Reactive Programming for Couch Potatoes

(Nothing against couch potatoes)

Hi, I'm Andrew.

Friendly neighborhood programmer at Carbon Five.

Carbon Five

I've been an OO programmer for a very long time.

The paradigms there have served me well.

But the functional world was beckoning

- Declarative over imperative
- Pure functions
- Dataflow
- Not having to worry so much about state

I'm a runner.

I've been running for a very long time.

(Consistently injured for just as long.)

How do I stop getting hurt?

Develop a quicker stride rate

Run with a metronome

So you can learn to internalize the correct stride cadence.

I've heard it said:

"Reactive programming is programming with asynchronous data streams."

Let's dive into reactive and make a pedometer.

(Couch potatoes unite.)

Today's talk

- Intro to FRP
- RxJS
- Building a pedometer
- Cycle.js
- Throwing it on a Pebble watch

I've heard it said:

"Reactive programming is programming with asynchronous data streams."

OK. Back up. Let's talk about streams.

Streams are like pipes.

Streams are like pipes.

Streams are like pipes.

Streams are like pipes.

Helpful (?) analogy

Streams are asynchronous arrays that change over time.

```
let prices = [1, 5, 10, 11, 22]

// 5 seconds later...
[1, 5, 10, 11, 22, 44]

// 10 seconds later...
[1, 5, 10, 11, 22, 44, 100]

// And you get the ability to observe changes:
prices.on('data', (thing) => console.log(thing))

// => 100
```

Hopefully more helpful analogy

Streams are like arrays, only:

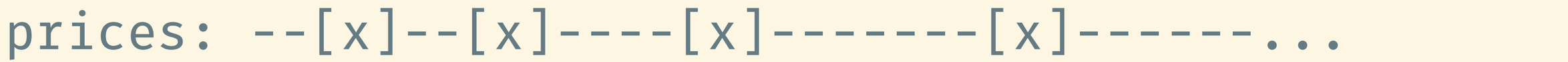
- You can't peek "into" the stream to see the past or future.
- You're holding onto the end of the pipe!
- You can only observe what comes through, at that moment in time.

```
prices: --[1]
```

```
prices: --[x]--[5]
```

```
prices: --[x]--[x]----[10]
```

```
prices: --[x]--[x]----[x]--------[11]
```

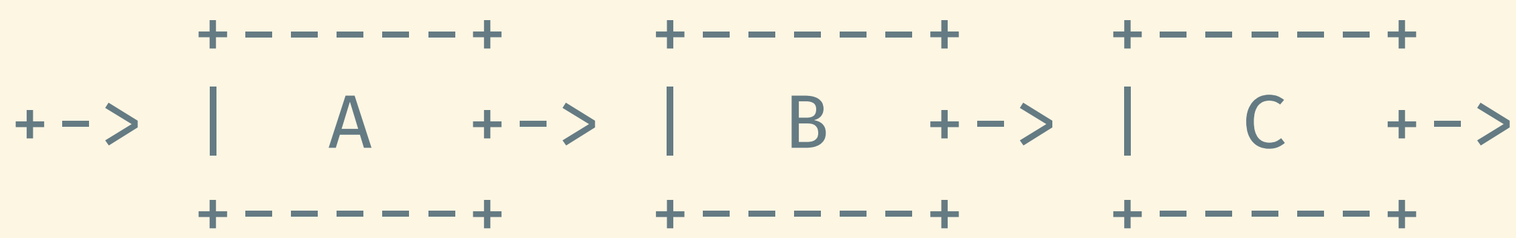

prices: --[x]--[x]----[x]-------[x]------...

Streams are in:

Look familiar?

- Unix pipes: `ls | grep 'foo' > output.log`
- Websockets
- Twitter Streaming API
- Node `streams` lib
- Gulp
- Express

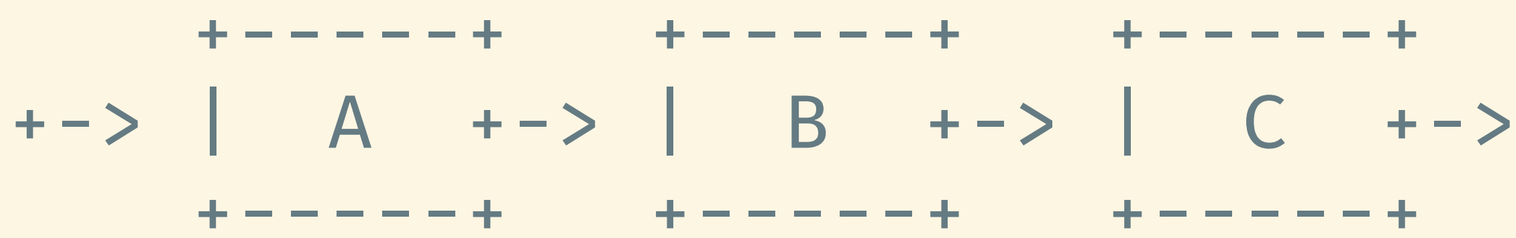
An aside on Node streams



You may be familiar with Node streams.

- Chainable with `pipe()`
- Backpressure capabilities to handle mismatched producers/consumers

An aside on Node streams



- BUT: You must manage stream state: `start`, `data`, `end`.
- BUT: lack of expressive functional operators

Let's forget you've ever heard about them (for now).

"Reactive programming is programming with asynchronous data streams."

streams = Observables

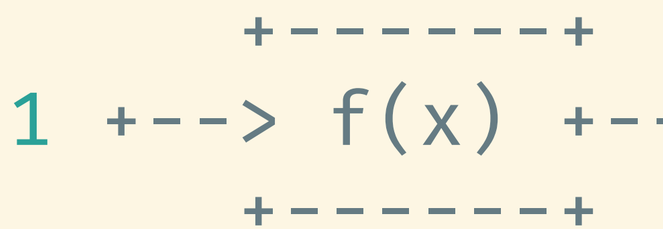
We will use the terms interchangeably tonight.

F is for Functional.

```
let f = (x) => x + 4
```

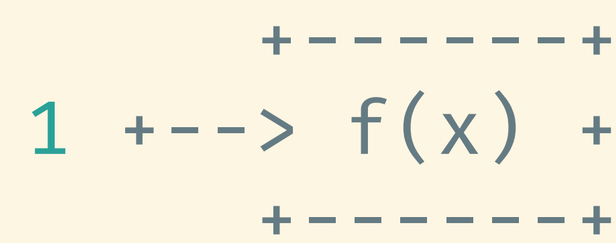
F is for Functional.

```
let f = (x) => x < 10
```



F is for Functional.

```
let f = (x) => x < 10
```



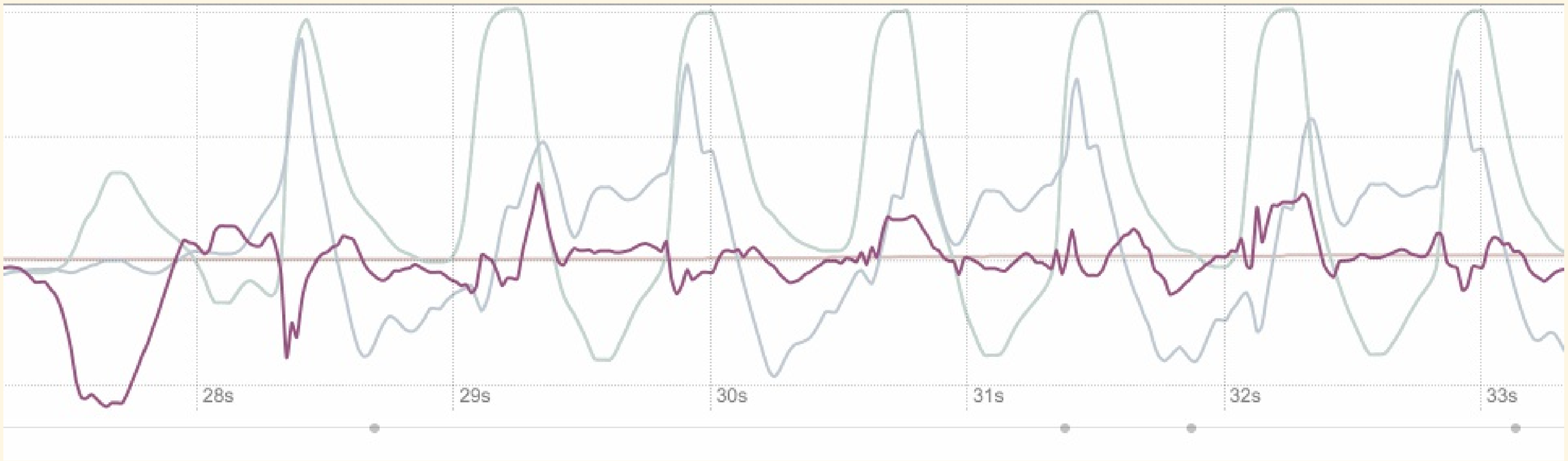
You just wait. There's lots more boxes and arrows where that came from.

Streamify all the things.

Now let's think of these functions in the context of streams.

Step by step, build the pedometer.

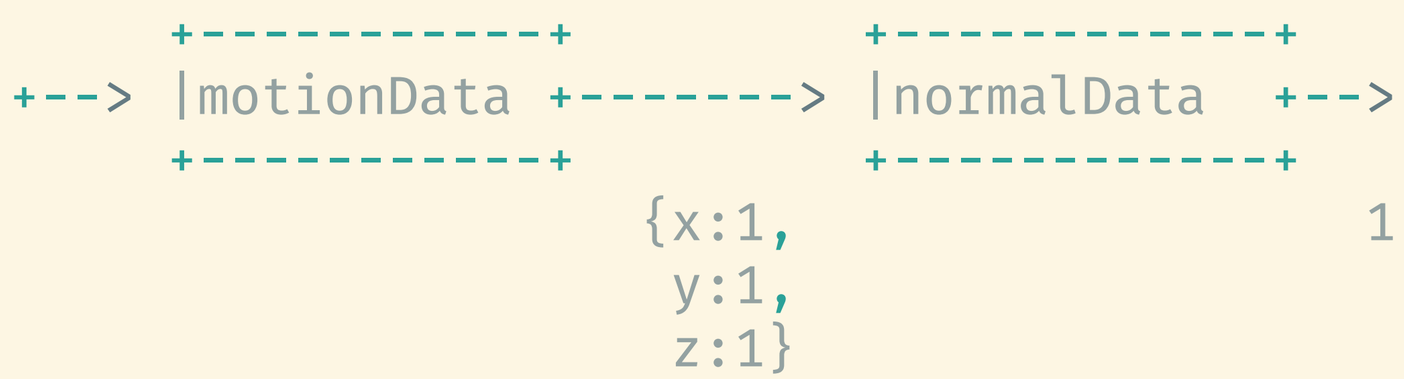
Normalize the Accelerometer data

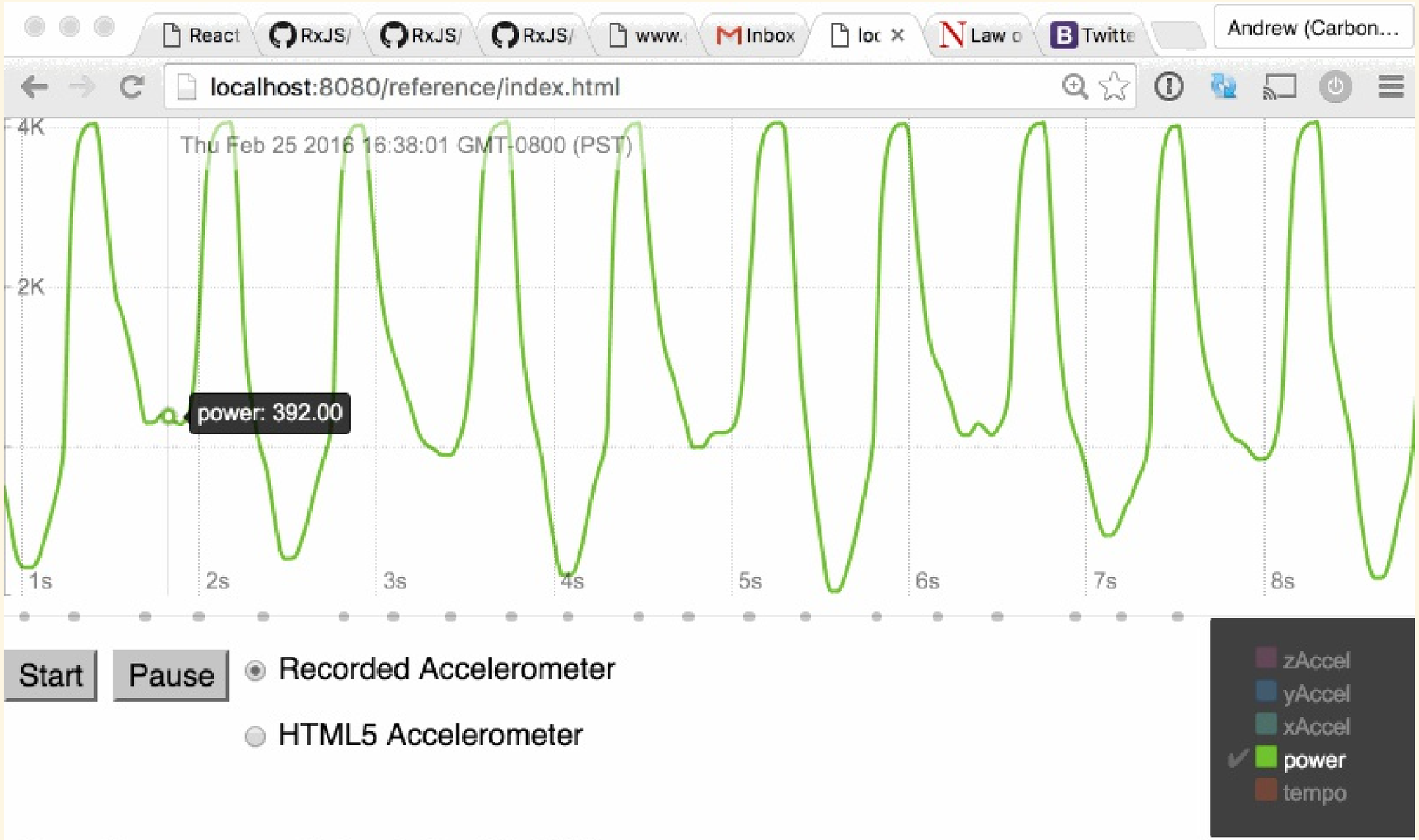


Normalize the Accelerometer data

```
let motionData = eventsFromAccelerometer()
let normalData = motionEvents.map(acceleration => acceleration.y);
```

```
motion: ---[{x:1,y:1,z:1}]--[{x:1,y:2,z:2}]--[{x:1,y:-100,z:1}]-->
normal:  ---[1]--------------[2]--------------[-100]-->
```




React
RxJS/
RxJS/
RxJS/
www.
Inbox
loc
Law o
Twitte
Andrew (Carbon…
localhost:8080/reference/index.html
4K
2K
Thu Feb 25 2016 16:38:01 GMT-0800 (PST)
power: 392.00
1s
2s
3s
4s
5s
6s
7s
8s
Start
Pause
Recorded Accelerometer
HTML5 Accelerometer
zAccel
yAccel
xAccel
power
tempo
Cadence: 98.68 SPM
App state: STARTED
Input device: STUB_INPUT
{"x":1304,"y":-288,"z":-192,"time":1456447087673,"power":-288}

Keep your marbles.

```
  data: ---[1]--[2]--[-100]-->
output: ---[t]--[t]--[f]----->
```

Marble diagrams are a thing: RxMarbles

rxmarbles.com

RxMarbles

Interactive diagrams of Rx Observables

TRANSFORMING OPERATORS

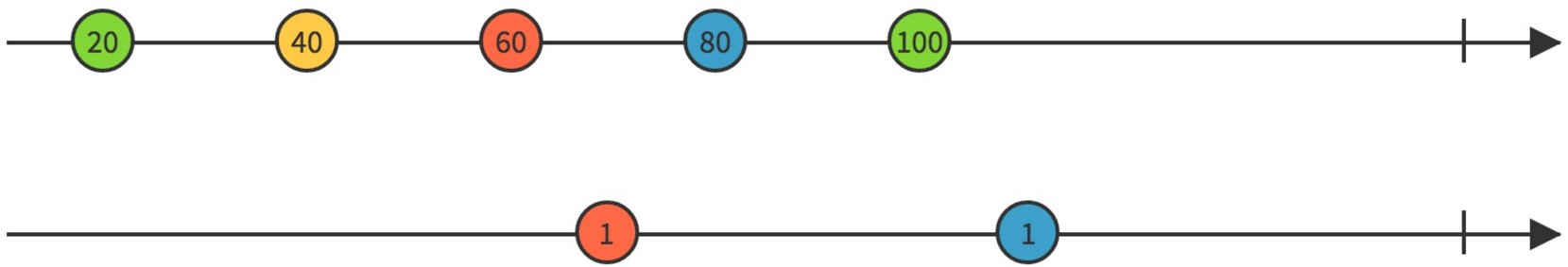
delay
delayWithSelector
findIndex
map
scan
debounce
debounceWithSelector

COMBINING OPERATORS

combineLatest
concat
merge
sample
startWith
withLatestFrom
zip

FILTERING OPERATORS

distinct
distinctUntilChanged
elementAt
filter
find
first
last
pausable



merge

Your Rx functional toolbelt

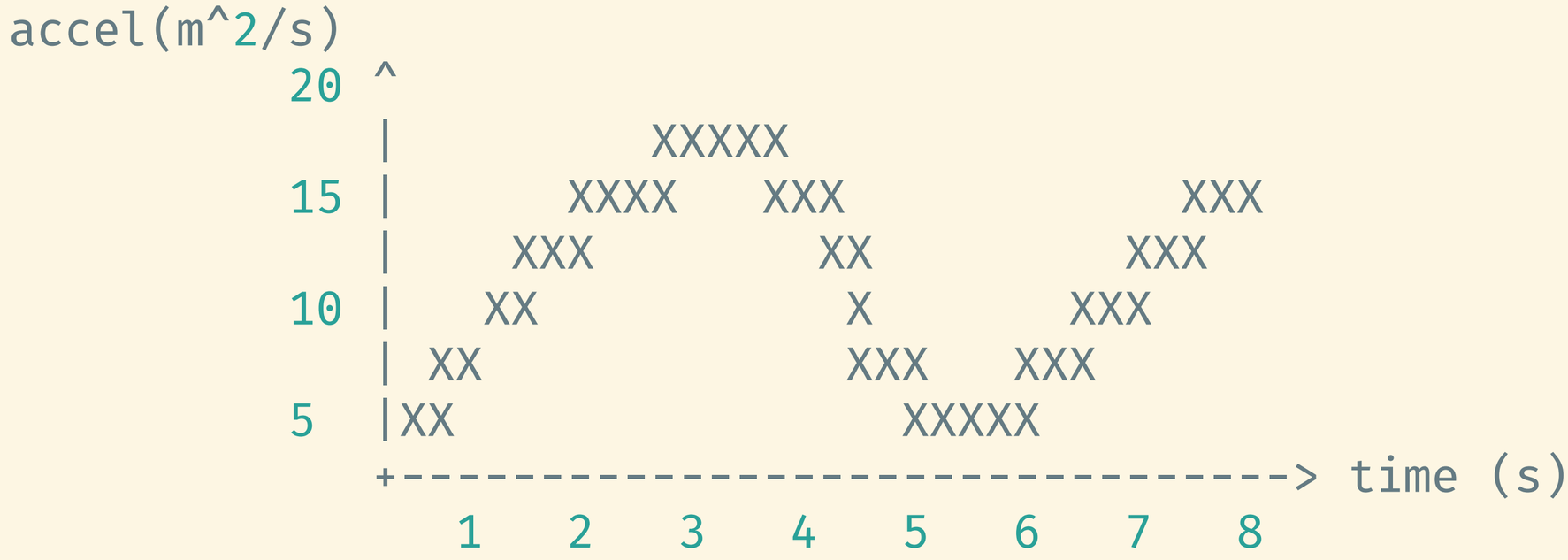
- `map`
- `filter`
- `zip`
- `flatMap/concatMap`
- `reduce/scan`
- `debounce`
- `combineWithLatest`
- `merge`

Your Rx functional toolbelt

- `map`
- `filter`
- `zip`
- `flatMap/concatMap`
- `reduce/scan`
- `debounce`
- `combineWithLatest`
- `merge`

Special stream-oriented semantics here.

Peaks and troughs = steps

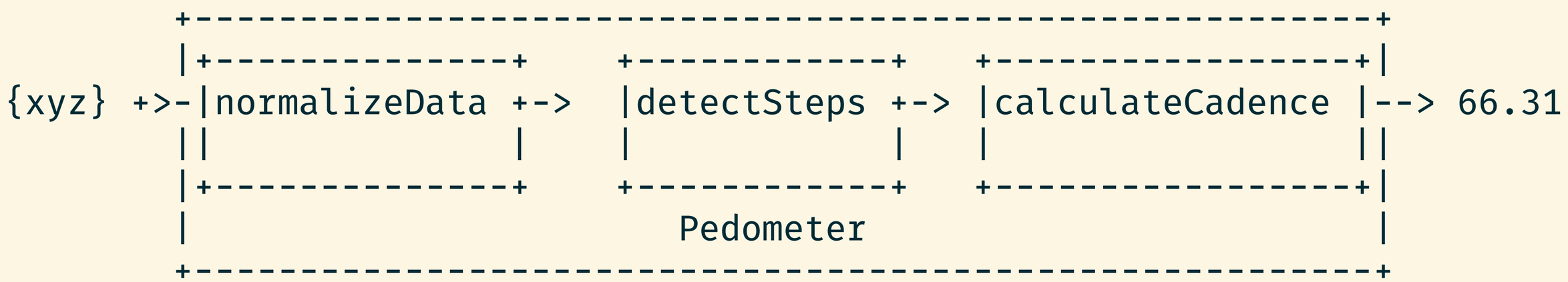



```
delta:      -[5]-[4]-[2]-[-2]-[-5]-[2]-[3]-[4]->
change:     -[+]-[+]-[+]-[-]--[-]--[+]-[+]-[+]->
stepEvents: -------------[S]-------[S]--------->
```

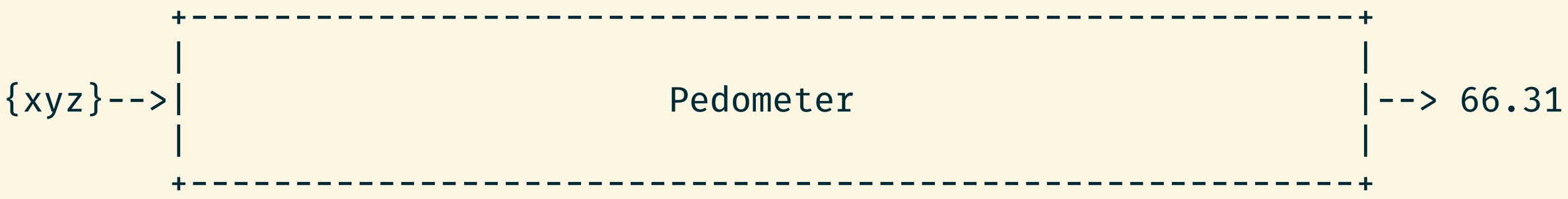
```
// stream items of form: { power: 10, time: 1432485925 }
function detectSteps(stream) {
  return stream
  // Group elements in sliding window of pairs
  .pairwise()
  // Calculate change and step signals
  .map(([e1, e2]) => {
    return {
      "timestamp": e1.time,
      "diff": e2.power - e1.power,
    }
  })
  .map(v => Object.assign(v, { changeSignal: (v > 0) ? '+' : '-' }))
  // Every time a changeSignal flips, then the event
  // becomes a step signal.
  .distinctUntilChanged(v => v.changeSignal)
  // Smooth out erratic changes in motion.
  .debounce(DEBOUNCE_THRESHOLD)
};
```

Voila! A beautiful pedometer.

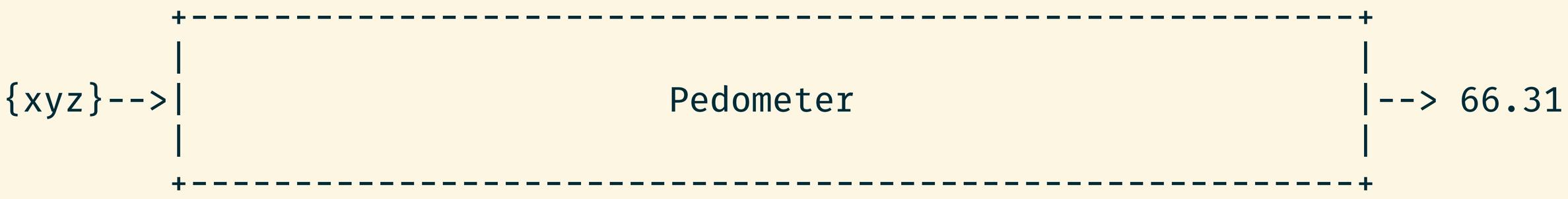
Voila! A beautiful pedometer.



Voila! A beautiful pedometer.



Voila! A beautiful pedometer.



Essentially one long transformation.

More complicated things lie on the horizon.

Full-featured, rich apps need:

- state management
- composability
- modularity

(Psst. We'll only need streams.)

FRP apps follow a common pattern:

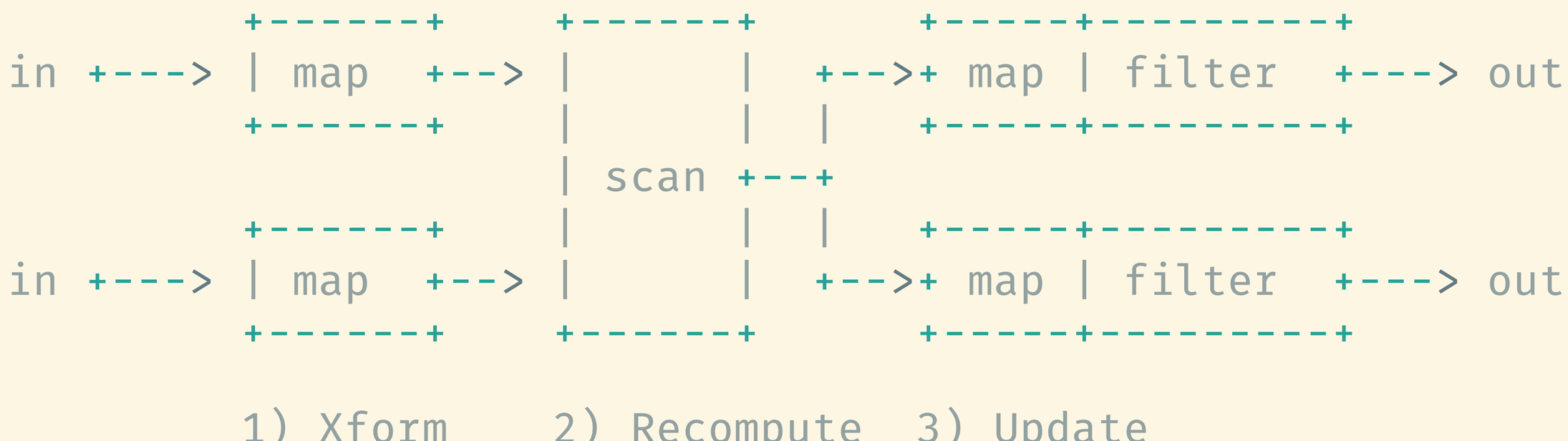
1. **Transform inputs** with `map()`

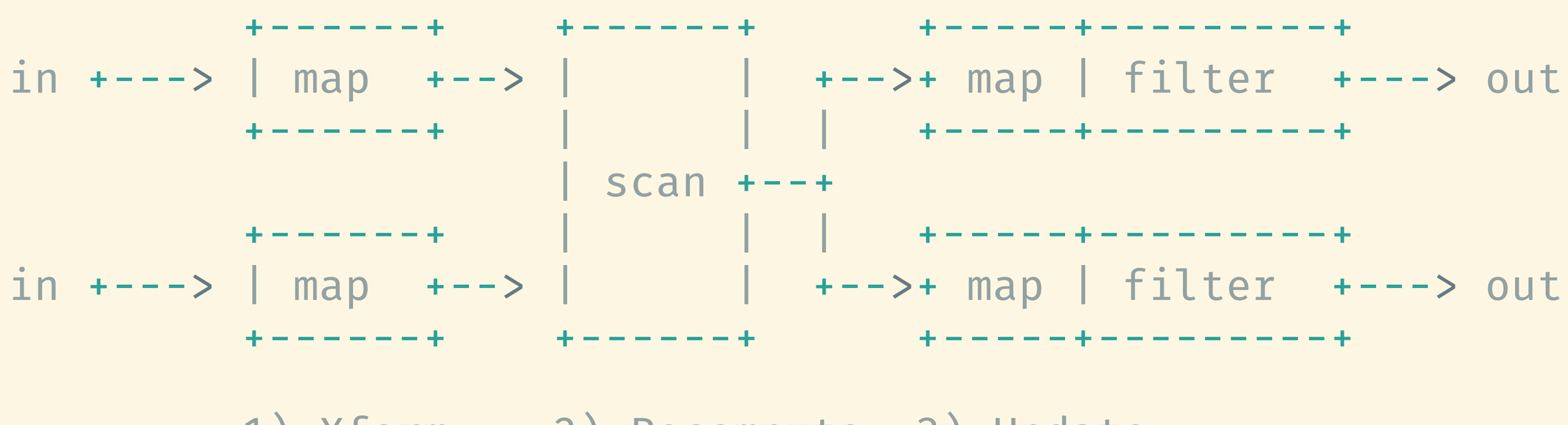
FRP apps follow a common pattern:

1. **Transform inputs** with `map()`
2. **Recompute state** with `scan()`

FRP apps follow a common pattern:

1. **Transform inputs** with `map()`
2. **Recompute state** with `scan()`
3. **Update outputs** with `map()` and `filter()`


in
map
map
scan
map | filter
map | filter
out
1) Xform
2) Recompute
3) Update



- `in`: DOM events. Domain events. HTTP responses.
- `out`: DOM updates. Domain events. HTTP requests.

1. Transform inputs with `map()`

Ask yourself: What are the inputs into the app?

Well, we have our accelerometer.

```
let accelerometerData =
  Rx.Observable.fromEvent(window,
                          'devicemotion')
  .map(e => Object.assign({}, e.accelerometer))

// accelerometerData: --[ { x:, y:, z: } ]--->
```

Well, we have our accelerometer.

```
let accelerometerData =
  Rx.Observable.fromEvent(window,
                          'devicemotion')
  .map(e => Object.assign({}, e.accelerometer))

// accelerometerData: --[ { x:, y:, z: } ]--->
```

We should also plug that into our pedometer.

```
let cadence = connectPedometer(accelerometerData)
.map(cadence => ({ name: CADENCE_EMITTED,
                   value: cadence }))

// cadence: --[ { name: CADENCE_EMITTED, value: 66.1234 } ]-->
```

Transform inputs, cont'd

There's also DOM event data to account for:

```
let startButton =
  Rx.Observable.fromEvent($('button#start'), 'click')
    .map((e) => { { name: 'START' } } }

// startButton: --[ { name: 'START' } ]-->
```

Merge these together into an input stream:

```
let actions = Rx.Observable.merge(
    startButton,
    cadence
)

// actions: --[ { name: START } ]--
//             [ { name: CADENCE_EMITTED, value: 66.1234 } ] -->
```

2. Recompute application state with `scan()`.

Ask yourself: What is the minimum amount of state that my app needs to store?

- Anything that the UI is dependent upon
- Anything that stores a value that is necessary for future events to compute from.

Application state for this app:

```
const initialState = {
  cadence: 0,
  runState: STOPPED,
}
```

Application state for this app:

```
const initialState = {
  cadence: 0,
  runState: STOPPED,
}
```

Note how it is a simple data structure.

Next we update the application state based on the current (incoming) event.

```
let currentState = actions.scan(
  (oldState, action) => {
    switch(action.name) {
    case START:
      return Object.assign({}, oldState { runState: STARTED })
    case CADENCE_EMITTED:
      return Object.assign({}, oldState, { cadence: action.value })
    default:
      return oldState
    }
  }, initialState)
  .startWith(initialState);

// actions: -----------------[START]--------[CADENCE_EMITTED, 66.12]->
// current: -[{STOPPED, 0}]--[{STARTED,0}]--[{STARTED, 66.12}]-->
```

3. Update outputs (UI) upon state change.

Conditionally update the UI based on the state of the app's `runState`.

```
currentState
.filter(newState => newState.runState === STARTED)
.subscribe(newState => {
  $('.output').text(
    `${newState.cadence} steps per minute (SPM)`
  );
});
```

Extra RxJS bit: call `subscribe` to attach an observer and "activate" the stream.

```
currentState.subscribe(newState => {
  // Perform side effects like:
  // Render the DOM
  // Make an HTTP request
  // Push an event onto a Websocket
});
```

Extra RxJS bit: call `subscribe` to attach an observer and "activate" the stream.

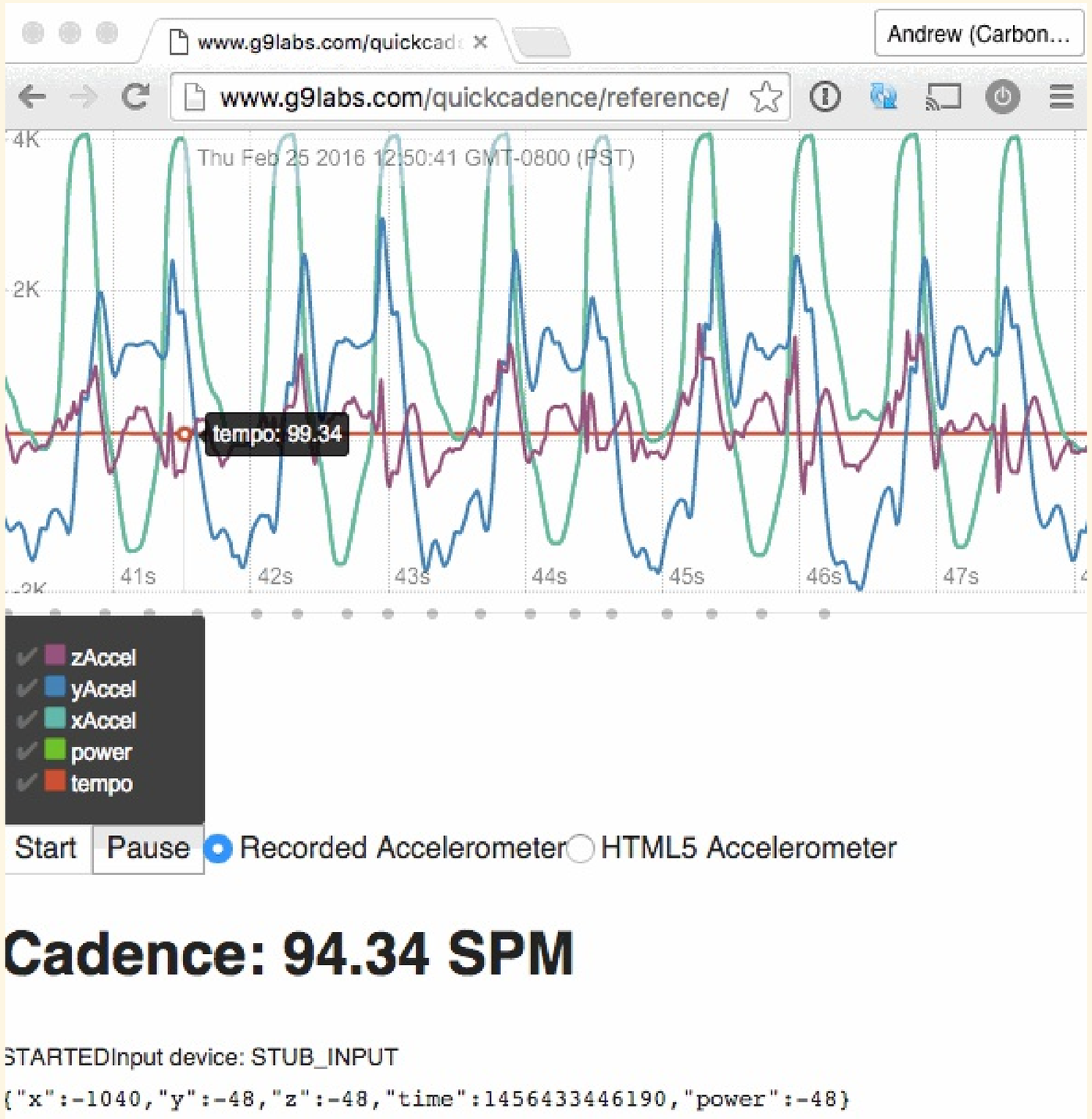
```
currentState.subscribe(newState => {
  // Perform side effects like:
  // Render the DOM
  // Make an HTTP request
  // Push an event onto a Websocket
});
```

(Cold) streams produce values only after an Observer attaches.

Phew! Let's see it in action.

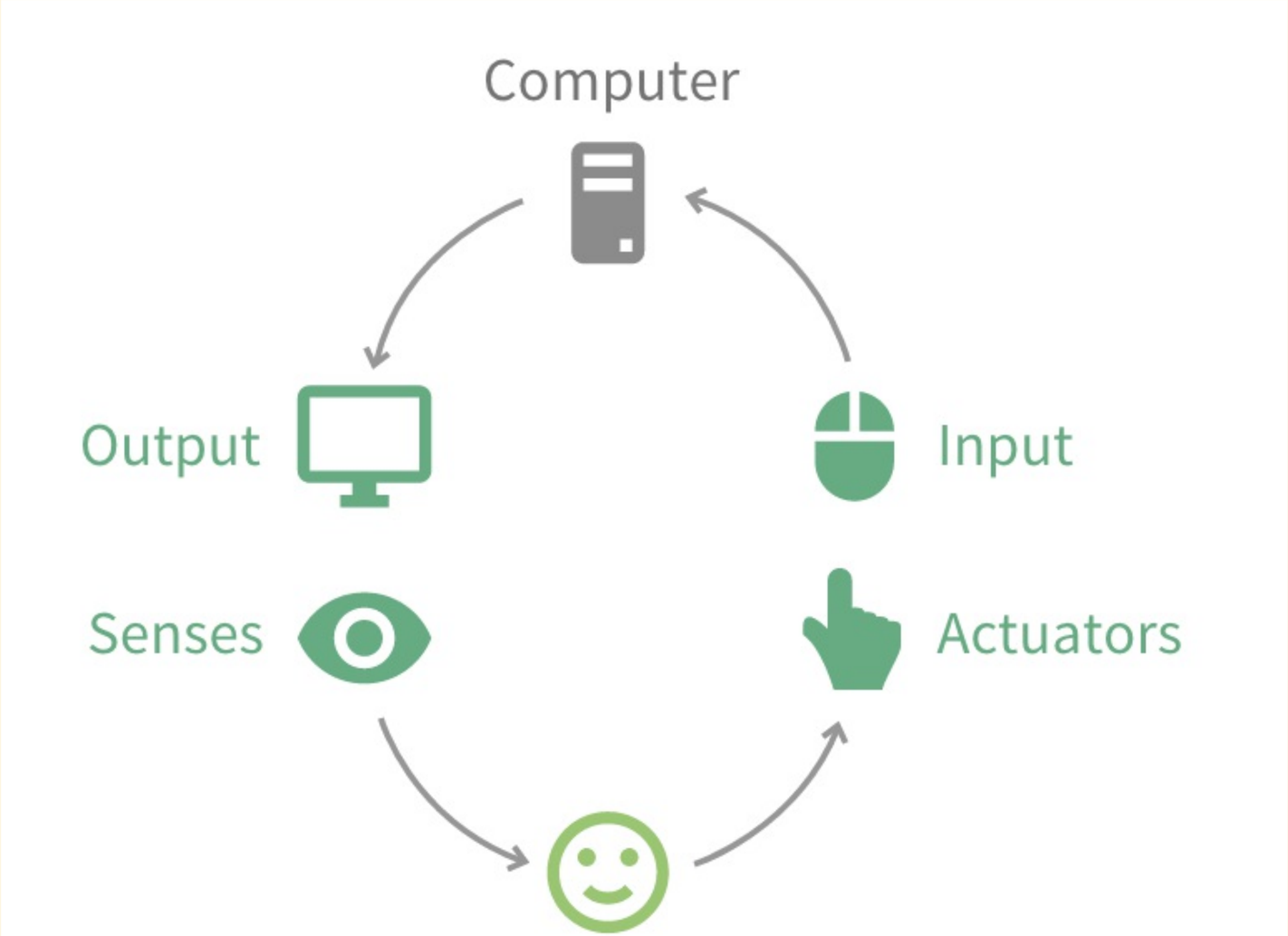
http://tinyurl.com/rxcadence

(Open this on your phone!)


www.g9labs.com/quickcad
Andrew (Carbon…
www.g9labs.com/quickcadence/reference/
4K
Thu Feb 25 2016 12:50:41 GMT-0800 (PST)
2K
tempo: 99.34
41s
42s
43s
44s
45s
46s
47s
zAccel
yAccel
xAccel
power
tempo
Start
Pause
Recorded Accelerometer
HTML5 Accelerometer
Cadence: 94.34 SPM
STARTEDInput device: STUB_INPUT
{"x":-1040,"y":-48,"z":-48,"time":1456433446190,"power":-48}

Zoom out: Organizing your FRP app with Cycle.js

High level insight from Cycle: apps are really feedback loops:



Dialogue Abstraction

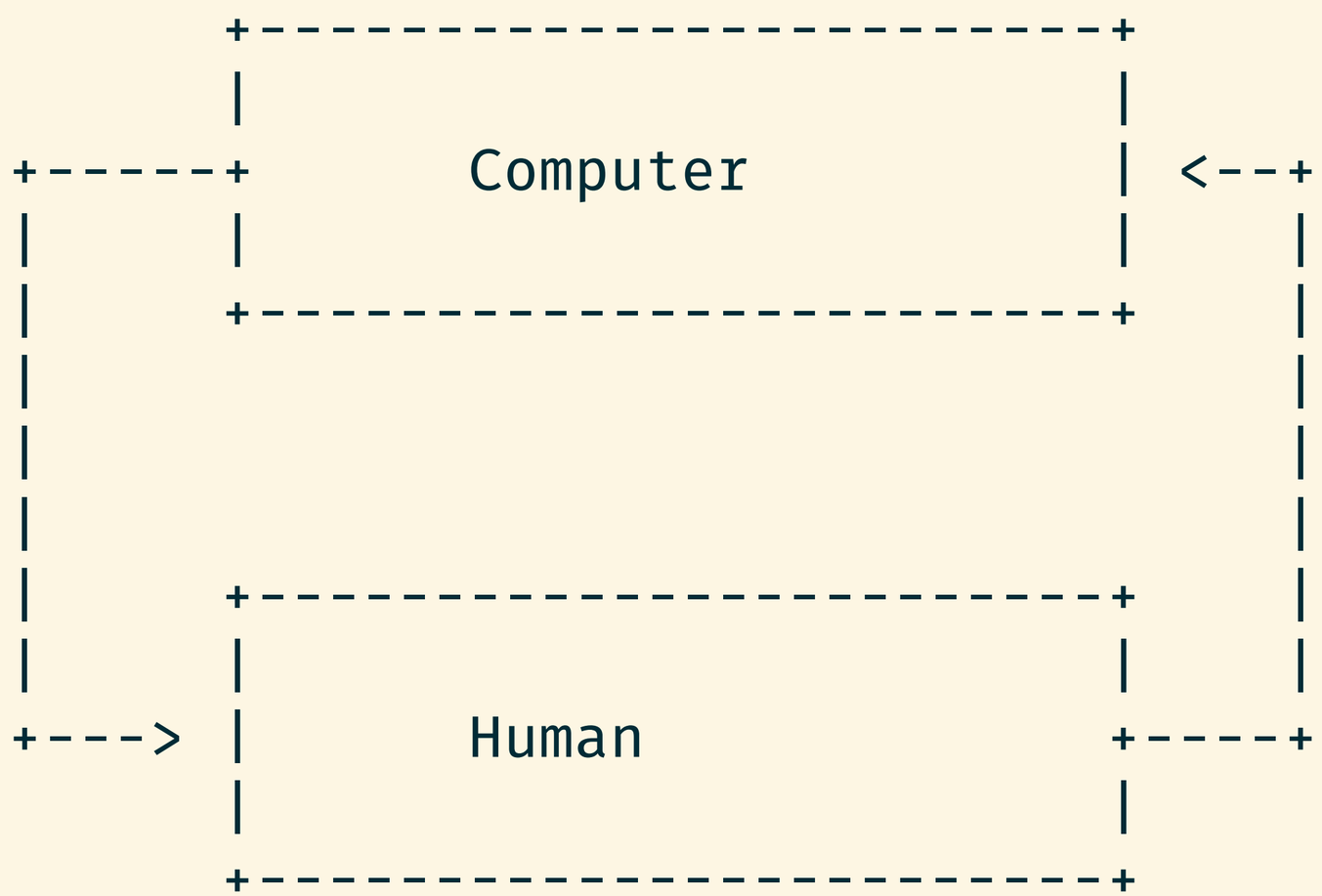
The computer is a function,

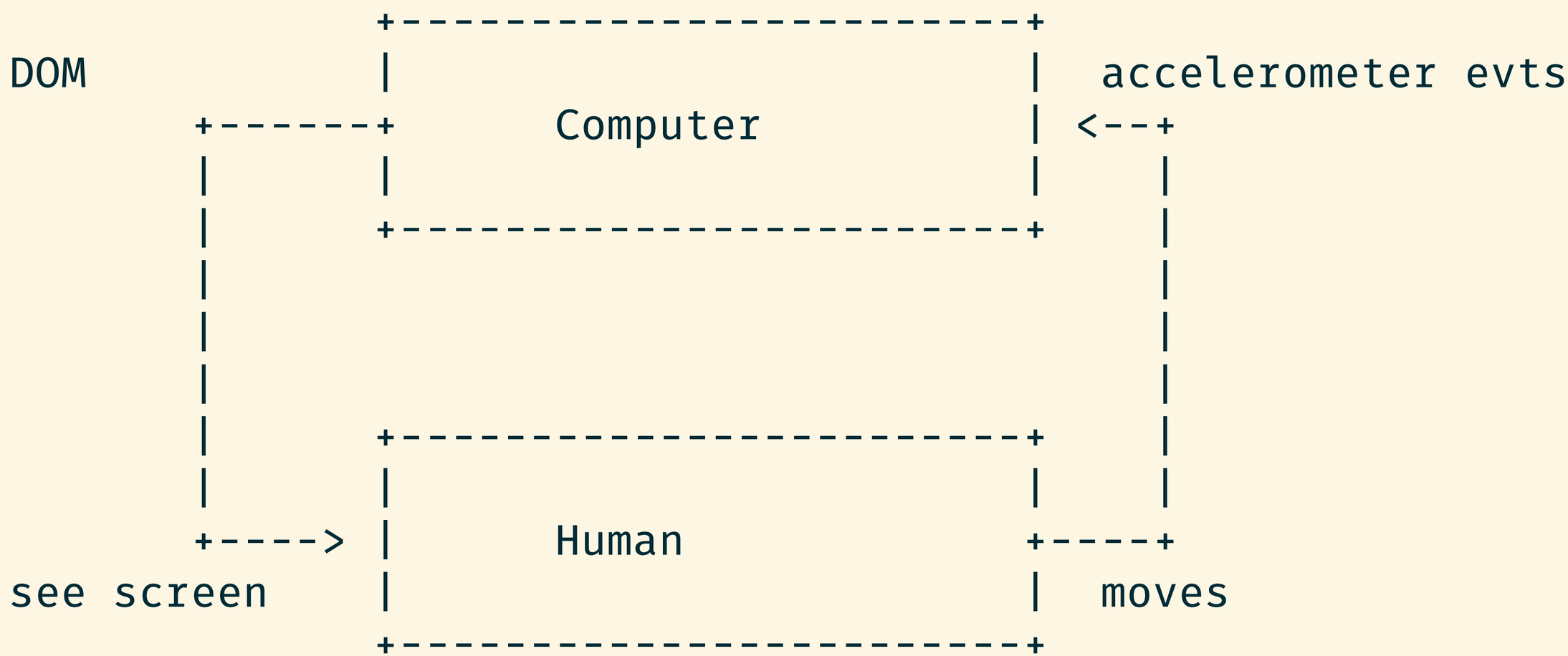
- Taking **inputs** from the keyboard, mouse, touchscreen,
- and **outputs** through the screen, vibration, speakers.

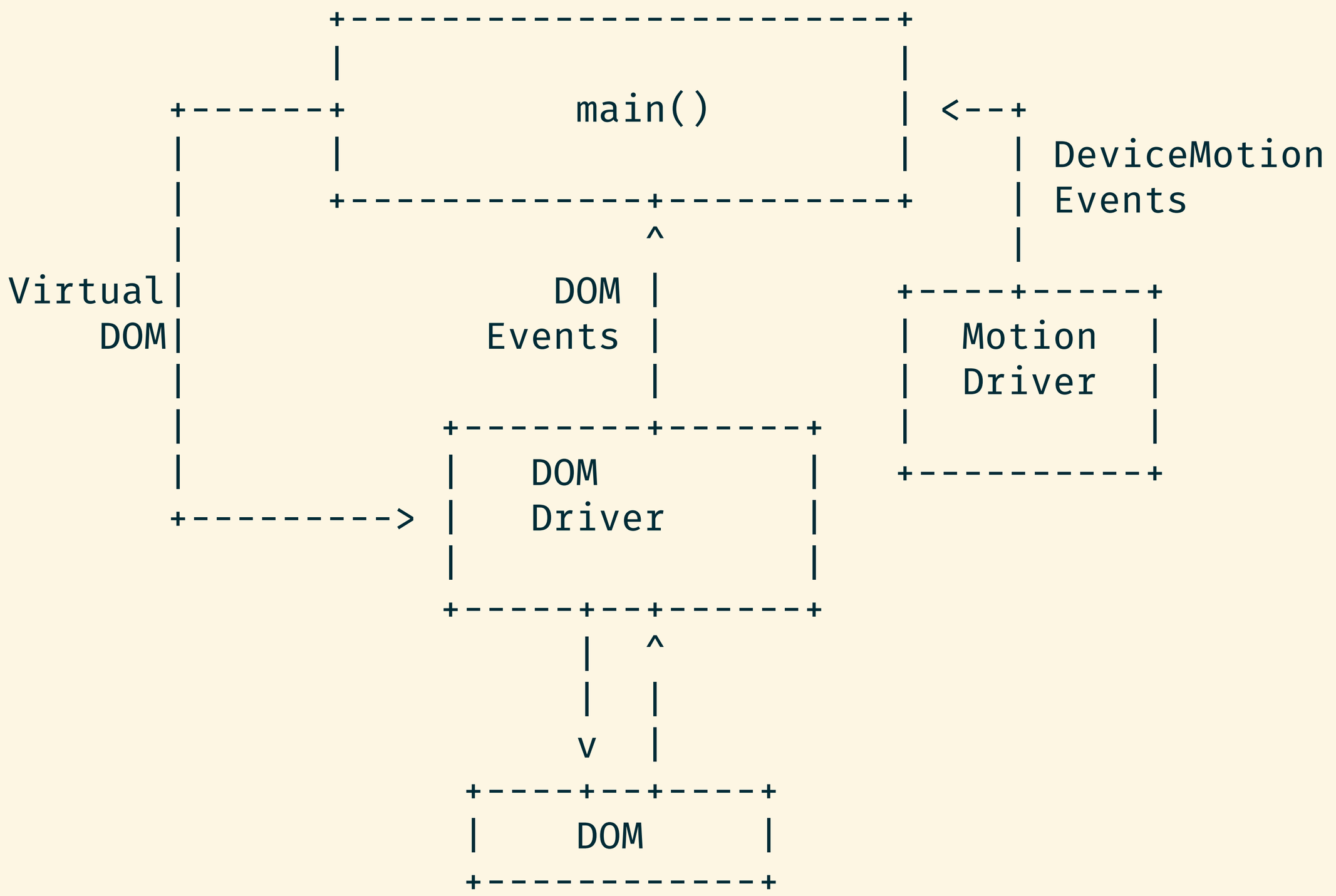
The human is a function,

- Taking **inputs** from their eyes, hands, ears,
- and **outputs** through their fingers.

Inputs and outputs you say?


Computer
Human


DOM
Computer
accelerometer evts
see screen
Human
moves


main()
DeviceMotion
Events
Virtual
DOM
DOM
Events
Motion
Driver
DOM
Driver
DOM

```
import Rx from 'rx';
import Cycle from '@cycle/core';
import {div, input, p, makeDOMDriver} from '@cycle/dom';

function main(sources) {
  const sinks = {
    DOM: sources.DOM.select('input').events('change')
      .map(ev => ev.target.checked)
      .startWith(false)
      .map(toggled =>
        div([
          input({type: 'checkbox'}), 'Toggle me',
          p(toggled ? 'ON' : 'off')
        ])
      )
  };
  return sinks;
}

Cycle.run(main, {
  DOM: makeDOMDriver('#app')
});
```

See how it's done:

- Cycle.js Introduction
- RxCadence test harness app

FRP reducer pattern

- Cycle.js: Reducer pattern
- Redux: Actions/Reducers/React
- Elm: Model/Update/View

Oh, about the Pebble

You can load arbitrary Javascript libraries (like RxJS) on a Pebble!

Cloud Pebble: http://www.cloudpebble.com

PebbleJS: https://pebble.github.io/pebblejs/

https://cloudpebble.net/ide/project/183505#

CLOUDPEBBLE DOCUMENTATION PROJECTS SETTINGS SIGN OUT

SWEETCADENCE

SETTINGS

TIMELINE (PREVIEW)

COMPILATION

GITHUB

SOURCE FILES ADD NEW

accelerometerManager.js

appController.js

app.js

dummyAppController.js

js/RxCadence.js

js/vendor/bacon.js

js/vendor/objectInspect.js

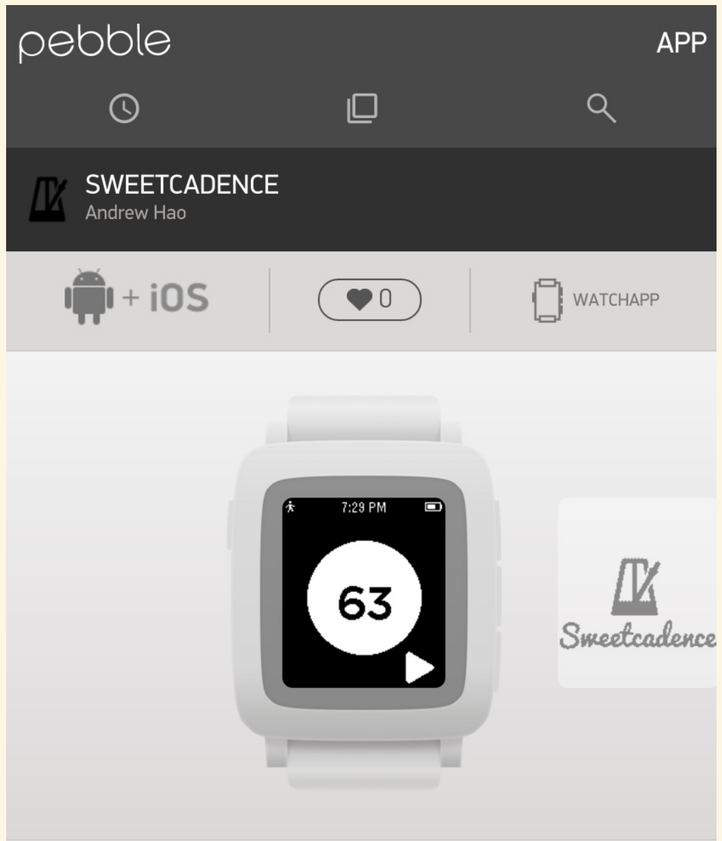
js/vendor/rx.all.js

js/vendor/rx.lite.js

```
1  // Wraps around the Accel library and provides higher-level event stream.
2  var Bacon = require('./js/vendor/bacon'),
3      RxCadence = require('./js/RxCadence').default,
4      inspect = require('./js/vendor/objectInspect'),
5      Rx = require('./js/vendor/rx.all');
6
7  var AccelerometerManager = function(accel, config) {
8    this.config = config || {rate: 10, samples: 25};
9    this.accel = accel;
10 };
11
12 AccelerometerManager.prototype = {
13   init: function() {
14     this.accel.config(this.config);
15     this.accel.init();
16     this._startRecording();
17   },
18
19   /**
20    * Returns a Stream of events
21    * @return Bacon.EventStream Stream of Floats representing cadence counts
22    */
23   getCadenceStream: function() {
24     return this.cadenceStream;
25   },
26
27   /**
28    * Sets up and connects the accelerometer events to a stream.
29    * @param Function callback
30    */
31   _startRecording: function() {
32     var dataStream = Rx.Observable.fromEvent(this.accel, 'data')
```


English

Sweetcadence



DESCRIPTION

Track the tempo of your runs with Sweetcadence on your wrist. You now have the ability to measure how fast your legs

Recap

- Learn to see everything as a stream.
- Slowly build your tool familiarity with RxJS. They are powerful, but they have a learning curve.

Recap (cont'd)

- Reducer pattern:
  - Map inputs
  - Recompute state
  - Update output
- Abstract your app as a dialogue between the user and the system.

Further reading (and many thanks!)

- "The Introduction to Reactive Programming You've Been Missing"
- "OMG Streams!"
- RxMarbles: http://rxmarbles.com/
- ReactiveX: http://reactivex.io/learnrx/
- Cycle.js docs: http://cycle.js.org

Thanks!

Github: andrewhao

Twitter: @andrewhao

Email: andrew@carbonfive.com

Image attributions:

- https://www.flickr.com/photos/alphageek/210677885/
- https://www.flickr.com/photos/95744554@N00/156855367/
- https://www.flickr.com/photos/autowitch/4271929/
- https://www.flickr.com/photos/internetarchivebookimages/
-