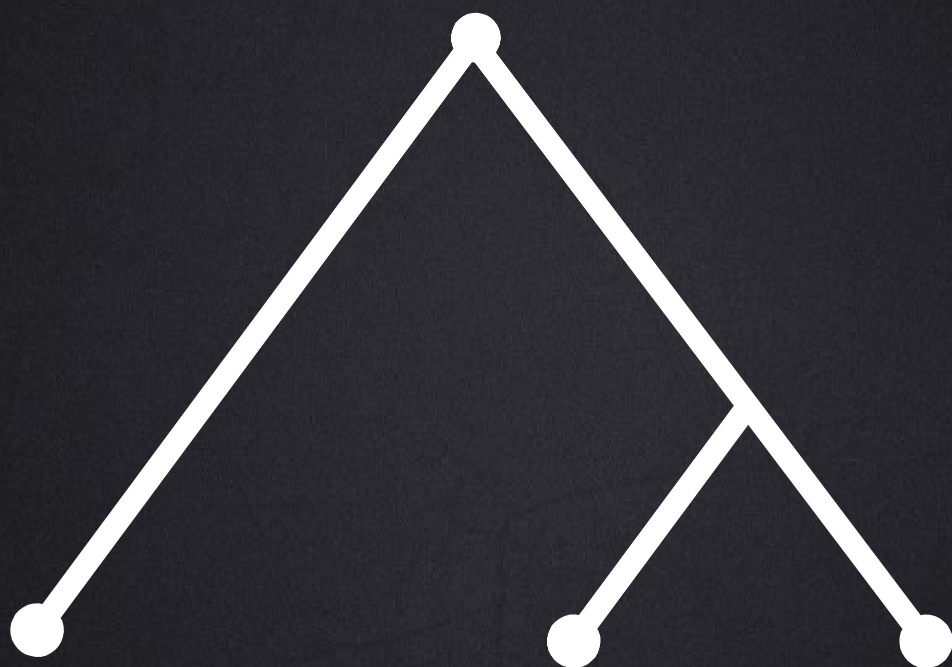
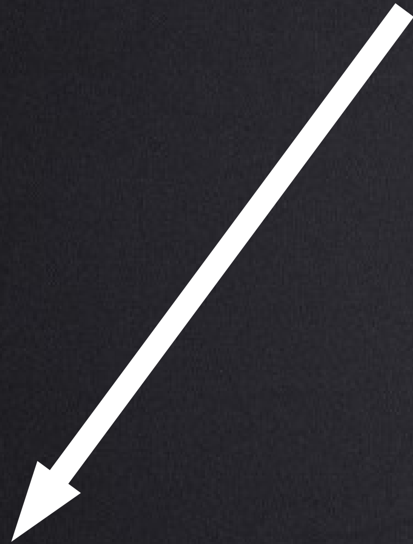
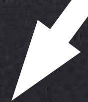
FRP in ClojureScript with Javelin

Alan Dipert
@alandipert

THE freshdiet
Fresh, Gourmet, Delivered Daily







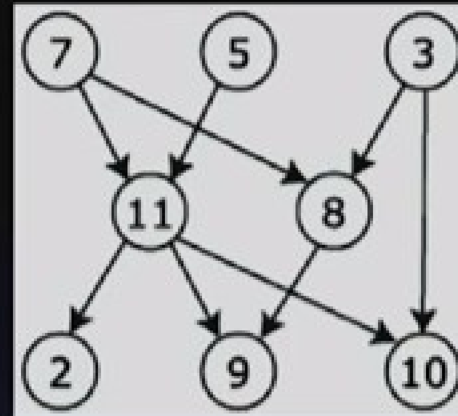
nginx
jetty
syslog-ng
redis
postgres

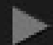
relevance

2012

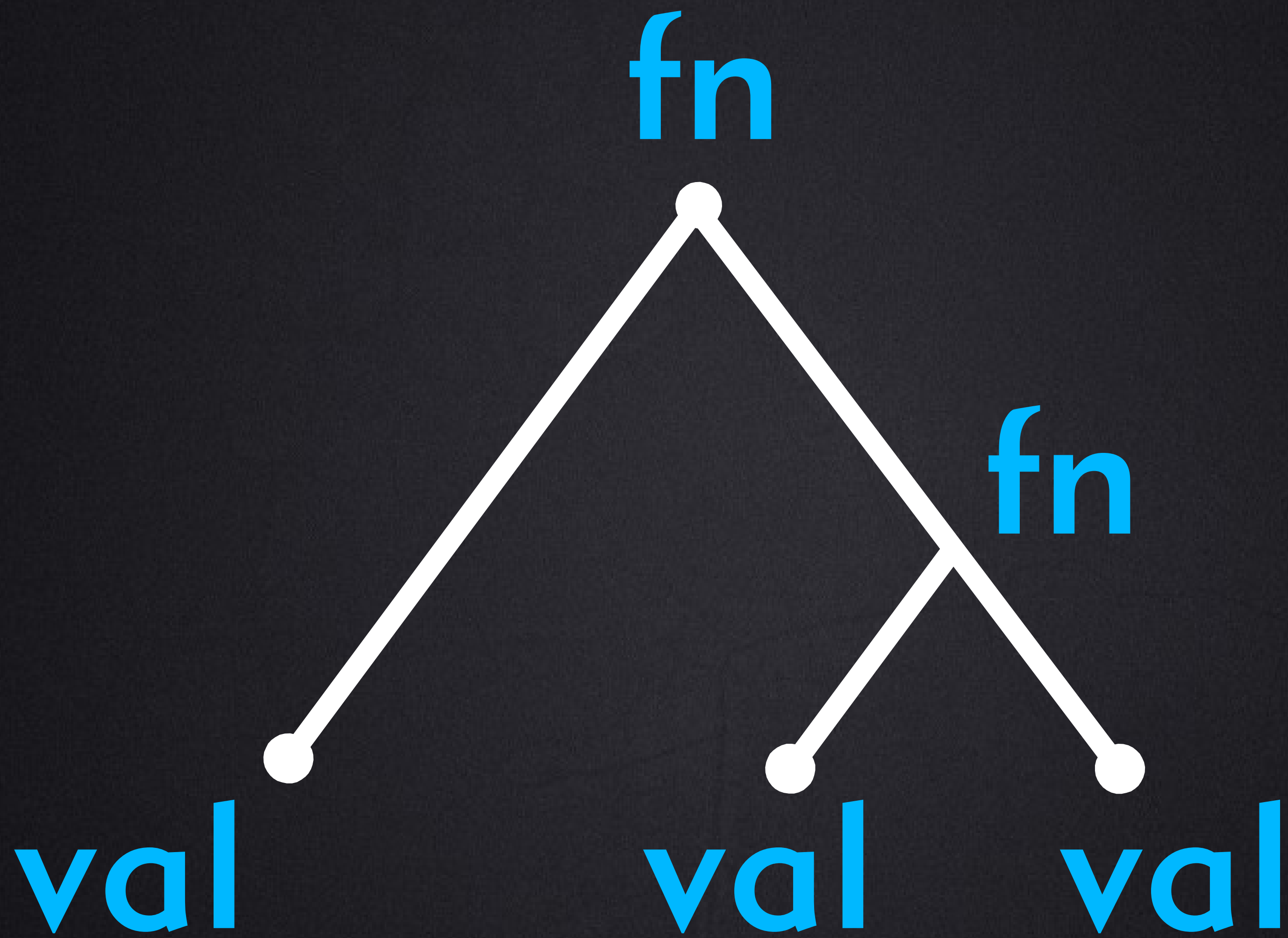
Read Between the Lines

- Where do the semantics of a system live?

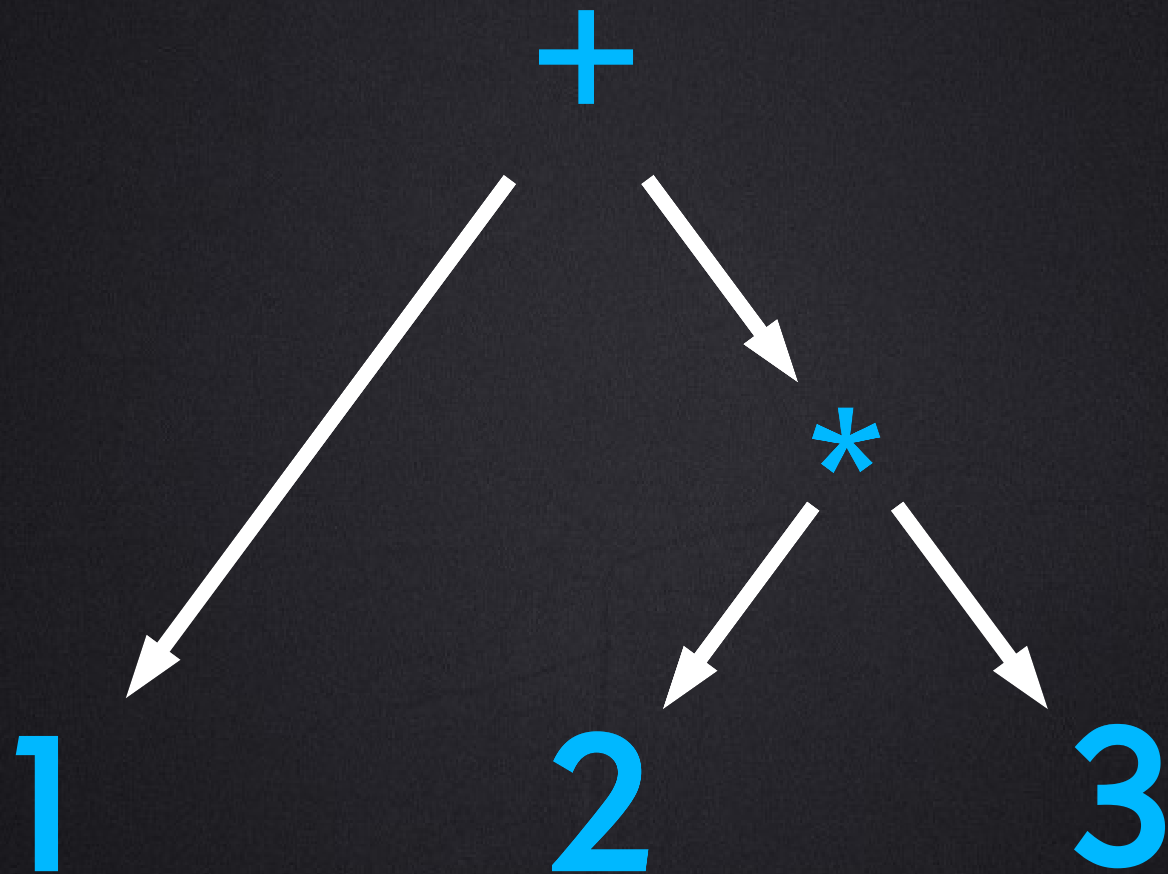




0:26:09 / 1:02:50


fn
fn
val
val
val

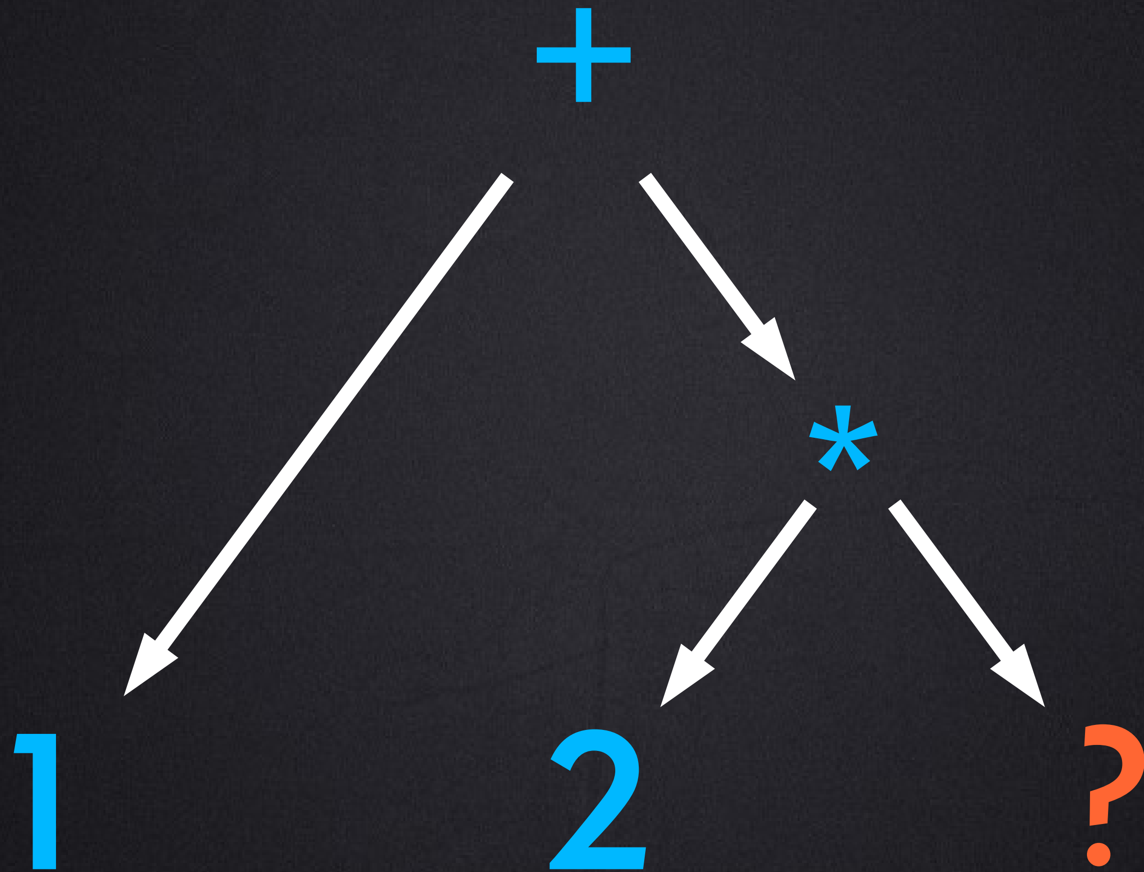
```
(+ 1 (* 2 3))
```



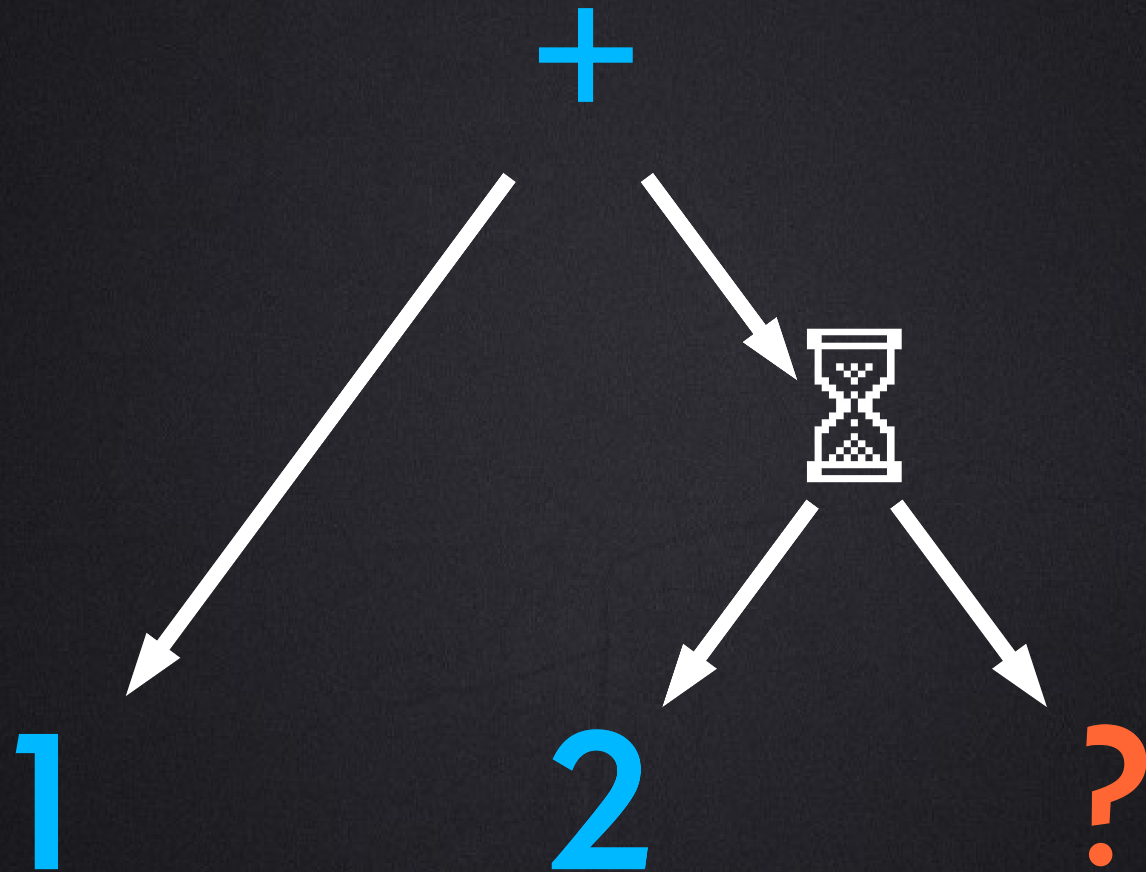
Applicative Evaluation

- Evaluate arguments
  - Syntax specifies order
- Apply arguments to functions
- No notion of time
  - Almost: in Clojure, CL args eval left to right

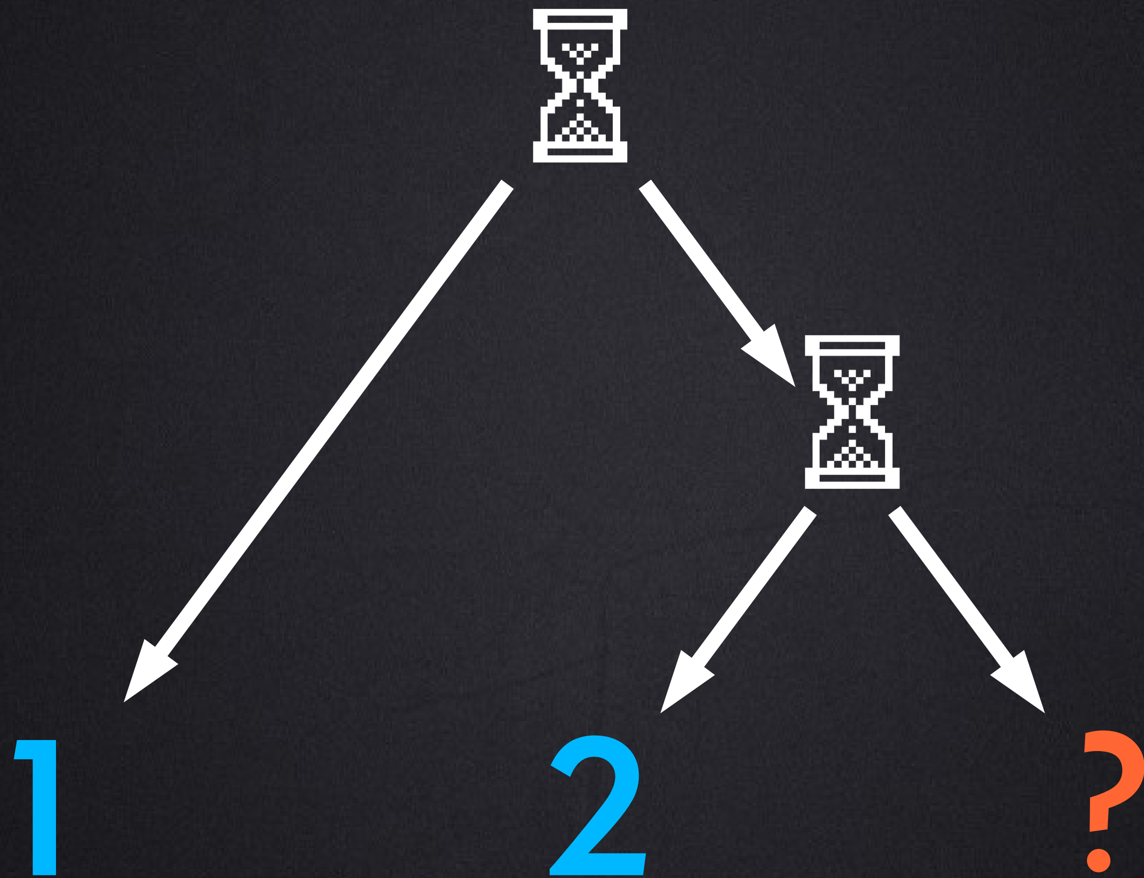
```
(+ 1 (* 2 ?))
```



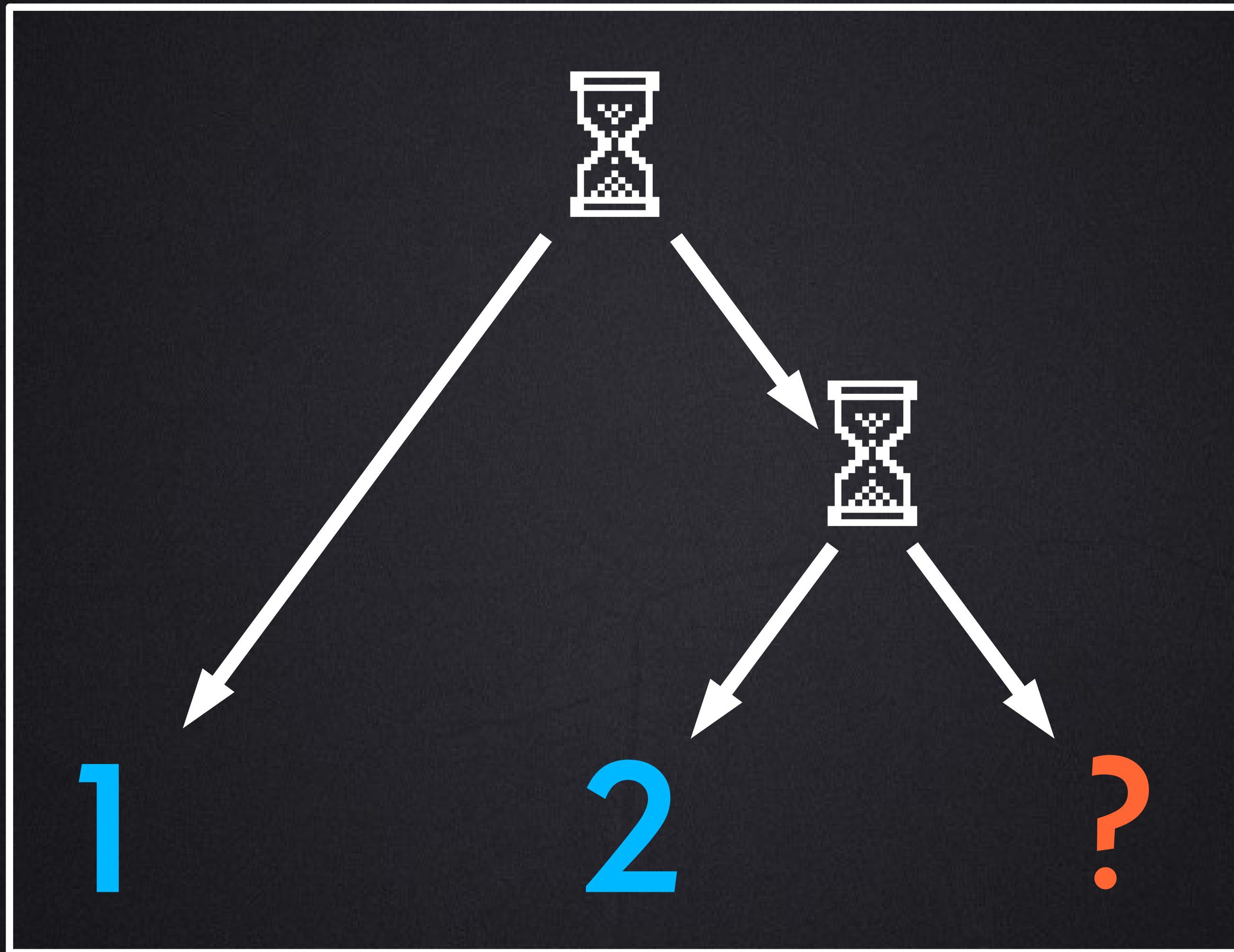
```
(+ 1 (* 2 ?))
```



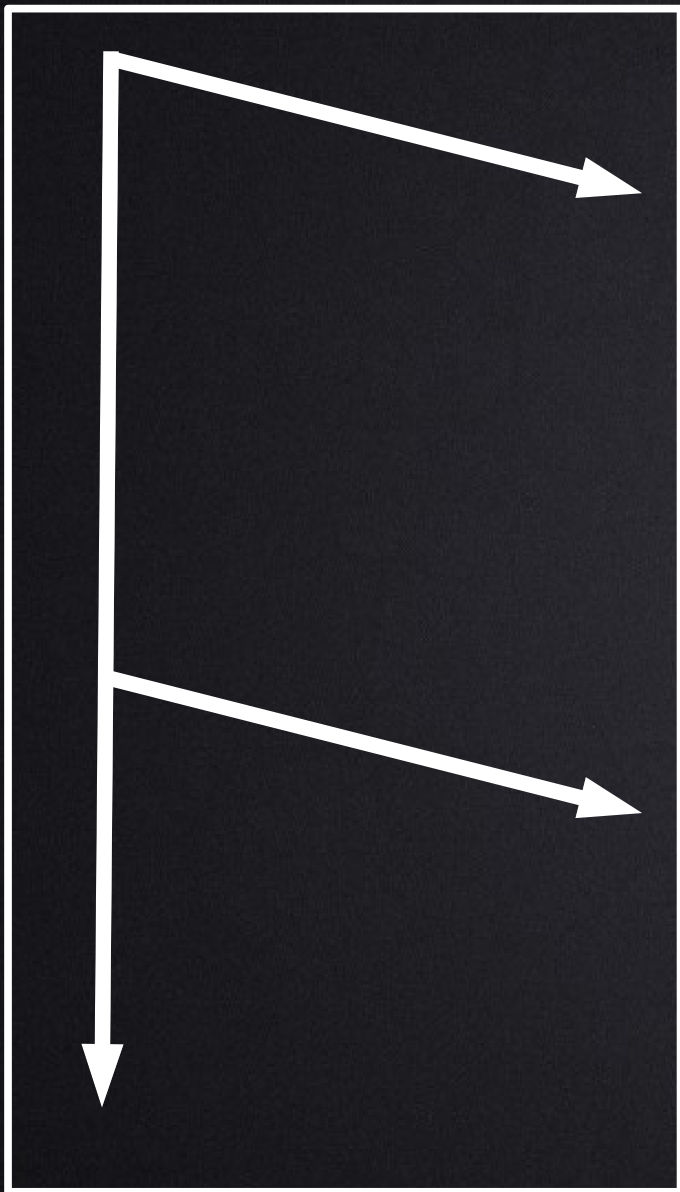
```
(+ 1 (* 2 ?))
```



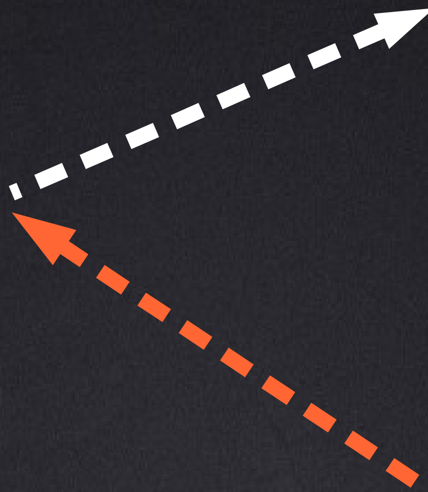
Thread

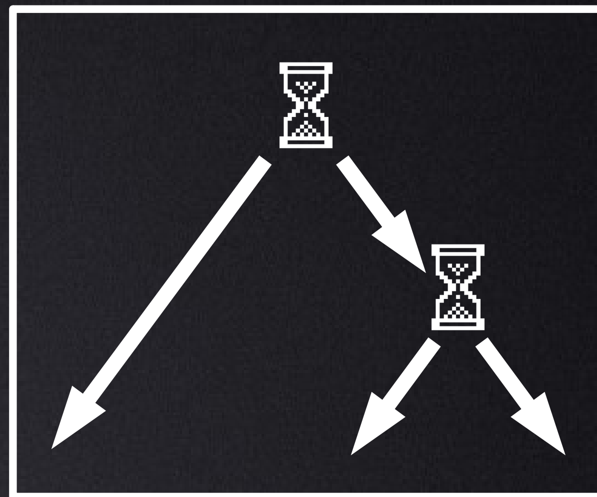


Parent Thread

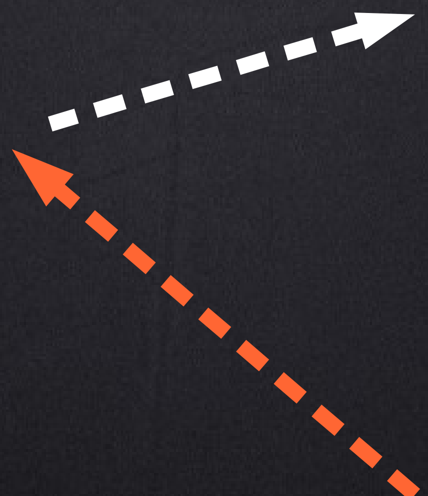


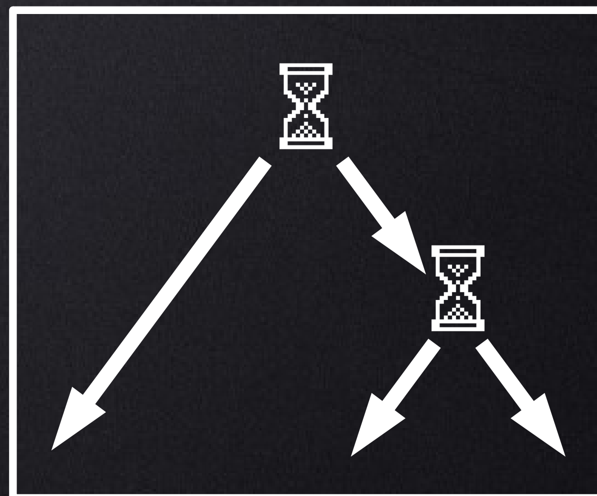
Child Thread

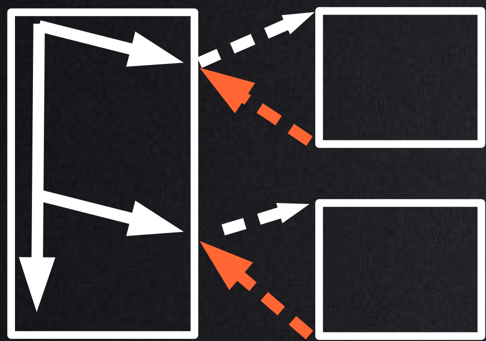




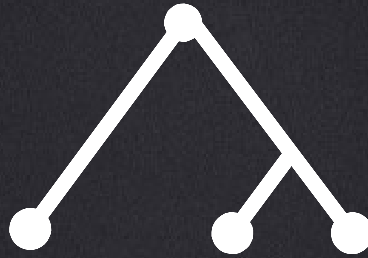
Child Thread





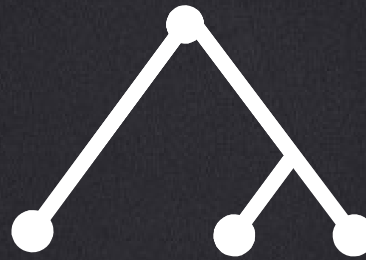


=

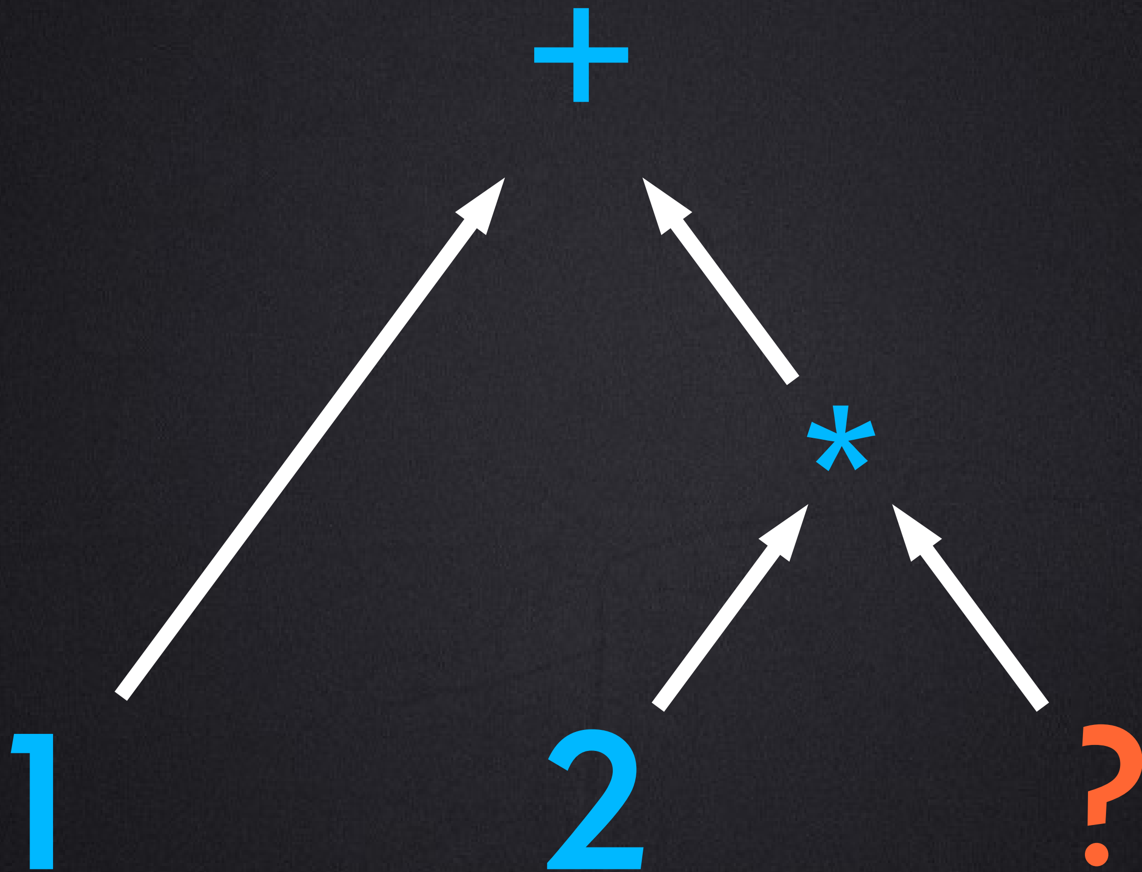


+ **time**

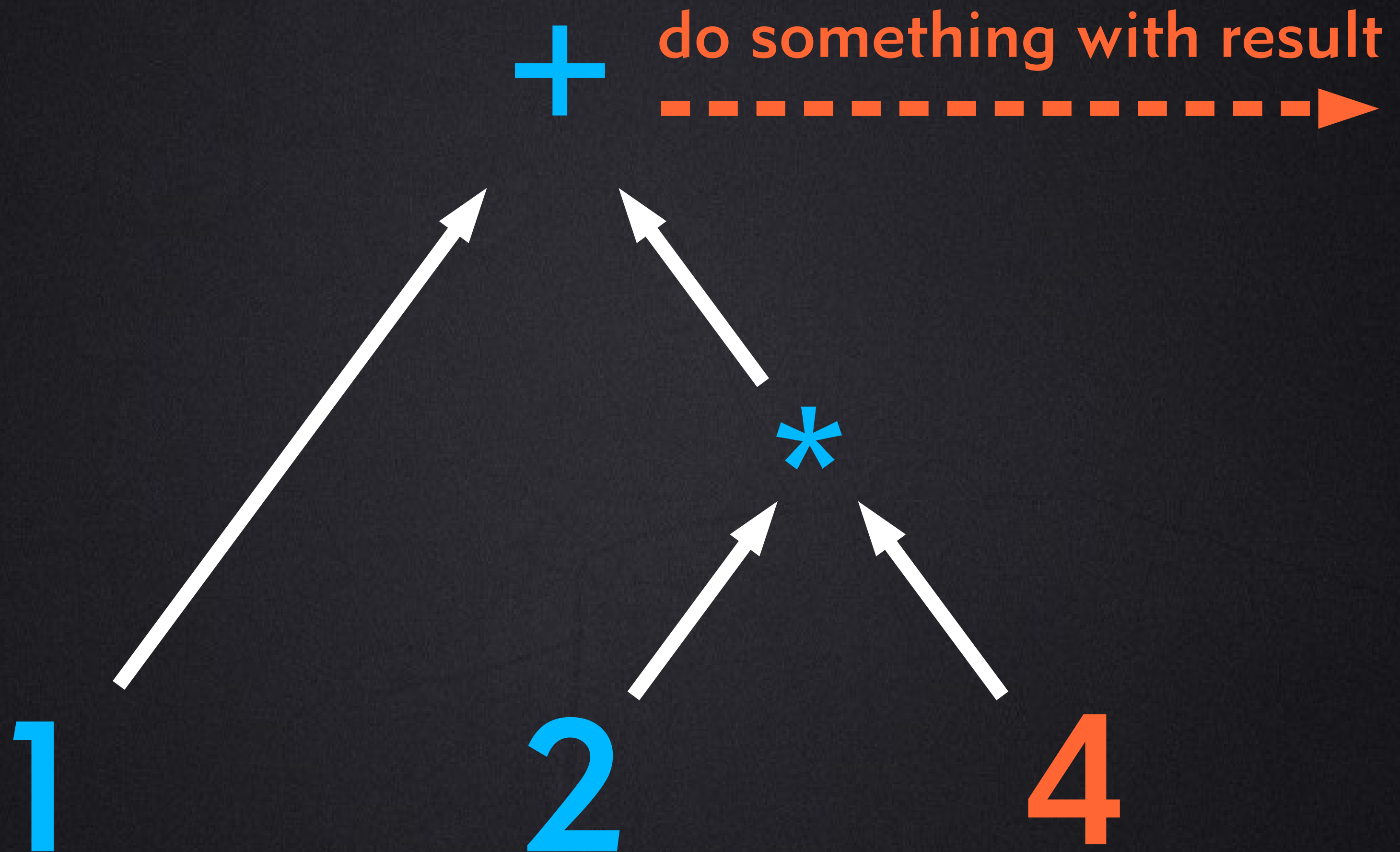
Functional
Reactive
Programming

=  + time

Reactive Evaluation



Reactive Evaluation



Reactive Evaluation

- Maintain dependency order of processes
- Evaluation triggered by availability of arguments
  - Instead of by invocation of parent
- Programs look/feel applicative
- We don't need threads (necessarily)

Browsers: bells, whistles, gotchas

- JavaScript event loop
  - Callbacks everywhere
- DOM (Document Object Model)
  - DOM elements contribute to application state, but are poor variables: they're mutable globals

see hlisp! https://github.com/tailrecursion/hlisp-starter

FRP: Objects in Play

- Event Streams
  - Sources of zero or more values over time
  - Incoming events trigger activity in whatever is listening
- Behaviors (a.k.a. Signals)
  - Always-valued boxes, `ref`-like
  - Value can be derived from a function applied to 1 or more “constituent” objects
  - Application of this function is triggered by activity in constituent objects

FRP: Fundamental Operations

- Event Streams
  - `filter`, `map`, `merge` etc.
  - `startsWith`: return a new Behavior backed by the stream, provided an initial value
- Behaviors
  - `lift`: return a new Behavior provided 1 or more other objects and a function
  - `changes`: return a new Event Stream carrying a Behavior's value over time

FRP with Flapjax: Example

HTML

```html
<body onload="demo.start()"
  <h3>Flapjax Demo</h3>
  <input type="text" id="n1" value="0"/>
  <input type="text" id="n2" value="0"/>
  <span  id="sum">0</span>
</body>
```

Browser

Flapjax Demo

0 + 0 = 0

Browser

Flapjax Demo

0 + 0 = 0

ClojureScript

```
(defn extractFloatE [id]
  (F/mapE parse-float (F/extractValueE id)))

(defn start []
  (let [n1  (extractFloatE "n1")
        n2  (extractFloatE "n2")
        sum (F/liftB + n1 n2)]
    (F/insertValueB sum "sum" "innerHTML")))
```

Browser

Flapjax Demo

0 + 0 = 0

ClojureScript

```
(defn extractFloatE [id]
  (F/mapE parse-float (F/extractValueE id)))

(defn start []
  (let [n1  (extractFloatE "n1")
        n2  (extractFloatE "n2")
        sum (F/liftB + n1 n2)]
    (F/insertValueB sum "sum" "innerHTML")))
```

Browser

Flapjax Demo

0 + 0 = 0

M-x ceremony-mode

```
(defn extractFloatE [id]
  (F/mapE parse-float (F/extractValueE id)))

(defn start []
  (let [n1  (extractFloatE "n1")
        n2  (extractFloatE "n2")
        sum (F/liftB + n1 n2)]
    (F/insertValueB sum "sum" "innerHTML")))
```

Browser

Flapjax Demo

0 + 0 = 0

ClojureScript

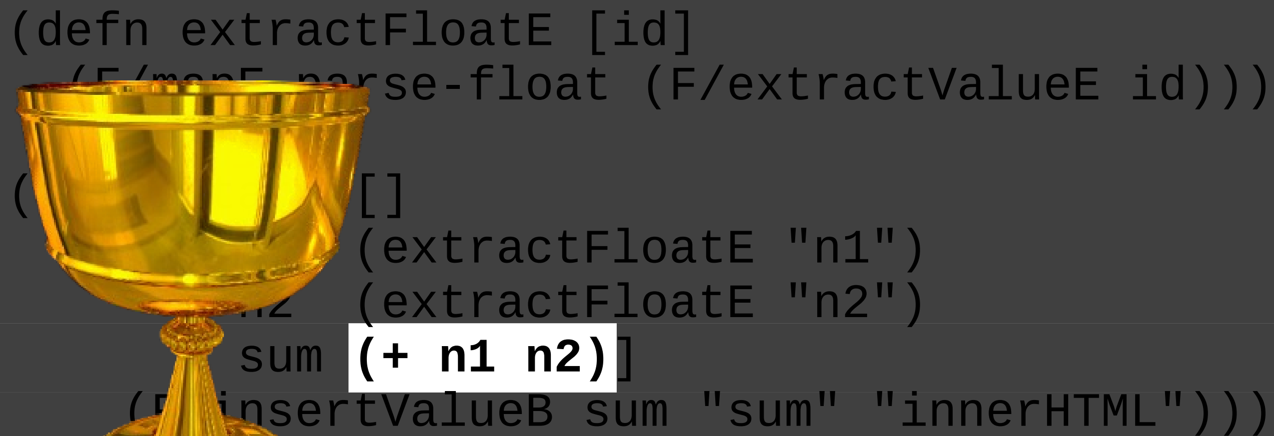
```
(defn extractFloatE [id]
  (F/mapE parse-float (F/extractValueE id)))

(defn start []
  (let [n1  (extractFloatE "n1")
        n2  (extractFloatE "n2")
        sum (F/liftB + n1 n2)]
    (F/insertValueB sum "sum" "innerHTML")))
```

Browser

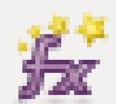
Flapjax Demo

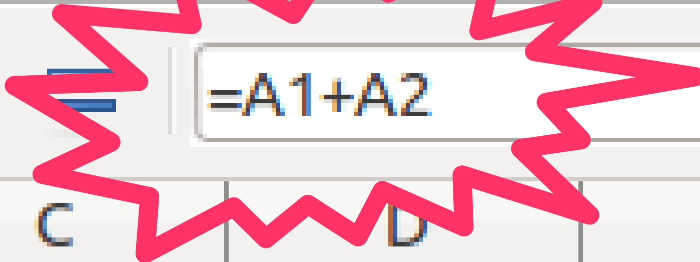
0 + 0 = 0

ClojureScript



A3





=A1+A2

omg

	A	B	C	D	E	F
1	0					
2	0					
3	0					
4						
5						
6						
7						
8						
9						
10						
11						
12						
13						
14						
15						
16						
17						
18						

Spreadsheets

- Input cells
  - user inserts values, evaluation propagates when new values are entered
- Formula cells
  - user defines, evaluated when constituent cells change value
- If spreadsheets are so awesome, why do we care about FRP?

Continuous vs. Discrete Propagation

- Spreadsheet propagation is **continuous**
  - Evaluation only happens if new values are introduced into the system
  - It's not possible to trigger evaluation without providing a new value
- FRP evaluation is continuous and/or **discrete**
  - `lift` can take Event Stream arguments
  - Event Streams trigger evaluation in dependents when any value is received, regardless of novelty

<opinion>

- FRP might be good for modeling I/O flows that are for side effects only (e.g. Rx Observables)
- FRP not awesome in ClojureScript
  - Application state spreads across the graph as intermediate, disparate Behaviors
  - Requires special control structures (`switchE`)
    - Implying Event Streams of Event Streams of Event Streams …
    - Hard to debug without static type system
  - Overlap between Behavior, Event Stream APIs
  - No integration point with Clojure state model

</opinion>

Javelin

Abstract spreadsheet library for reactive programming with values in ClojureScript.

Proudly delivered as a single macro, `cell`, that you write regular ClojureScript inside of.

ClojureScript

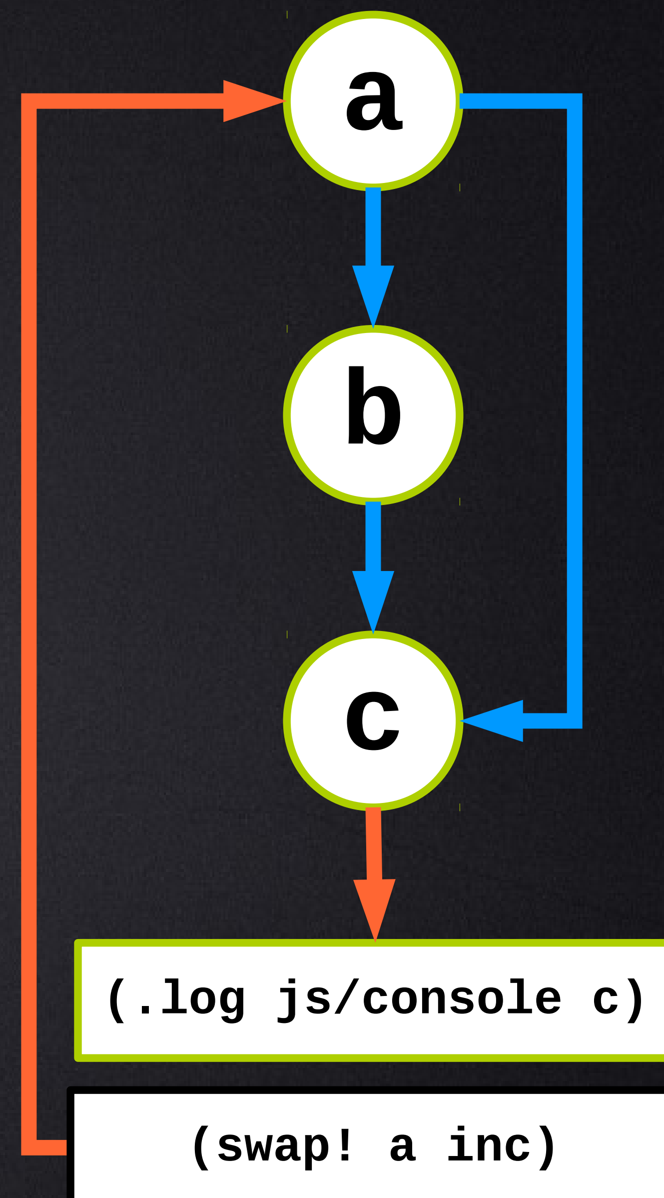
```
(defn start []
  (let [a (cell 0)
        b (cell (inc a))
        c (cell (+ 123 a b))]
    (cell (.log js/console c))
    (swap! a inc)
    (js/alert @b)))
```

input cell
formula cell
formula cell
anon. formula cell
mutation
dereference

ClojureScript

```
(let [a (cell 0)
      b (cell (inc a))
      c (cell (+ 123 a b))]
  (cell (.log js/console c))
  (swap! a inc))
```

Javelin guarantees that cell c sees only consistent values of a and b. This makes Javelin "glitch-free"



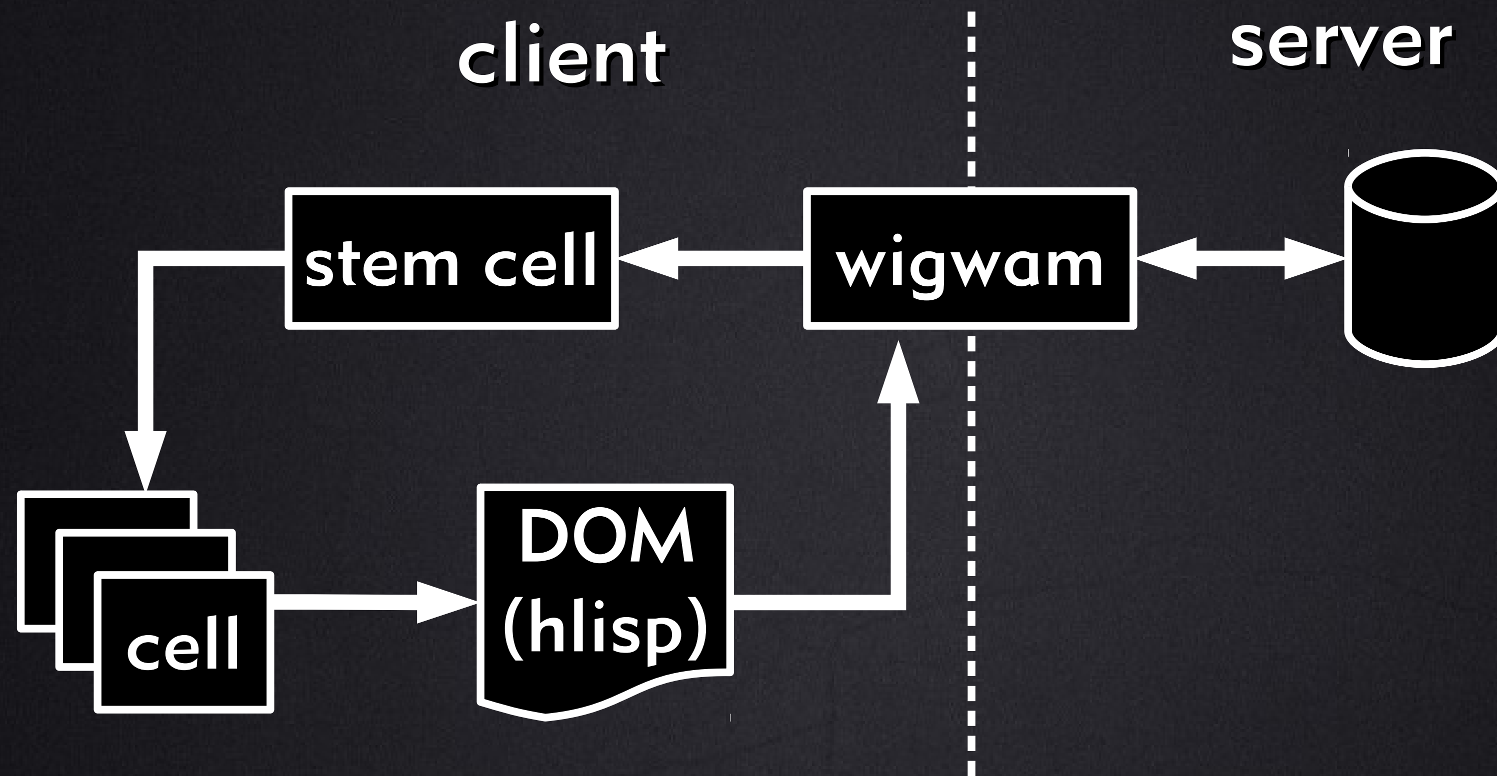
Javelin's Opinions

- At any point in time, a web application is in exactly one state
  - ...and that state is stored as a value in an input cell we call the "stem cell"
  - The stem cell is the root node of the dependency graph representing the application's behavior
- Everything the user sees or can do is governed by data in the stem cell
  - ...or derived formula cells

“Real” Javelin Apps

- We use 2 other things to build our applications:
  - hlisp: compiles HTML to ClojureScript
  - wigwam: server, client RPC machinery
- Available at https://github.com/tailrecursion

Javelin/hlisp/wigwam architecture



- DOM elements bound to cells (hlisp)
- RPC always return stem cell (wigwam)

Thank you!