest(suggestionApiUrl)));

/* when a suggestion value is produced, re-render the suggestion component */
suggestion1Stream.onValue(suggestion => renderSuggestion1(suggestion));
```

“Who to follow” suggestions box implementation

Now suppose that `makeRequest(suggestion100ApiUrl)` makes an asynchronous call to the Twitter API, requesting 100 account suggestions.

And suppose that

Questions?

Full demos available [here](#)