my adventure with Elm

Yan Cui (@theburningmonk)

agenda

Hi, my name is Yan Cui.

gamesys

I'm **not** an expert on Elm.

elm

Learn Examples Libraries Community Blog

elm

A functional reactive language for interactive applications

Try Install

Functional

Features like immutability and type inference help you write code that is short, fast, and maintainable. Elm makes them easy to learn too.

Reactive

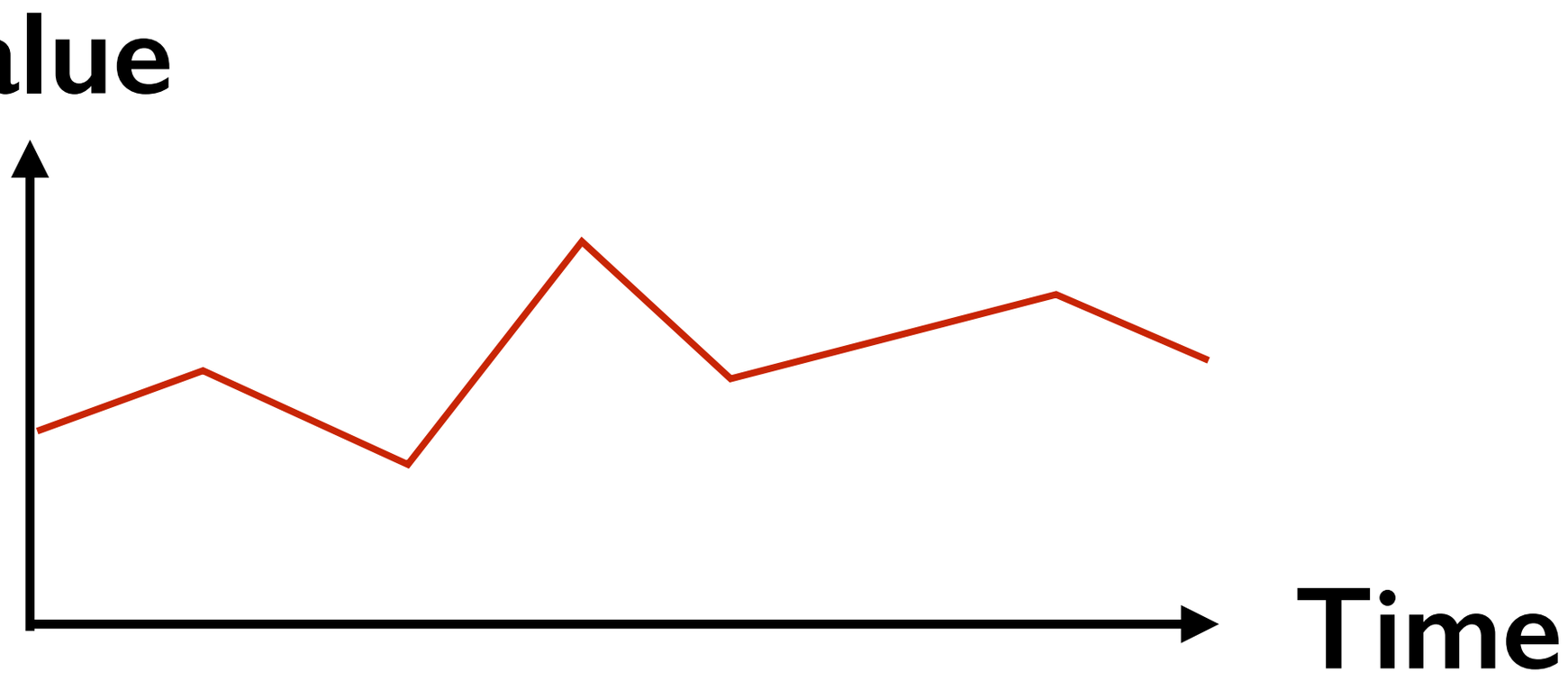
Elm is based on the idea of Functional Reactive Programming. Create highly interactive applications without messy callbacks or a tangle of shared state.

Compatible

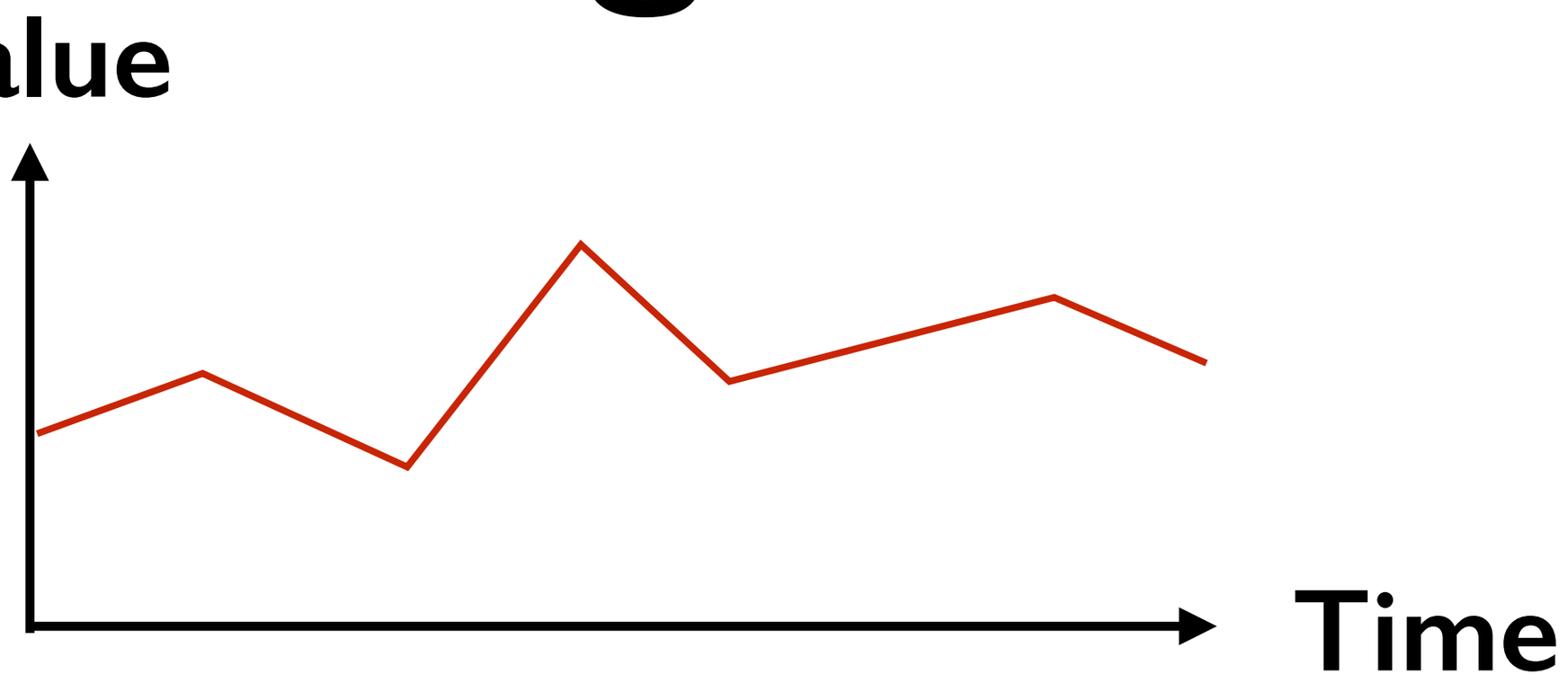
Elm compiles to HTML, CSS, and JavaScript. It is easy to use HTML and interop with JS, so it is simple to write part of your application in Elm.

Function Reactive Programming?

Value over Time

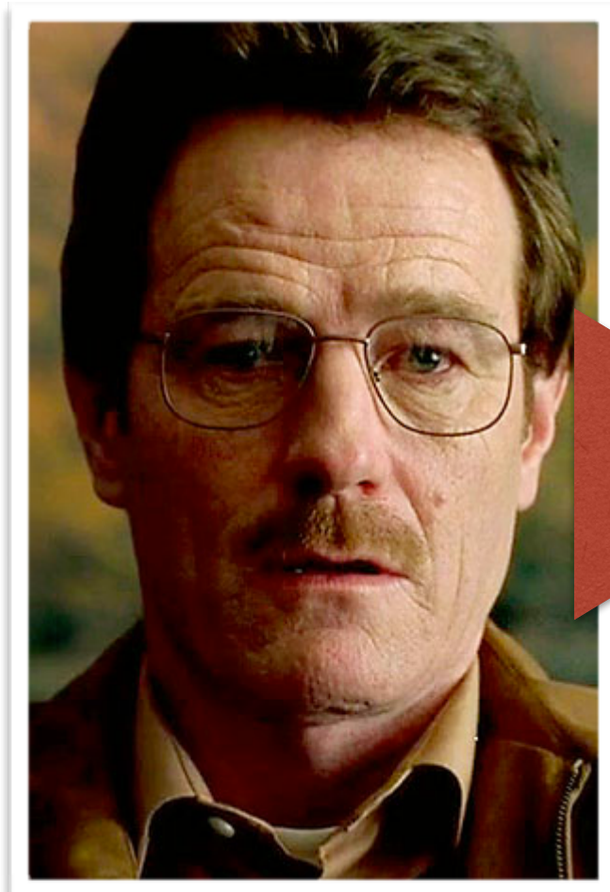

Value
Time

Signal

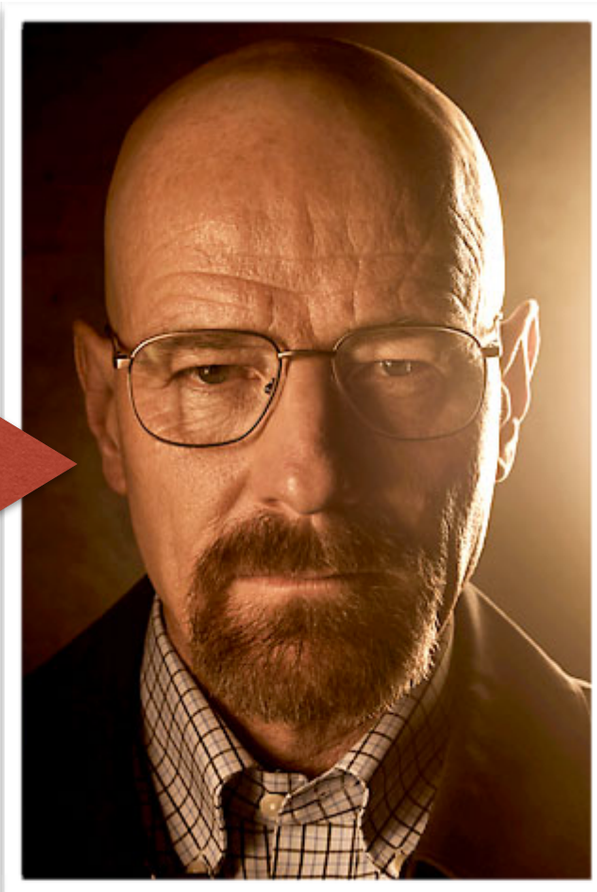



Canon
EOS
Rebel
XT


Canon
EOS
CANON ZOOM LENS
XT



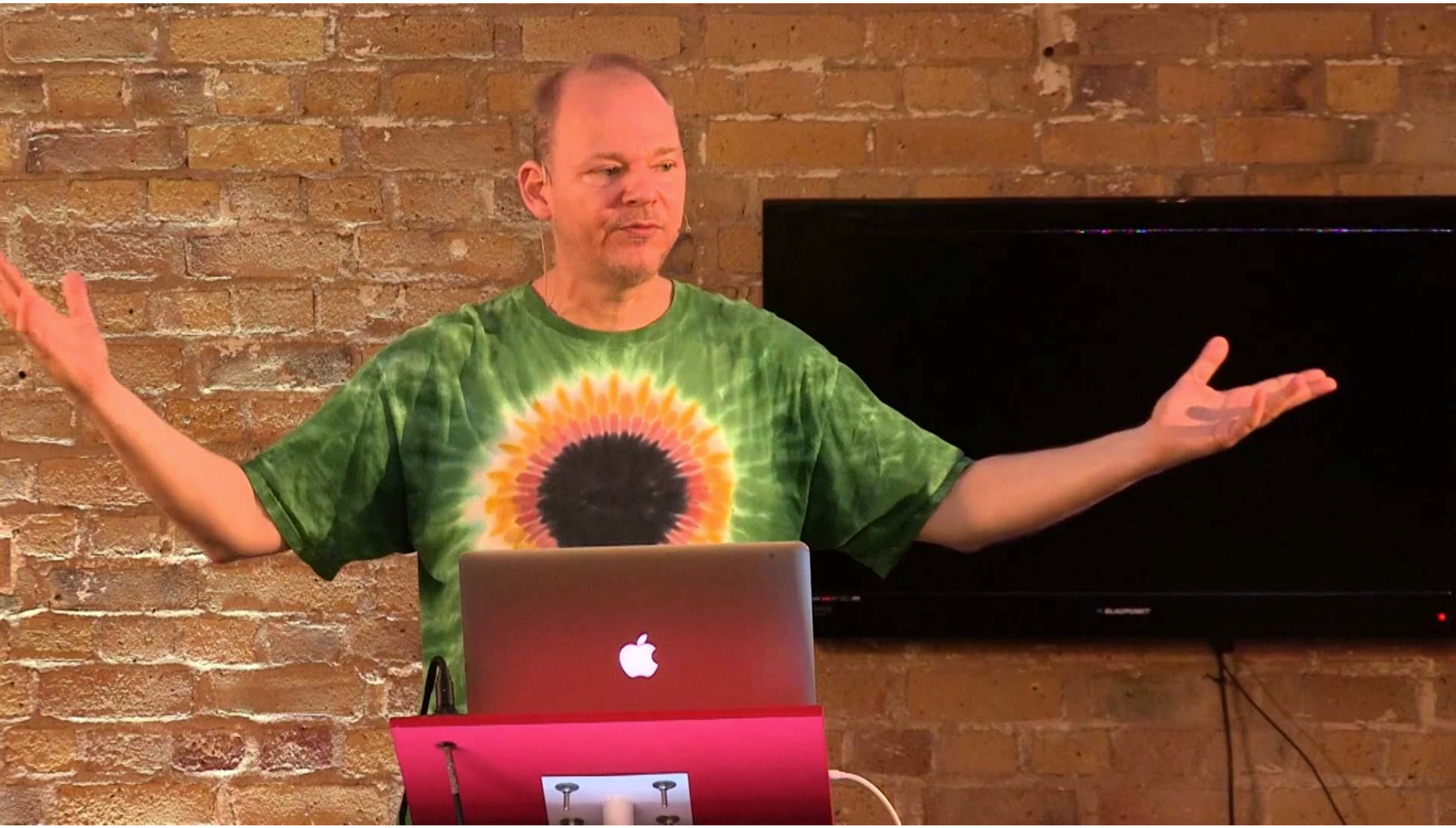
?




Breaking Bad
THE COMPLETE SERIES

Variable

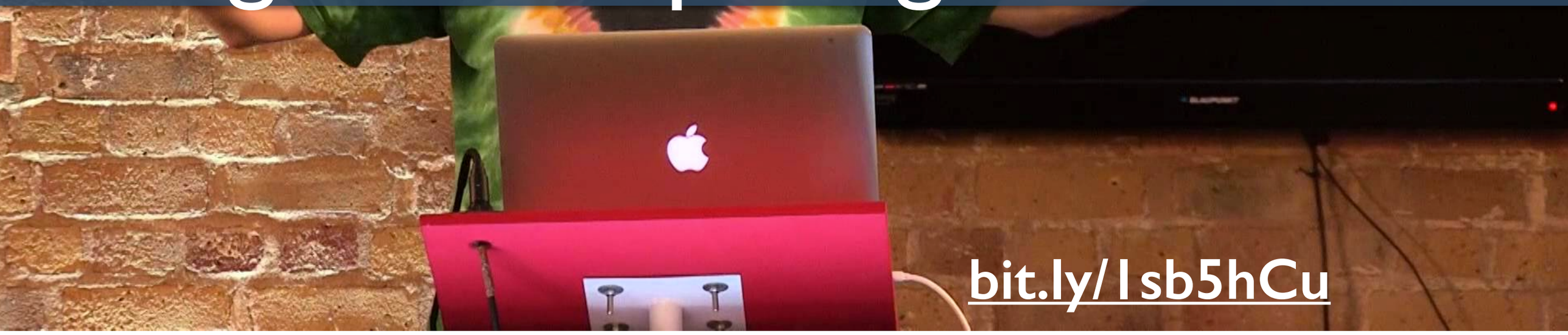
Signal




Java

Reactive is Dead,

long live composing side effects.



bit.ly/1sb5hCu

Reactive programming

From Wikipedia, the free encyclopedia

In computing, **reactive programming** is a programming paradigm oriented around data flows and the propagation of change. This means that it should be possible to express static or dynamic data flows with ease in the programming languages used, and that the underlying execution model will automatically propagate changes through the data flow.

For example, in an imperative programming setting, $a := b + c$ would mean that a is being assigned the result of $b + c$ in the instant the expression is evaluated. Later, the values of b and c can be changed with no effect on the value of a.

In reactive programming, the value of a would be automatically updated based on the new values.

A modern spreadsheet program is an example of reactive programming. Spreadsheet cells can contain literal values, or formulas such as "=B1+C1" that are evaluated based on other cells. Whenever the value of the other cells change, the value of the formula is automatically updated.

Another example is a hardware description language such as Verilog. In this case reactive programming allows us to model changes as they propagate through a circuit.

Reactive programming has foremost been proposed as a way to simplify the creation of interactive user interfaces, animations in real time systems, but is essentially a general programming paradigm.

For example, in a Model-view-controller architecture, reactive programming can allow changes in the underlying model to automatically be reflected in the view, and vice versa.[1]

Reactive programming

From Wikipedia, the free encyclopedia

In computing, **reactive programming** is a programming paradigm oriented around data flows and the propagation of change. This means that it should be possible to express static or dynamic data flows with ease in the programming languages used, and that the underlying execution model will automatically propagate changes through the data flow.

For example, in a Model-view-controller architecture, reactive programming can allow changes in the underlying model to automatically be reflected in the view, and vice versa.[1]

The Reactive Manifesto

Published on September 16 2014. (v2.0)

Organisations working in disparate domains are independently discovering patterns for building software that look the same. These systems are more robust, more resilient, more flexible and better positioned to meet modern demands.

These changes are happening because application requirements have changed dramatically in recent years. Only a few years ago a large application had tens of servers, seconds of response time, hours of offline maintenance and gigabytes of data. Today applications are deployed on everything from mobile devices to cloud-based clusters running thousands of multi-core processors. Users expect millisecond response times and 100% uptime. Data is measured in Petabytes. Today's demands are simply not met by yesterday's software architectures.

We believe that a coherent approach to systems architecture is needed, and we believe that all necessary aspects are already recognised individually: we want systems that are Responsive, Resilient, Elastic and Message Driven. We call these Reactive Systems.

Systems built as Reactive Systems are more flexible, loosely-coupled and scalable. This makes them easier to develop and amenable to change. They are significantly more tolerant of failure and when failure does occur they meet it with elegance rather than disaster. Reactive Systems are highly responsive,

Containment

Scalable

Location-Transparency

Elasticity

Loose-Coupling

Asynchronous

High-Availability

Non-Blocking

Replication

Resilient

Message-Passing

Isolation

The Reactive Manifesto

Published on

Organisations working [illegible] independently discovering
[illegible] that lo[illegible]e same. These systems are more
[illegible]xibl[illegible]positioned to me[illegible]

These changes are happening because ap[illegible]
dramatically in recent years. Only a few years [illegible] had tens
of servers, seconds of response time, hours of offlin[illegible]
gigabytes of [illegible]ata is n[illegible]
[illegible]s software
are[illegible]

We beli[illegible] coherent ar[illegible]ystems archite[illegible]
[illegible]cessary a[illegible] recognised
[illegible]esponsive, [illegible]d Mess[illegible]
[illegible]e Systems.

Systems built as Reactive Systems are mo[illegible]ely-coupled an[illegible]
scalable. This makes them easier to develop a[illegible]able to change. T[illegible]
significantly more tolerant of failure and when fa[illegible]re does occur they meet it
with elegance rather than disaster. Reactive Systems are highly responsive,

Reactive programming

From Wikipedia, the free encyclopedia

In computing, **reactive programming** is a programming paradigm oriented around data flows and the propagation of change. This means that it should be possible to express static or dynamic data flows with ease in the programming languages used, and that the underlying execution model will automatic

For example, in an imperative programming setting, $a := b + c$ would ... of $b + c$ in the instant the expression is evaluated. Later, the values of b and c can ... ue of a.

In reactive programming, the value of a would be automatically up ... d bas

Reactive Programming =

... ral values, or formulas such as "=B1+C1" that ... evaluated based on ... her cells. W ... value of the formula is automatically updated.

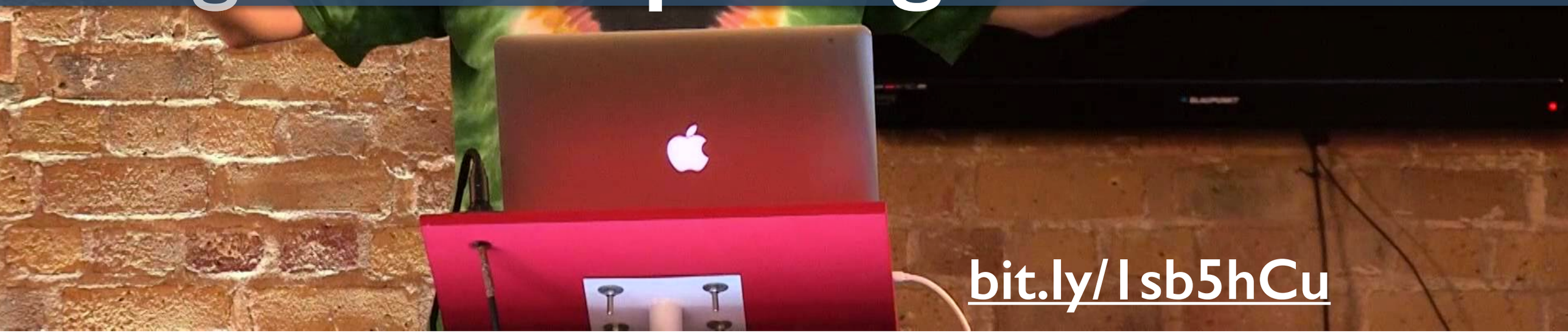
Another example is a hardware description language such as ... changes as they propagate through a circuit.

Reactive programming has foremost been proposed as a way to si ... es, animations in real time systems, but is essentially a general programming paradigm.

For example, in a Model-view-controller architecture, reactive programming can allow changes in the underlying model to automatically be reflected in the view, and vice versa.[1]

Reactive is Dead,

long live **composing side effects.**



bit.ly/1sb5hCu

"One thing I'm discovering is that **transforming data** is **easier** to think about than maintaining state."

- Dave Thomas

Imperative

x.f()

Functional

let y = f(x)

Imperative

x.f()

mutation

Functional

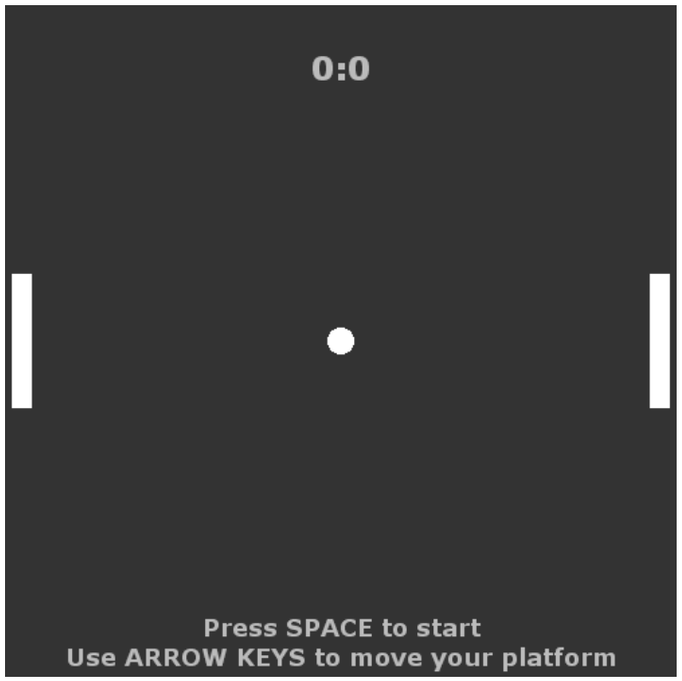
let y = f(x)

Short-term memory

From Wikipedia, the free encyclopedia

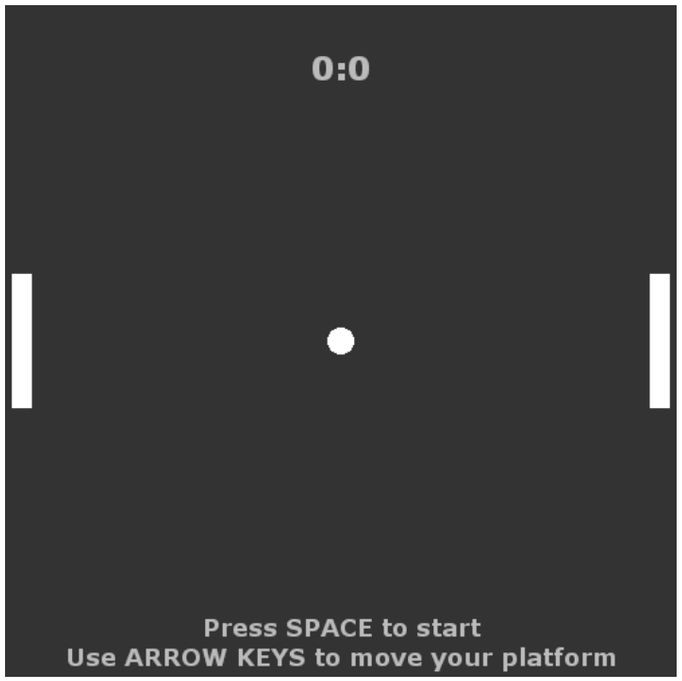
Short-term memory (or "primary" or "active memory") is the capacity for holding a small amount of information in mind in an active, readily available state for a short period of time. The duration of short-term memory (when rehearsal or active maintenance is prevented) is believed to be in the order of seconds. A commonly cited capacity is 7 ± 2 elements. In contrast, long-term memory can hold an indefinite amount of information.

Short-term memory should be distinguished from working memory, which refers to structures and processes used for temporarily storing and manipulating information (see details below).

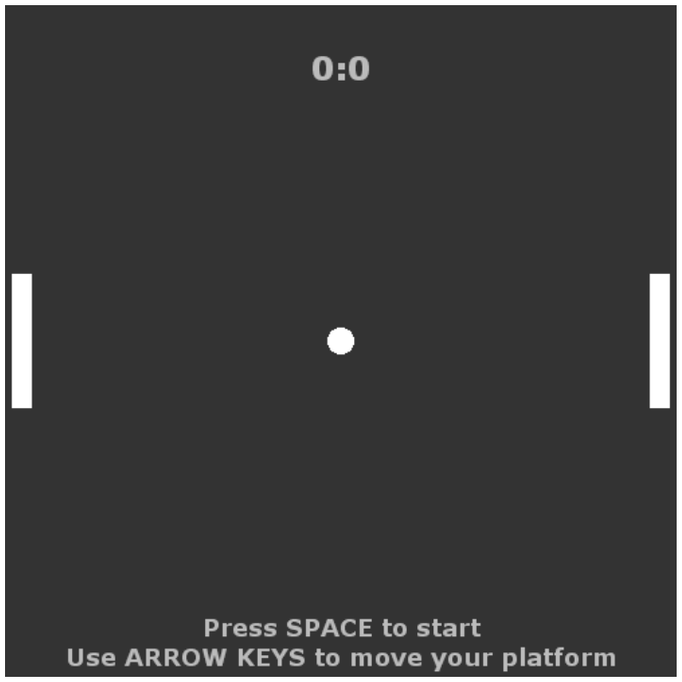


Move Up

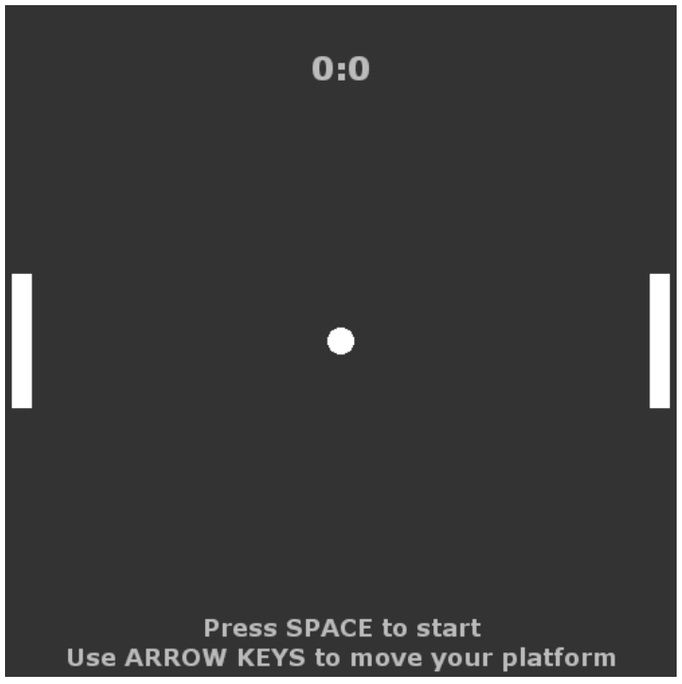
Move Down



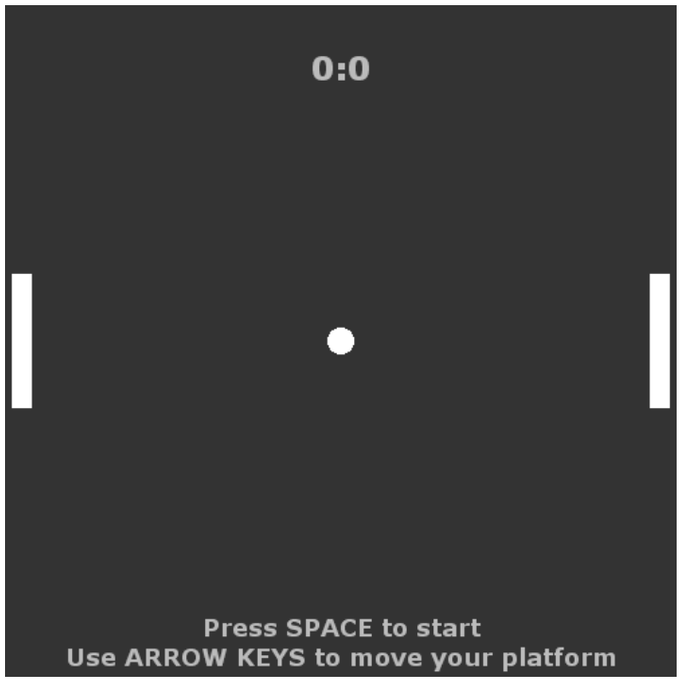

```
private var arrowKeyUp:Bool;
private var arrowKeyDown:Bool;

private var platform1:Platform;
private var platform2:Platform;
private var ball:Ball;
```



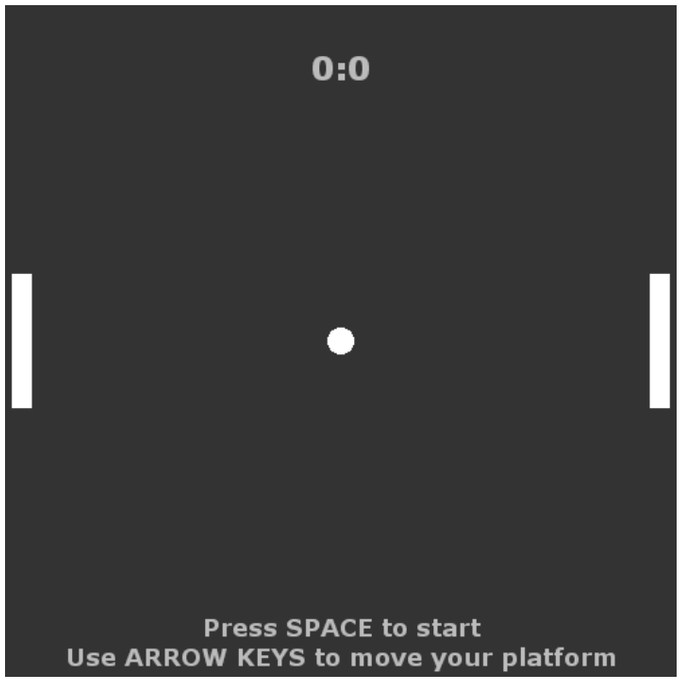

```
function keyDown(event:KeyboardEvent):Void {
    if (currentGameState == Paused &&
        event.keyCode == 32) {
        setGameState(Playing);
    } else if (event.keyCode == 38) {
        arrowKeyUp = true;
    } else if (event.keyCode == 40) {
        arrowKeyDown = true;
    }
}
```




```
function keyUp(event:KeyboardEvent):Void {
    if (event.keyCode == 38) {
        arrowKeyUp = false;
    } else if (event.keyCode == 40) {
        arrowKeyDown = false;
    }
}
```

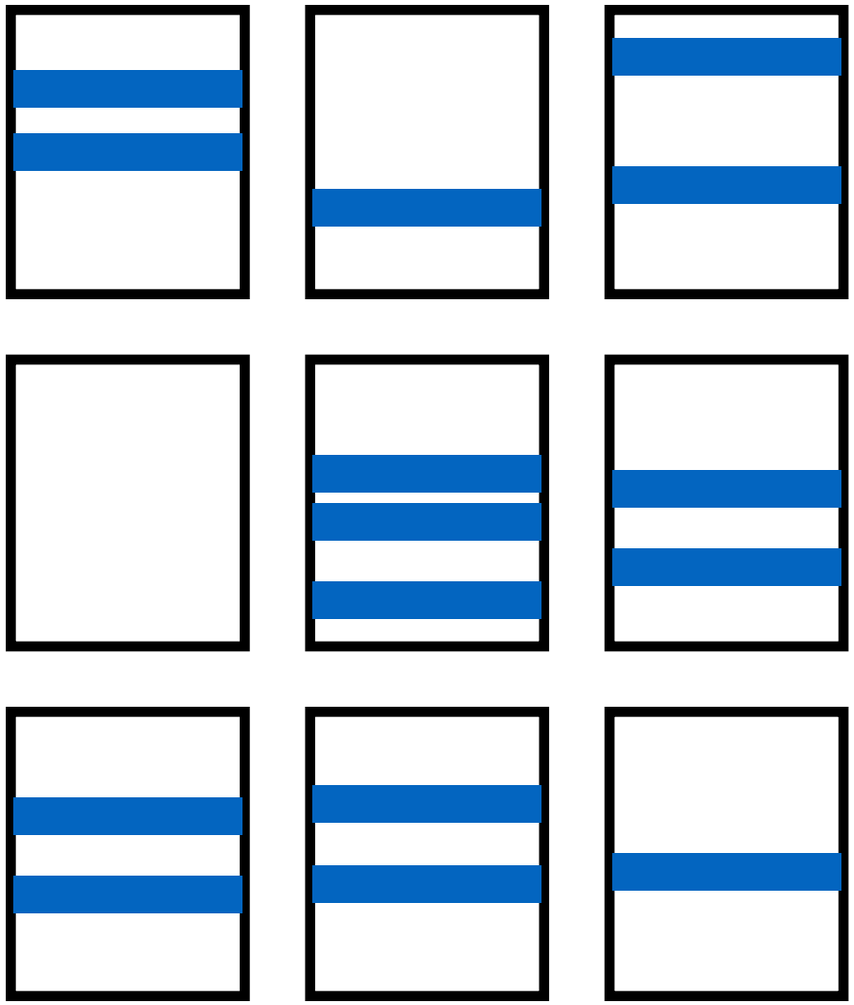



```
function everyFrame(event:Event):Void {
    if(currentGameState == Playing){
        if (arrowKeyUp) {
            platform1.y -= platformSpeed;
        }
        if (arrowKeyDown) {
            platform1.y += platformSpeed;
        }
        if (platform1.y < 5) platform1.y = 5;
        if (platform1.y > 395) platform1.y = 395;
    }
}
```



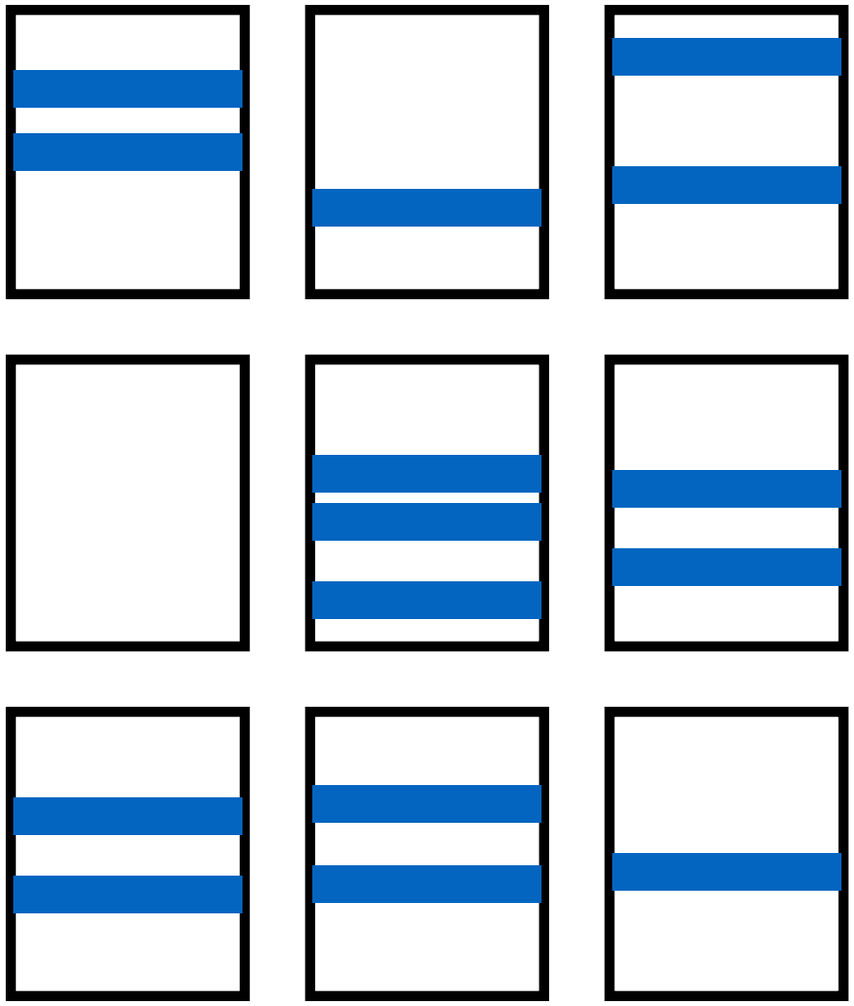

```
function everyFrame(event:Event):Void {
    if(currentGameState == Playing){
        if (arrowKeyUp) {
            platform1.y -= platformSpeed;
        }
        if (arrowKeyDown) {
            platform1.y += platformSpeed;
        }
        if (platform1.y < 5) platform1.y = 5;
        if (platform1.y > 395) platform1.y = 395;
    }
}
```

source files

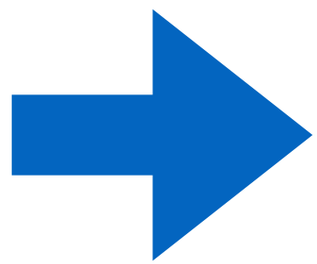


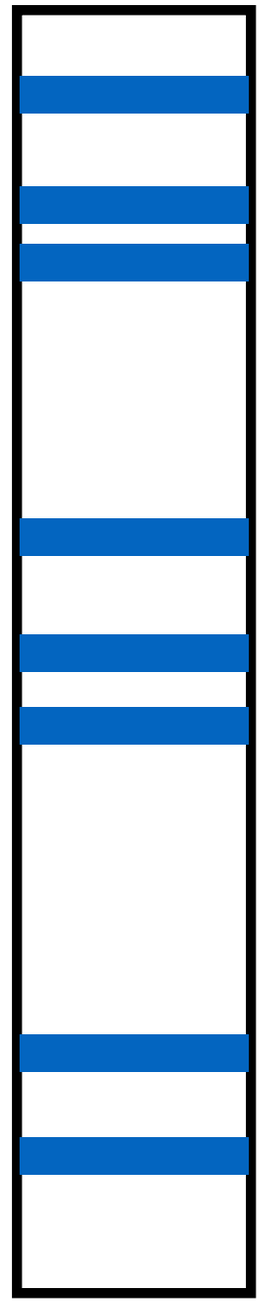
state changes

source files

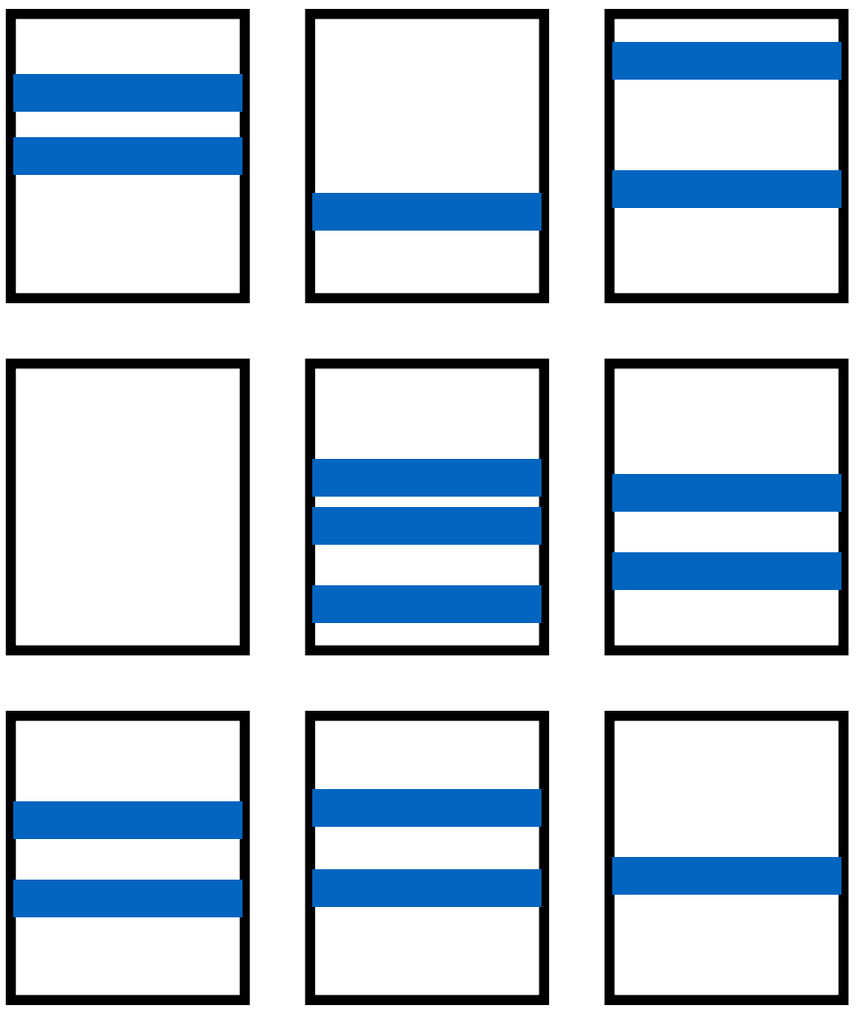


execution

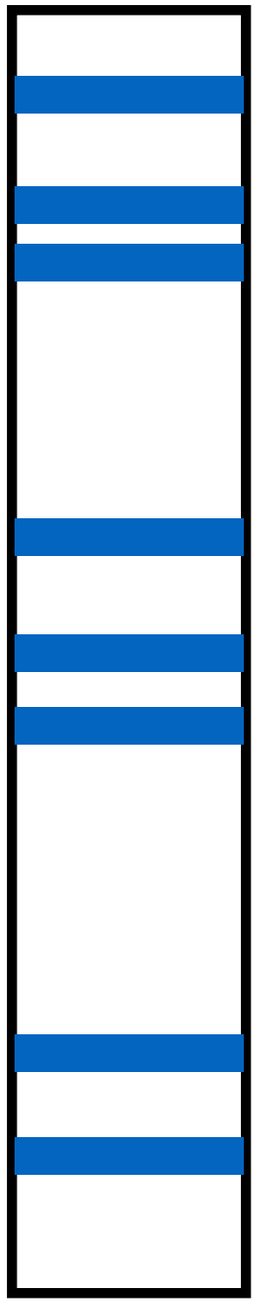


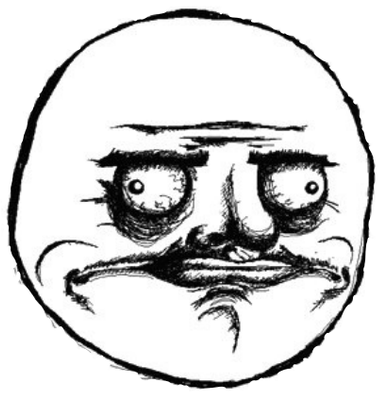


source files



execution





mental model

input	state	new state	behaviour
	{ x; y }	{ x; y-speed }	
	{ x; y }	{ x; y+speed }	
timer	{ x; y }	{ x; y }	draw platform
…	…	…	…

Imperative

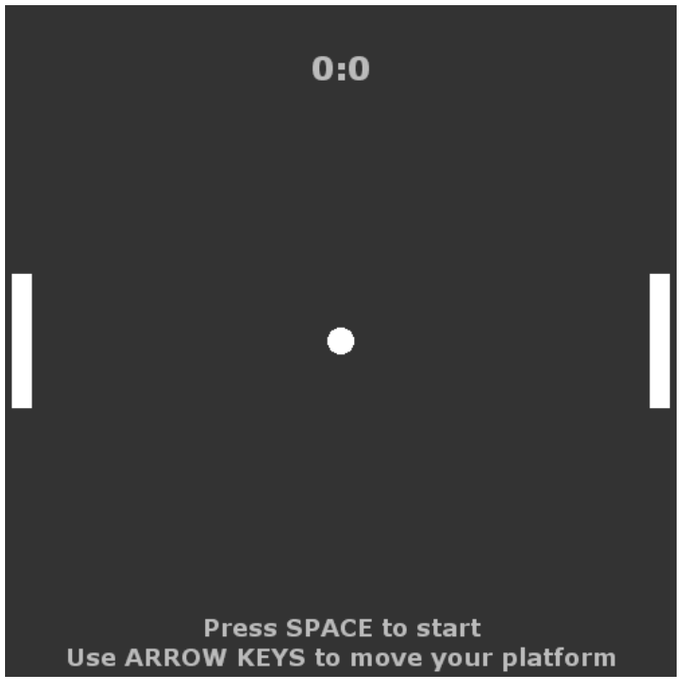
```
x.f()
```

Functional

```
let y = f(x)
```

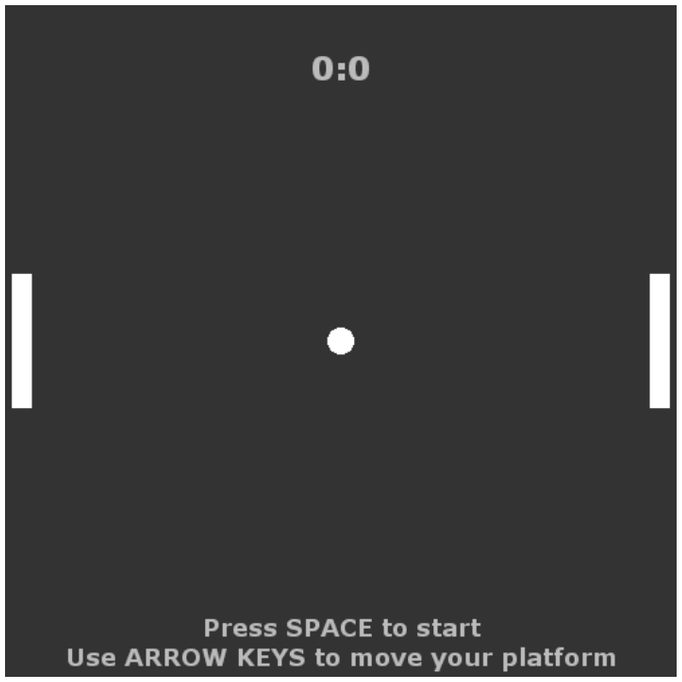
transformation

Transformations simplify problem decomposition

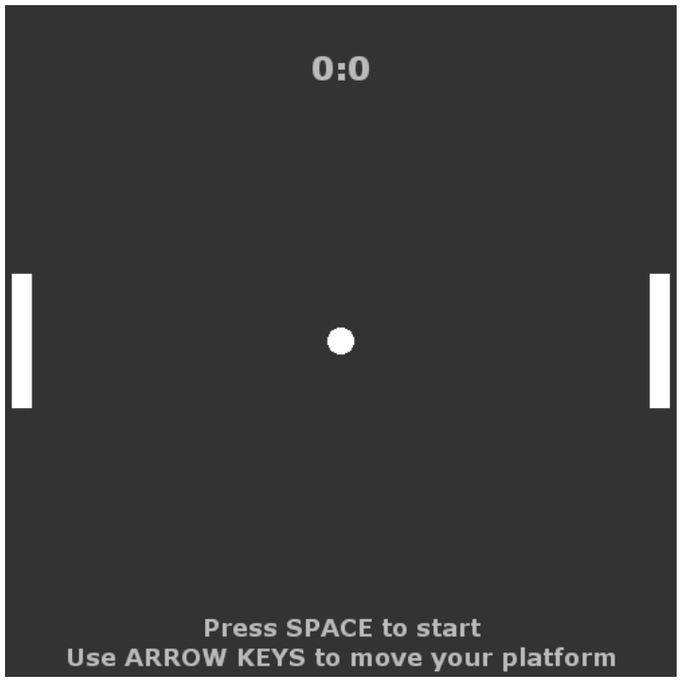


Move Up

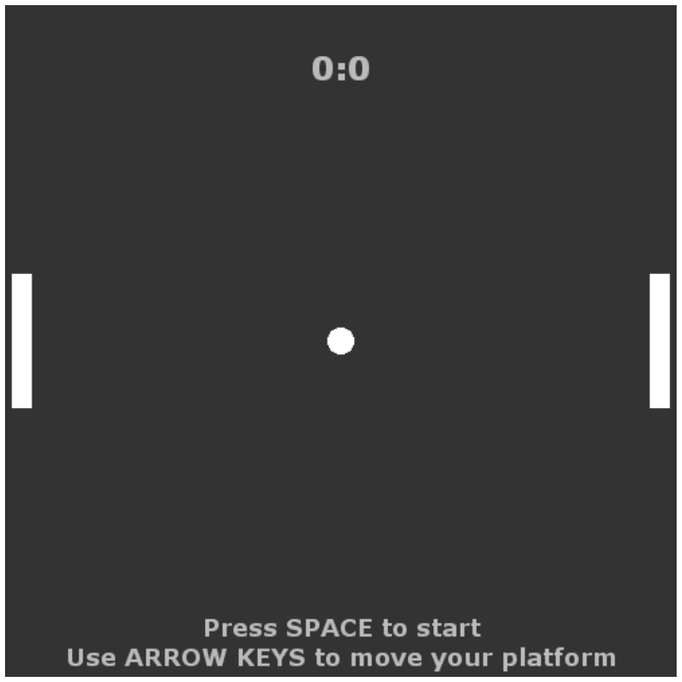
Move Down



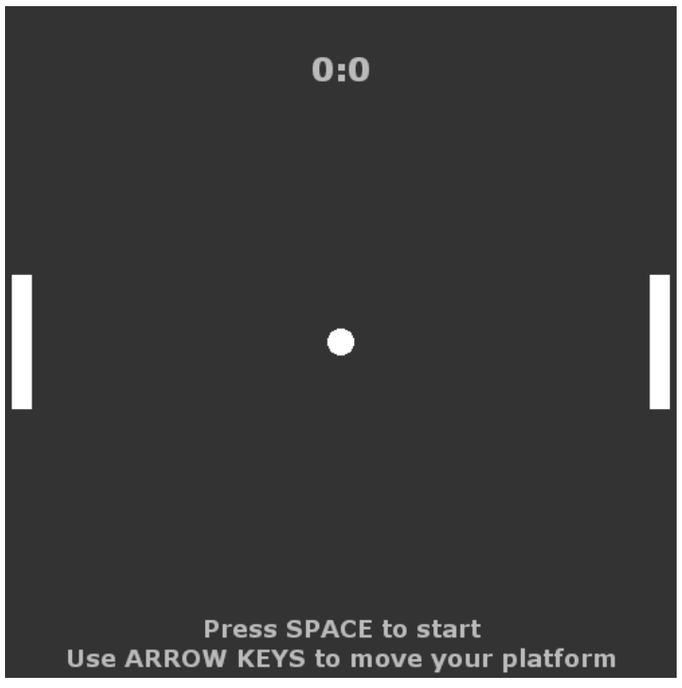

```
type alias Platform = {x:Int, y:Int}
defaultPlatform = {x=5, y=0}

delta = Time.fps 20
input = Signal.sampleOn delta Keyboard.arrows

cap x = max 5 <| min x 395

p1 : Signal Platform
p1 = foldp (\{x, y} s -> {s | y <- cap <| s.y + 5*y})
                  defaultPlatform
                  input
```



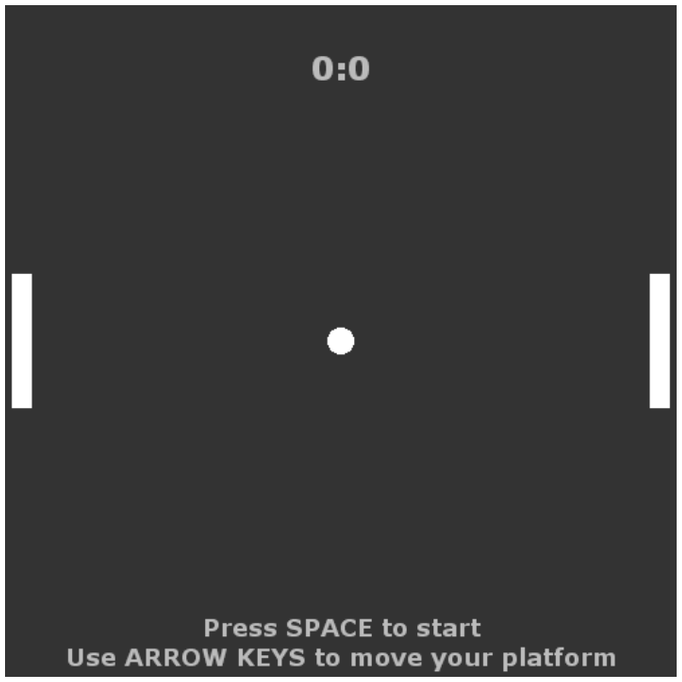

```
type alias Platform = {x:Int, y:Int}
defaultPlatform = {x=5, y=0}

delta = Time.fps 20
input = Signal.sampleOn delta Keyboard.arrows

cap x = max 5 <| min x 395

p1 : Signal Platform
p1 = foldp (\{x, y} s -> {s | y <- cap <| s.y + 5*y})
           defaultPlatform
           input
```



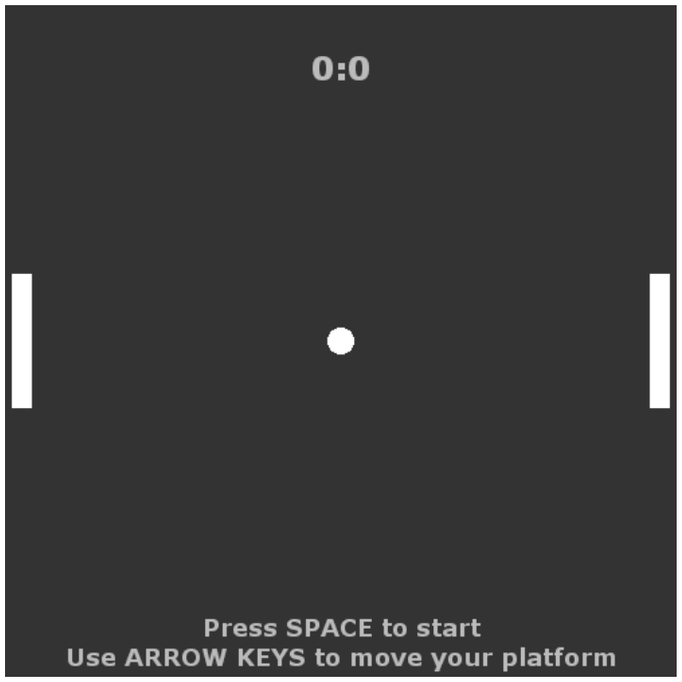

```
type alias Platform = {x:Int, y:Int}
defaultPlatform = {x=5, y=0}

delta = Time.fps 20
input = Signal.sampleOn delta Keyboard.arrows

cap x = max 5 <| min x 395

p1 : Signal Platform
p1 = foldp (\{x, y} s -> {s | y <- cap <| s.y + 5*y})
           defaultPlatform
           input
```



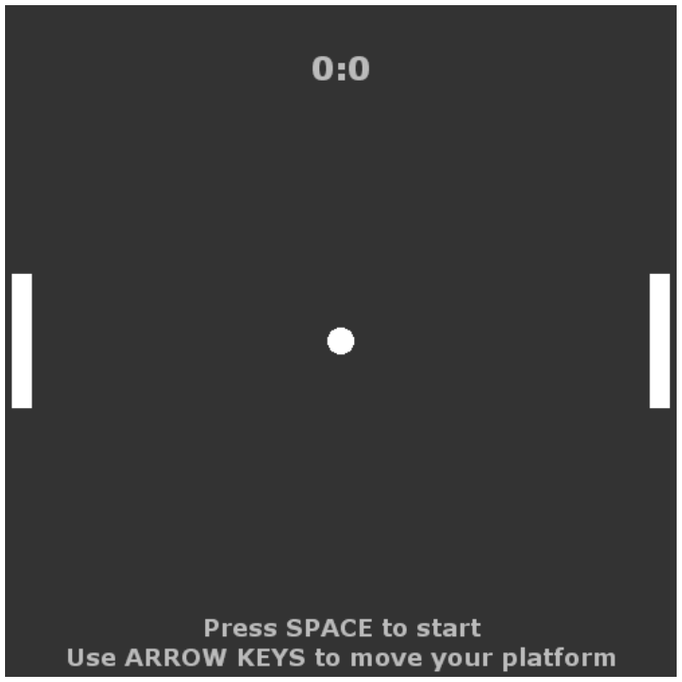
UP { x=0, y=1 }
DOWN { x=0, y=-1 }
LEFT { x=-1, y=0 }
RIGHT { x=1, y=0 }



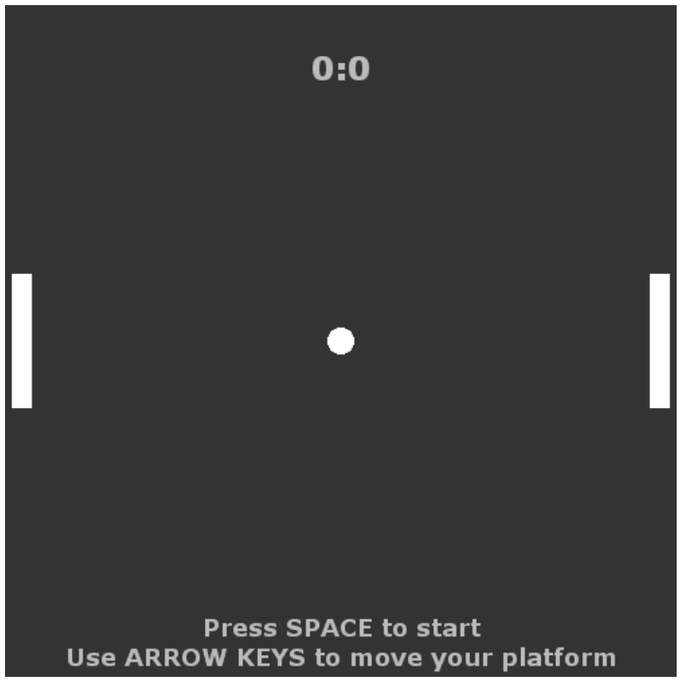

```
type alias Platform = {x:Int, y:Int}
defaultPlatform = {x=5, y=0}

delta = Time.fps 20
input = Signal.sampleOn delta Keyboard.arrows

cap x = max 5 <| min x 395

p1 : Signal Platform
p1 = foldp (\{x, y} s -> {s | y <- cap <| s.y + 5*y})
                  defaultPlatform
                  input
```



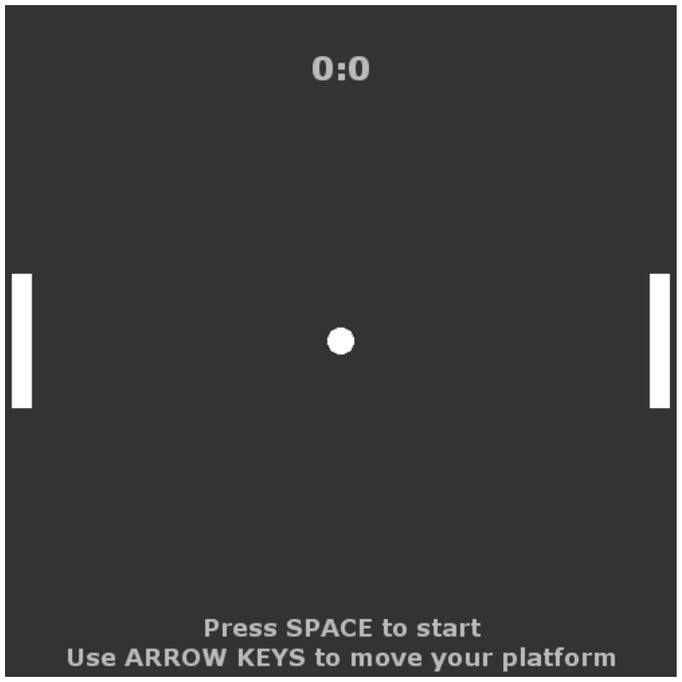

```
type alias Platform = {x:Int, y:Int}
defaultPlatform = {x=5, y=0}

delta = Time.fps 20
input = Signal.sampleOn delta Keyboard.arrows

cap x = max 5 <| min x 395

p1 : Signal Platform
p1 = foldp (\{x, y} s -> {s | y <- cap <| s.y + 5*y})
           defaultPlatform
           input
```




```
type alias Platform = {x:Int, y:Int}
defaultPlatform = {x=5, y=0}

delta = Time.fps 20
input = Signal.sampleOn delta Keyboard.arrows

cap x = max 5 <| min x 395

p1 : Signal Platform
p1 = foldp (\{x, y} s -> {s | y <- cap <| s.y + 5*y})
                  defaultPlatform
                  input
```




```
type alias Platform = {x:Int, y:Int}
defaultPlatform = {x=5, y=0}

delta = Time.fps 20
input = Signal.sampleOn delta Keyboard.arrows

cap x = max 5 <| min x 395

p1 : Signal Platform
p1 = foldp (\{x, y} s -> {s | y <- cap <| s.y + 5*y})
           defaultPlatform
           input
```




```
type alias Platform = {x:Int, y:Int}
defaultPlatform = {x=5, y=0}

delta = Time.fps 20
input = Signal.sampleOn delta Keyboard.arrows

cap x = max 5 <| min x 395

p1 : Signal Platform
p1 = foldp (\{x, y} s -> {s | y <- cap <| s.y + 5*y})
                  defaultPlatform
                  input
```

Rx = Dart = Elm

Observable Stream Signal

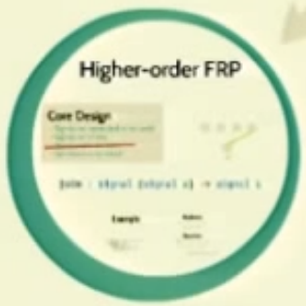
strangeloop

Sept 17-19, 2014 - St. Louis, MO
http://thestrangeloop.com

Static Graphs

Dynamic Graphs

Pursuing Reconfigurable Graphs

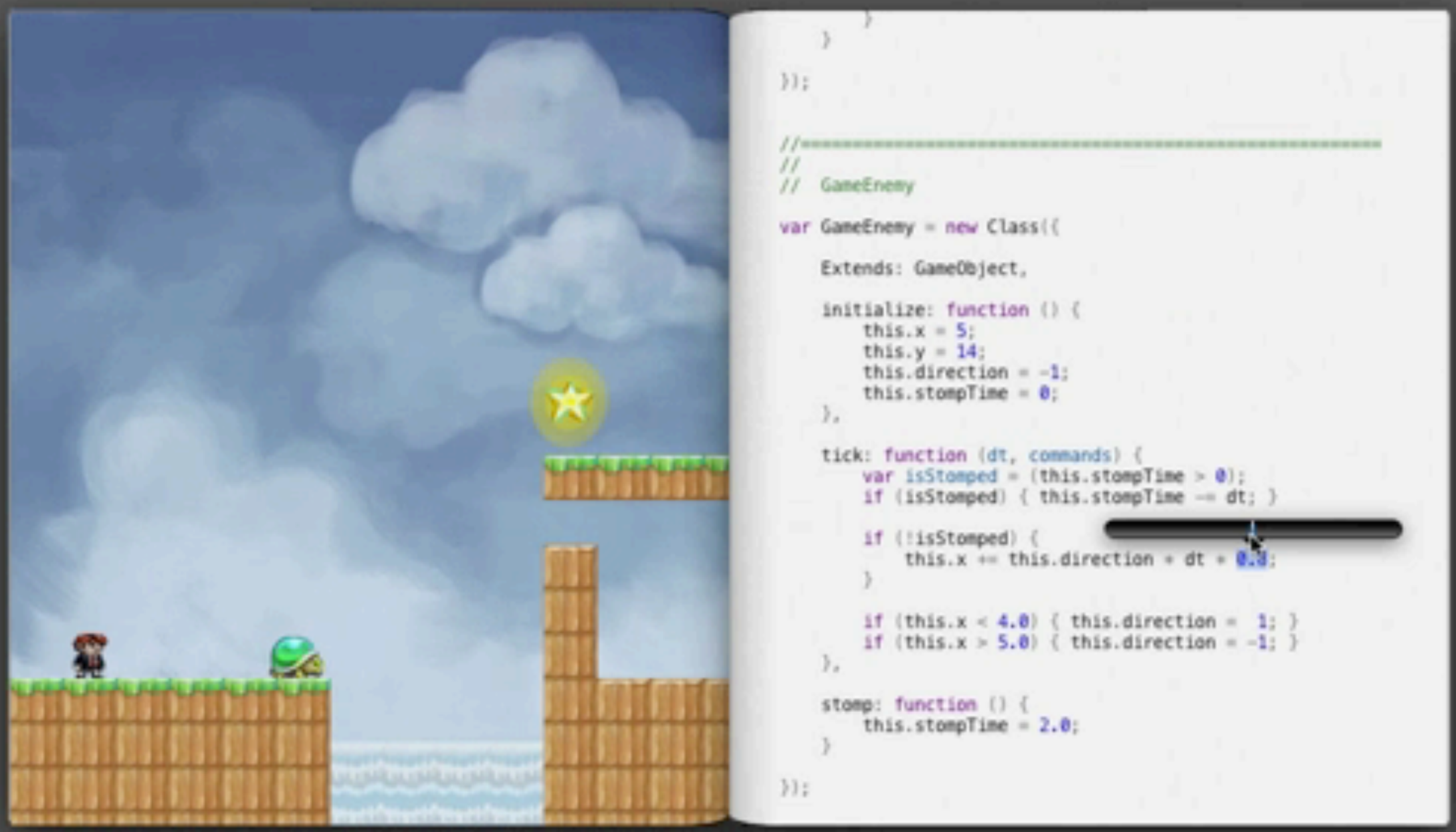


Infinite Signals

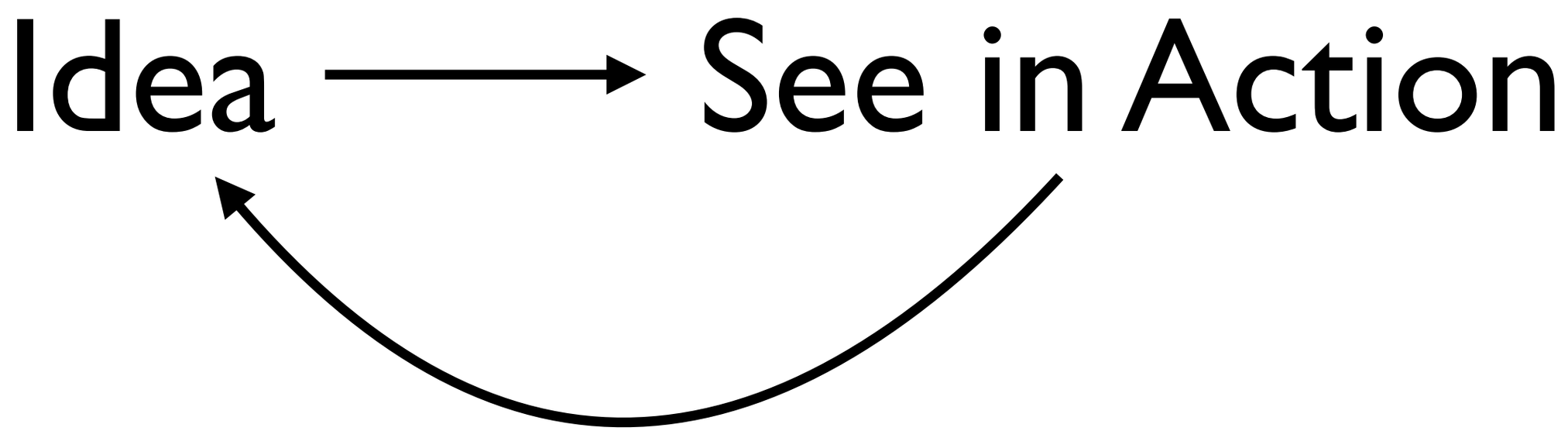
Finite Event Streams

Neither!


DELTA



```
    }
});

//============================================================
//
//  GameEnemy

var GameEnemy = new Class({

    Extends: GameObject,

    initialize: function () {
        this.x = 5;
        this.y = 14;
        this.direction = -1;
        this.stompTime = 0;
    },

    tick: function (dt, commands) {
        var isStomped = (this.stompTime > 0);
        if (isStomped) { this.stompTime -= dt; }

        if (!isStomped) {
            this.x += this.direction * dt * 0.3;
        }

        if (this.x < 4.0) { this.direction =  1; }
        if (this.x > 5.0) { this.direction = -1; }
    },

    stomp: function () {
        this.stompTime = 2.0;
    }

});
```


Idea
See in Action

KHANACADEMY | Subject: Computer pro... | About | Donate | Search for subjects, skills, an | Log in | Unclaimed points

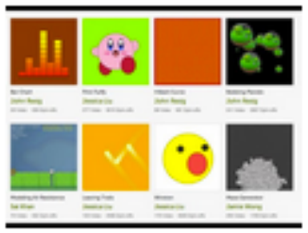
Computer programming

Learn the fundamentals of programming with the popular JavaScript language and ProcessingJS library. Write your own programs and share them, explore programs made by others, and learn from each other's programs!

» Learn Programming

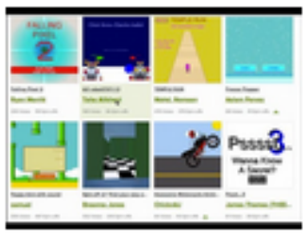
+ Create Program

Documentation



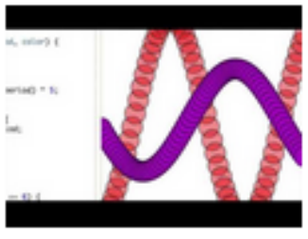
Intro to JS: Drawing & Animation

In these tutorials, you'll learn how to use the JavaScript language and the ProcessingJS library to create fun drawings and animations. If you've never programmed before, start here to learn how!



Advanced JS: Games & Visualizations

Now that you know how to program in JavaScript and make basic drawings and animations, how could you use that knowledge to make games and visualizations?



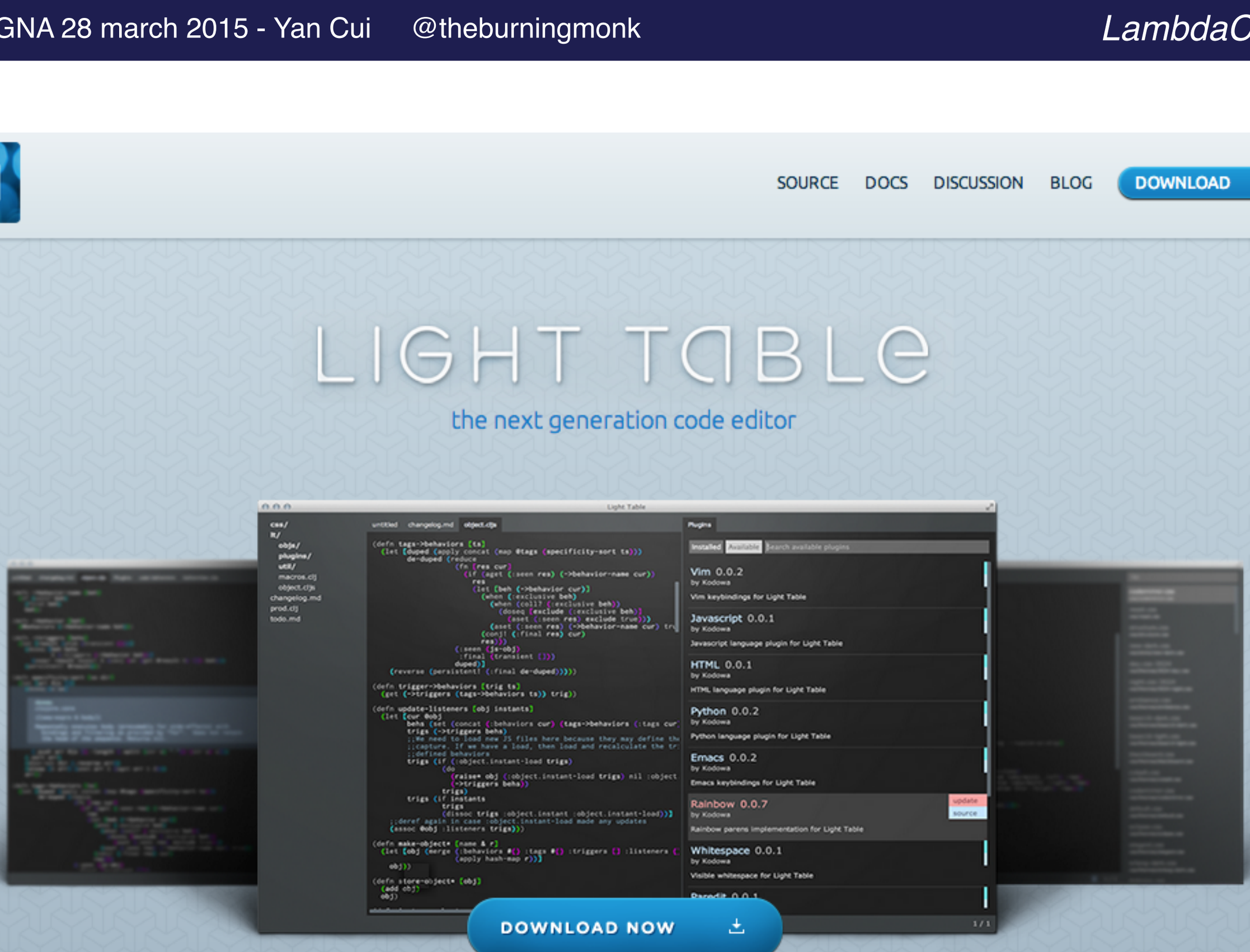
Advanced JS: Natural Simulations

Learn how to use JavaScript, ProcessingJS, and mathematical concepts to simulate nature in your programs. These tutorials

SOURCE DOCS DISCUSSION BLOG DOWNLOAD

LIGHT TABLE

the next generation code editor



DOWNLOAD NOW

Elm Debugger: Mario

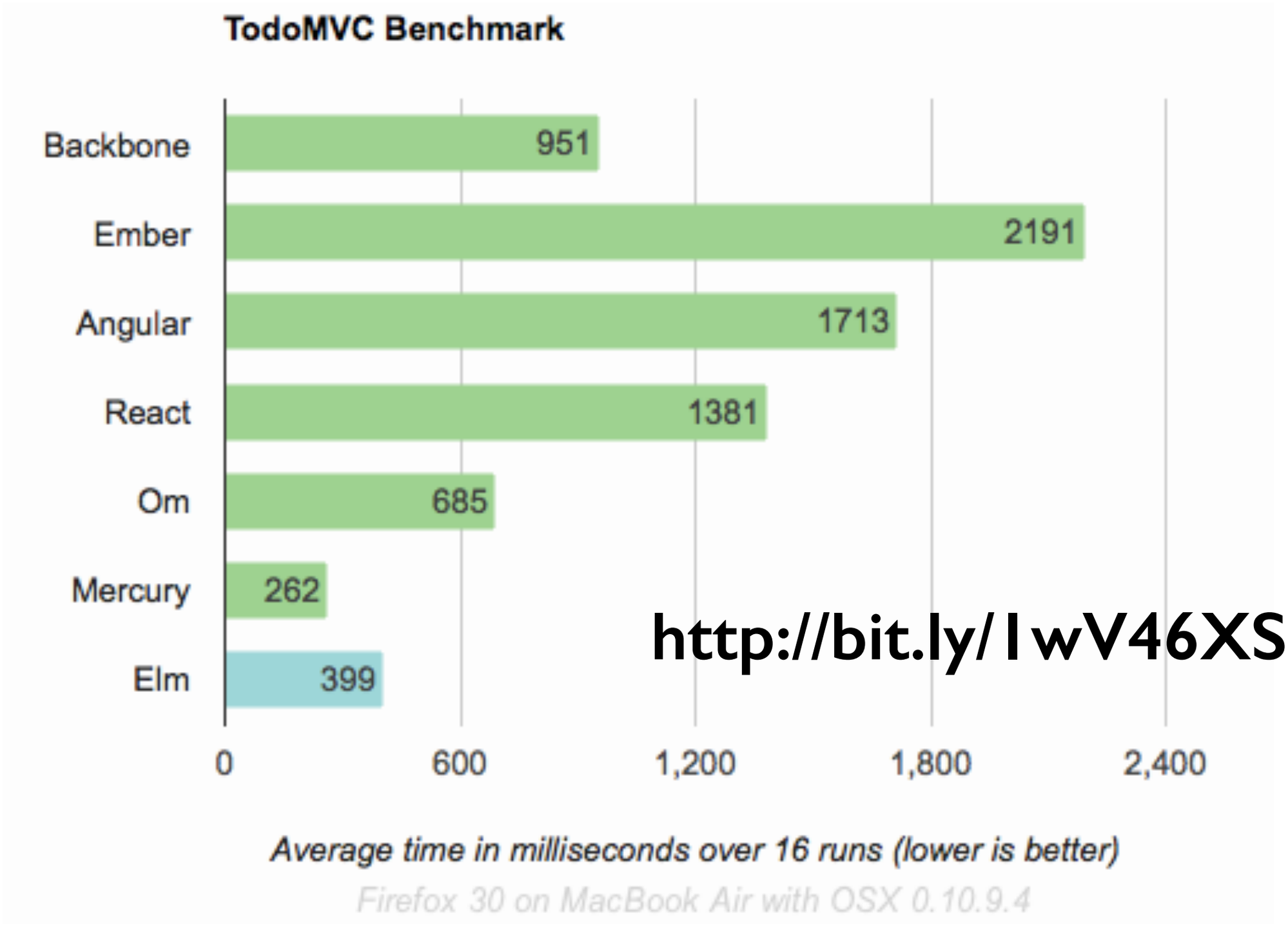
debug.elm-lang.org/edit/Mario.elm

```
34
35 ity dt
36  keys
37  keys
38 ics dt
39 g.watch "mario"
40
41 Model -> Model
42  =
43  0 && mario.vy == 0 then { mario | v
44
45  -> Model -> Model
46 ) =
47
48 f mario.y > 0 then mario.vy - dt/4 e
49
50
51  -> Model -> Model
52 ) =
53
```

Built-in syntax for defining values, pronounce... more

Auto-update: ☑ Hot Swap Compile

Hints: ☑ Options: ☐



http://bit.ly/1wV46XS

Elm Basics

```
add x y = x + y
```

```
add : Int -> Int -> Int
add x y = x + y
```

```
calcAngle start end =
  let distH  = end.x - start.x
      distV  = end.y - start.y
  in atan2 distV distH
```

```
calcAngle start end =
    let distH  = end.x - start.x
        distV  = end.y - start.y
    in atan2 distV distH
```

```
calcAngle start end =
    let distH  = end.x - start.x
        distV  = end.y - start.y
    in atan2 distV distH
```

```
multiply x y = x * y

triple = multiply 3
```

```
multiply x y = x * y

triple = multiply 3
```

```
f a b c d = ...
f : Int ->
      (Int ->
        (Int ->
          (Int -> Int)))
```

```
double list = List.map (\x -> x * 2) list
```

```
double list = List.map ((*) 2) list
```

```
tuple1 = (2, "three")
tuple2 = (2, "three", [4, 5])
```

```
tuple4 = (,) 2 “three”
tuple5 = (,,) 2 “three” [4, 5]
```

x = { age=42, name=“foo” }

lightweight, labelled data structure

```
x.age    -- 42
x.name   -- “foo”
```

```
x.age      -- 42
x.name     -- “foo”

.age x     -- 42
.name x    -- “foo”
```

-- clone and update

```
y = { x | name <- "bar" }
```

```
type alias Character =
  { age : Int, name : String }
```

```
type alias Named a = { a | name : String }
type alias Aged a = { a | age : Int }
```

lady : Named (Aged { })

lady = { name=“foo”, age=42 }

```
getName : Named x -> String
getName { name } = name
```

```
getName : Named x -> String
getName { name } = name
```

```
getName lady    -- “foo”
```

```
type Status = Flying Pos Speed
            | Exploding Radius
            | Exploded
```

aka.

“sums-and-products”

data structures

sums :
choice between variants of a type

```
type Status = Flying Pos Speed
            | Exploding Radius
            | Exploded
```

products :
tuple of types

```
type Status = Flying Pos Speed
            | Exploding Radius
            | Exploded
```

```
drawCircle x y radius =
    circle radius
    |> filled (rgb 150 170 150)
    |> alpha 0.5
    |> move (x, y)
```

```
drawCircle x y radius =
    circle radius
    |> filled (rgb 150 170 150)
    |> alpha 0.5
    |> move (x, y)
```

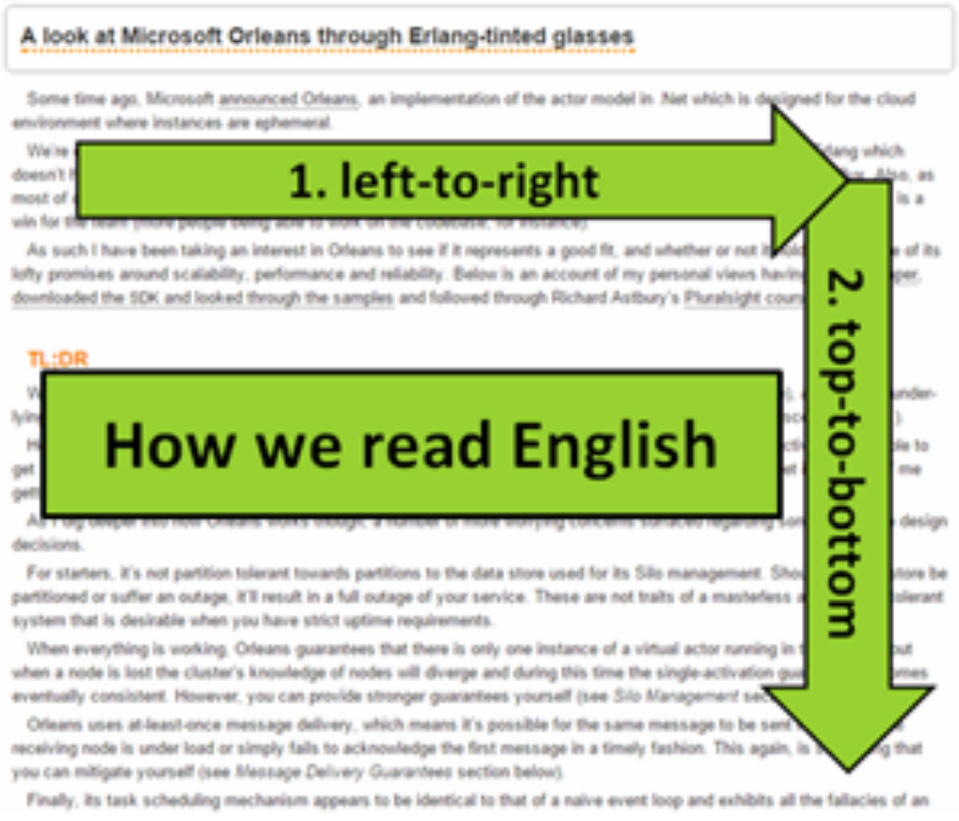
```
filled :Color -> Shape -> Form
```

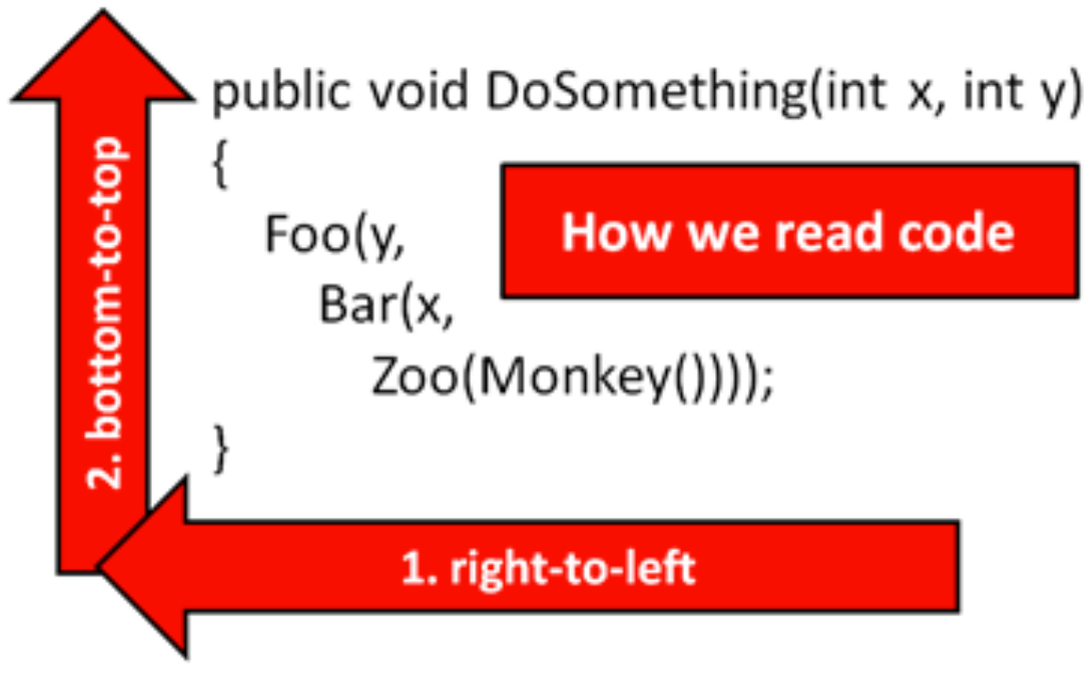
```
drawCircle x y radius =
    circle radius
    |> filled (rgb 150 170 150)
    |> alpha 0.5
    |> move (x, y)
```

```
drawCircle x y radius =
  circle radius
  |> filled (rgb 150 170 150)
  |> alpha 0.5
  |> move (x, y)
```

"…a clean design is one that supports **visual thinking** so people can meet their informational needs with a **minimum** of **conscious effort**."

- Daniel Higginbotham
(www.visualmess.com)


A look at Microsoft Orleans through Erlang-tinted glasses
1. left-to-right
2. top-to-bottom
How we read English
TL;DR


public void DoSomething(int x, int y)
{
Foo(y,
Bar(x,
Zoo(Monkey())));
}
How we read code
2. bottom-to-top
1. right-to-left

```
dr
    radius |> circle
|> filled (rgb 150 1 150)
|> alpha 0.5
|> move (x, y)
```

1. left-to-right

2. top-to-bottom

drawCircle : Int -> Int -> Float -> Form

```
drawCircle x y =
  circle
  >> filled (rgb 150 170 150)
  >> alpha 0.5
  >> move (x, y)
```

```
drawCircle x y =
  circle
    >> filled (rgba 150 170 150)
    >> alpha 0.5
    >> move (x, y)
```

circle : Float -> Shape

```
drawCircle x y =
  (Float -> Shape)
  >> filled (rgb 150 170 150)
  >> alpha 0.5
  >> move (x, y)
```

```
drawCircle x y =
    (Float -> Shape)
    >> filled (rgb 150 170 150)
    >> alpha 0.5
    >> move (x, y)
```

filled : Color -> Shape -> Form

```
drawCircle x y =
    (Float -> Shape)
```

Curried!

```
    >> filled (rgb 150 170 150)
    >> alpha 0.5
    >> move (x, y)
```

```
filled : Color -> Shape -> Form
```

```
drawCircle x y =
    (Float -> Shape)
    >> filled (rgb 150 170 150)
    >> alpha 0.5
    >> move (x, y)
```

Shape -> Form

```
drawCircle x y =
    (Float -> Shape)
    >> (Shape -> Form)
    >> alpha 0.5
    >> move (x, y)
```

```
drawCircle x y =
    (Float -> Shape)
>> (Shape -> Form)
>> alpha 0.5
>> move (x, y)
```

```
drawCircle x y =
  (Float -> Shape)
>> (Shape -> Form)
>> alpha 0.5
>> move (x, y)
```

```
drawCircle x y =
   (Float -> Shape)
   >> (Shape -> Form)
   >> alpha 0.5
   >> move (x, y)
```

```
drawCircle x y =
  (Float -> Form)
>> alpha 0.5
>> move (x, y)
```

```
drawCircle x y =
  (Float -> Form)
>> (Form -> Form)
>> move (x, y)
```

```
drawCircle x y =
  (Float -> Form)
  >> (Form -> Form)
  >> move (x, y)
```

```
drawCircle x y =
  (Float -> Form)
>> move (x, y)
```

```
drawCircle x y =
  (Float -> Form)
>> (Form -> Form)
```

```
drawCircle x y =
  (Float -> Form)
>> (Form -> Form)
```

```
drawCircle x y =
  (Float -> Form)
```

drawCircle : Int -> Int -> **(Float -> Form)**

```
greet name =
  case name of
  "Yan"   -> “hi, theburningmonk"
  _       -> “hi,“ ++ name
```

```
greet name =
  case name of
    "Yan"   -> “hi, theburningmonk"
    _       -> “hi,“ ++ name
```

```
fizzbuzz n =
  if | n % 15 == 0 -> "fizz buzz"
     | n % 3  == 0 -> "fizz"
     | n % 5  == 0 -> "buzz"
     | otherwise   -> show n
```

Mouse.position
Mouse.clicks
Mouse.isDown
...

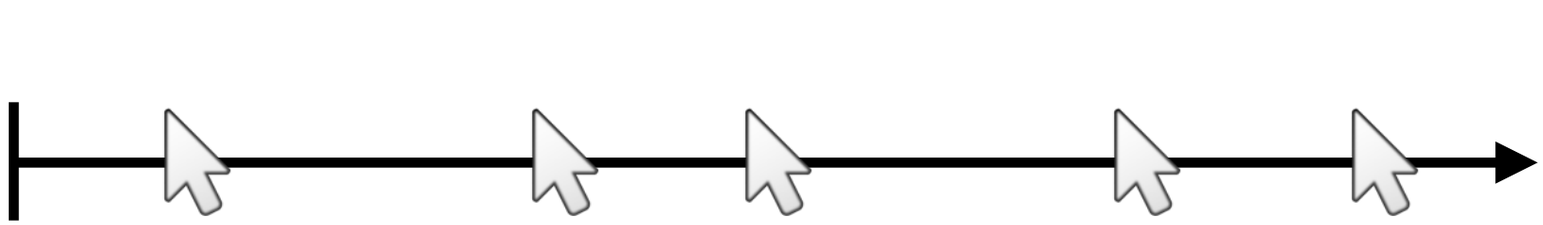
Window.dimension
Window.width
Window.height

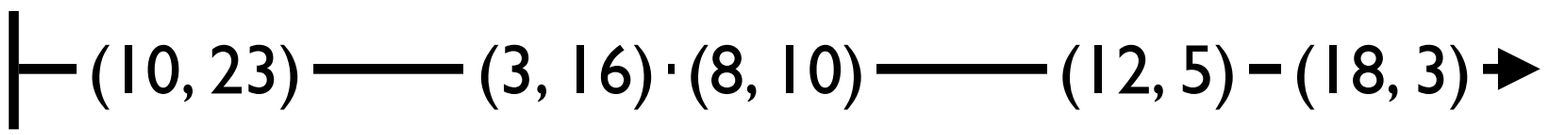
Time.every
Time.fps
Time.timestamp
Time.delay
...

Mouse.position : Signal (Int, Int)

Mouse.position : Signal (Int, Int)

Mouse.position : Signal (Int, Int)

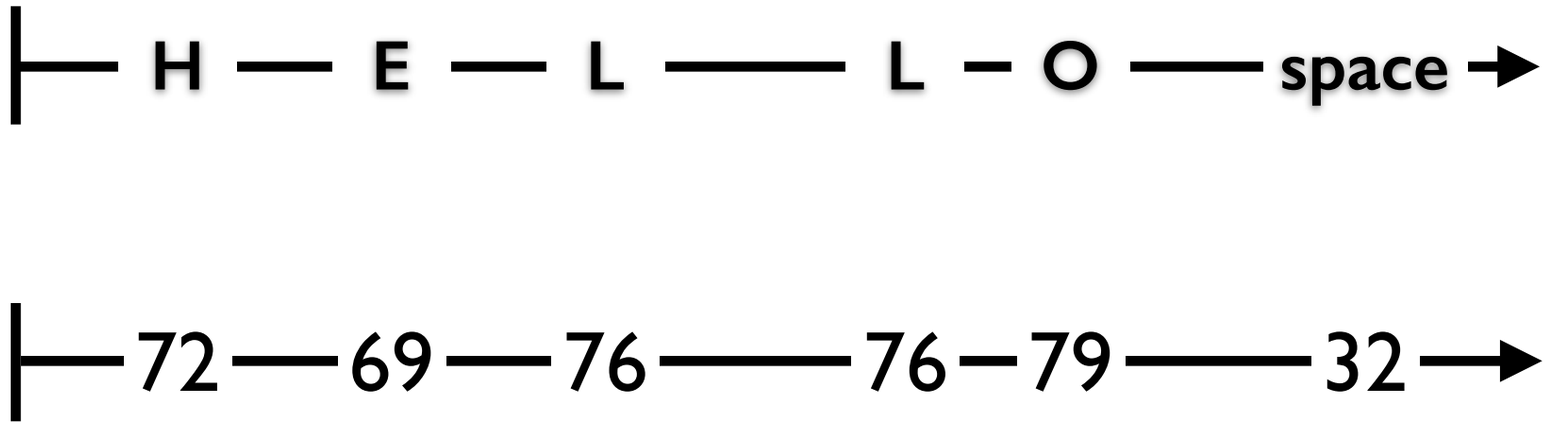




Keyboard.lastPressed : Signal Int

Keyboard.lastPressed : Signal Int

Keyboard.lastPressed : Signal Int



```
map : (a -> b) -> Signal a -> Signal b
```

<~

Signal of num of pixels in window

```
(\(w, h) -> w*h) <~ Window.dimensions
```

```
(\(w, h) -> w*h) <~ Window.dimensions
```

(Int, Int) -> Int Signal **(Int, Int)**

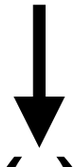
```
Signal Int
(\(w, h) -> w*h) <~ Window.dimensions
(Int, Int) -> Int    Signal (Int, Int)
```


```
(\(w, h) -> w*h) <~ Window.dimensions
```

```
map2 : (a -> b -> c)
       -> Signal a
       -> Signal b
       -> Signal c
```

≈

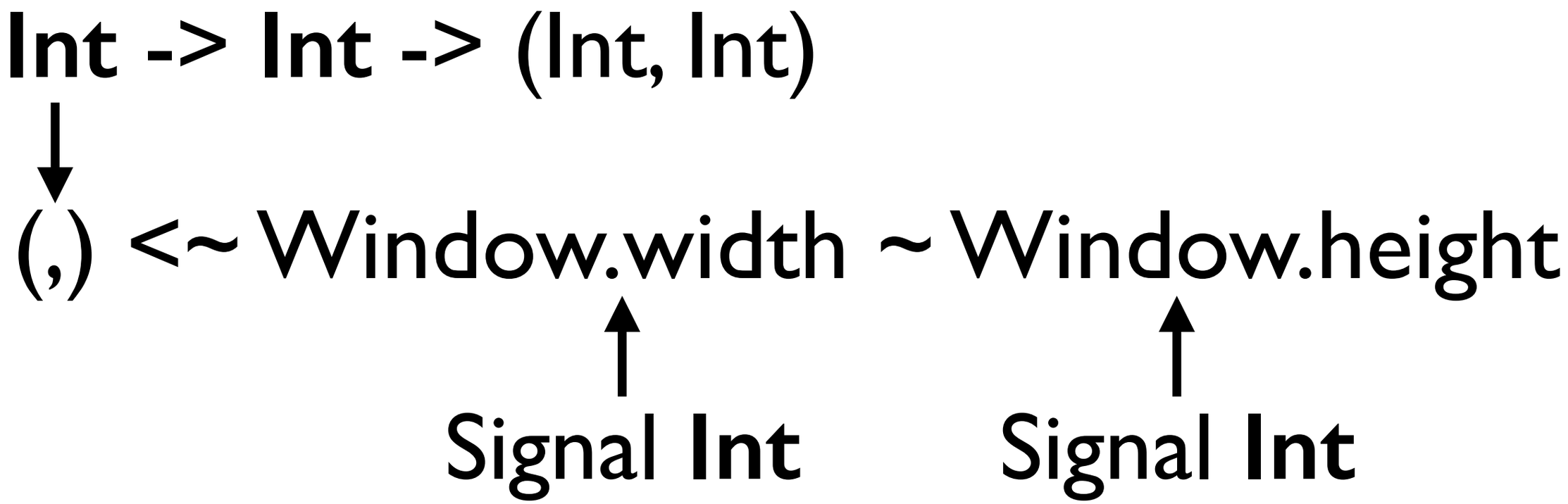
```
a -> b -> (a, b)
```



```
(,) <~ Window.width ~ Window.height
```

Signal Int

Signal Int


Int -> Int -> (Int, Int)
(,) <~ Window.width ~ Window.height
Signal Int
Signal Int

```
map3 : (a -> b -> c -> d)
    -> Signal a
    -> Signal b
    -> Signal c
    -> Signal d
```

```
(,,) <~ signalA ~ signalB ~ signalC
```

map4 : …
map5 : …
map6 : …
map7 : …
map8 : …

```
foldp : (a -> b -> b)
        -> b
        -> Signal a
        -> Signal b
```

```
foldp : (a -> b -> b)
        -> b
        -> Signal a
        -> Signal b
```

```
foldp : (a -> b -> b)
        -> b
        -> Signal a
        -> Signal b
```

```
foldp : (a -> b -> b)
        -> b
        -> Signal a
        -> Signal b
```

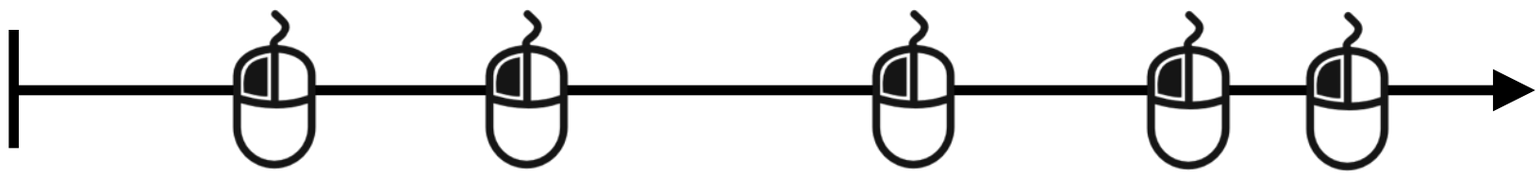
```
foldp : (a -> b -> b)
        -> b
        -> Signal a
        -> Signal b
```

```
foldp : (a -> b -> b)
        -> b
        -> Signal a
        -> Signal b
```

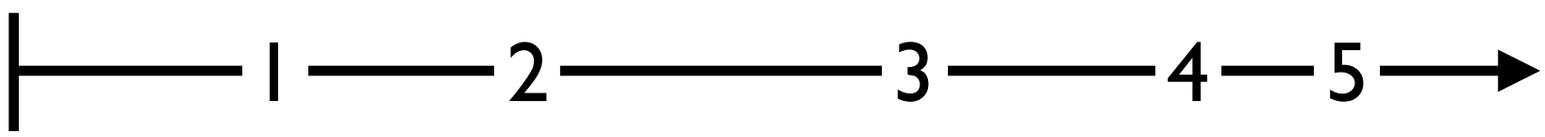
```
foldp (\_ n -> n + 1) 0 Mouse.clicks
```

```
foldp (\_ n -> n + 1) 0 Mouse.clicks
```

```
foldp (\_ n -> n + 1) 0 Mouse.clicks
```



```
foldp (\_ n -> n + 1) 0 Mouse.clicks
```



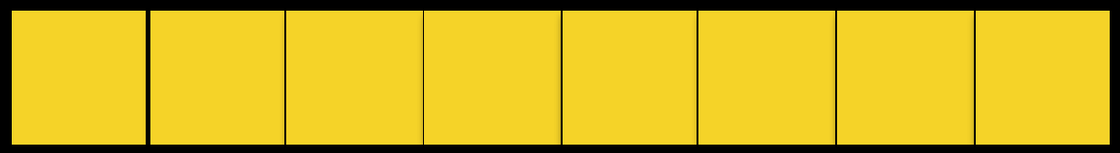
```
UP      { x=0,  y=1  }
DOWN    { x=0,  y=-1 }
LEFT    { x=-1, y=0  }
RIGHT   { x=1,  y=0  }
```

```
merge  : Signal a -> Signal a -> Signal a
mergeMany : List (Signal a) -> Signal a
...
```

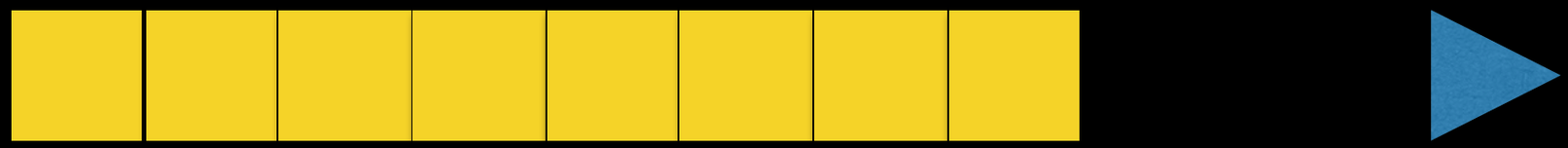
Js Interop, WebGL

HTML layout, dependency management, etc.

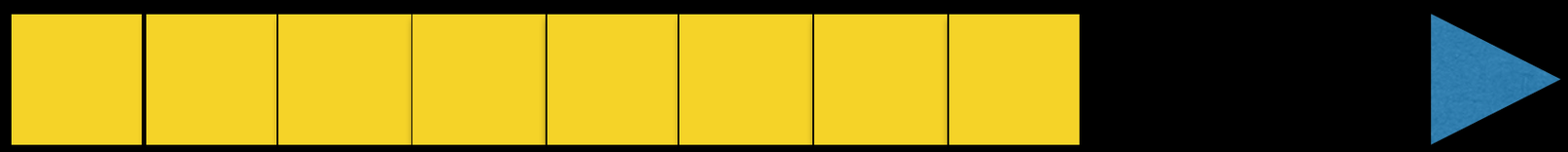

Press SPACE to start.



8 segments

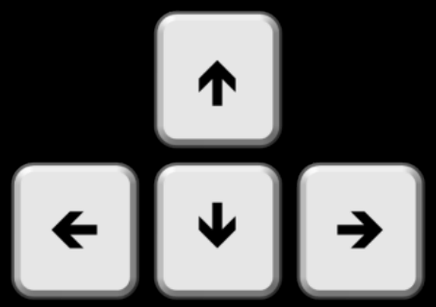


direction



direction

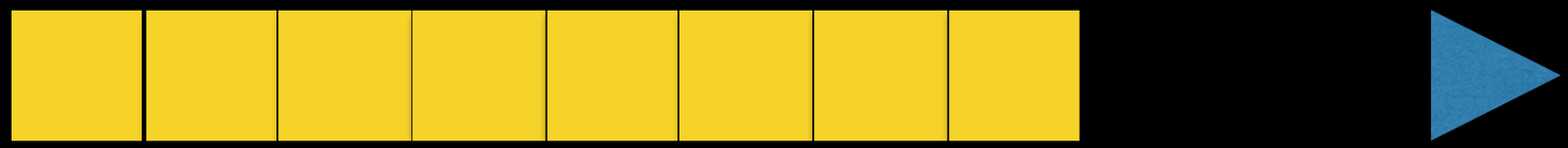
change



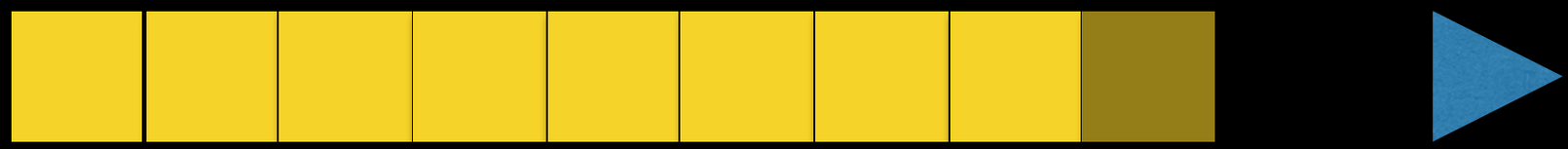
not allowed

no change

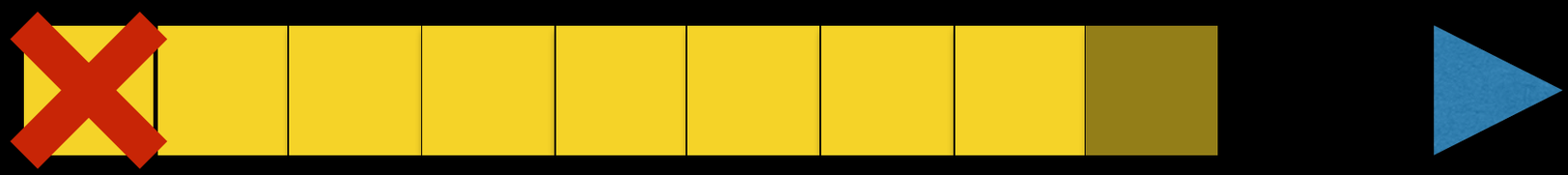
change



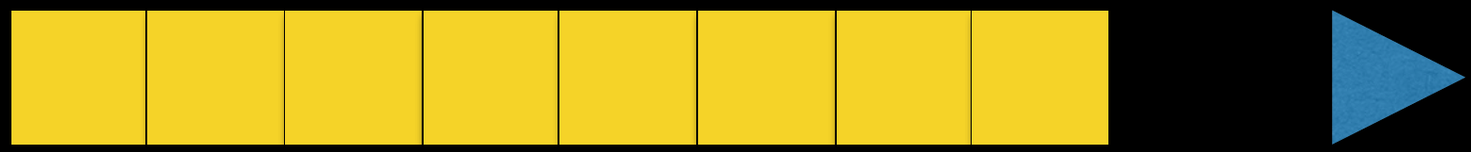
direction



direction

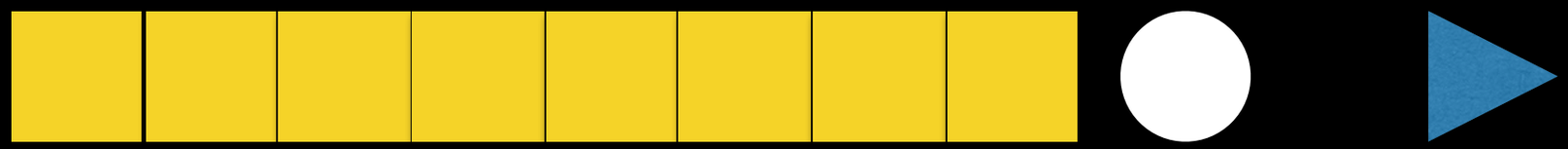


direction



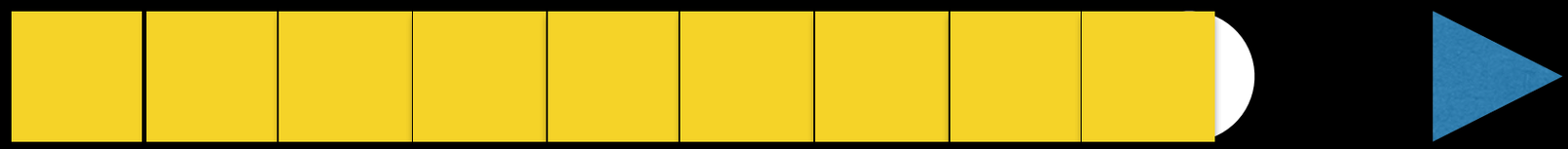
direction

cherry



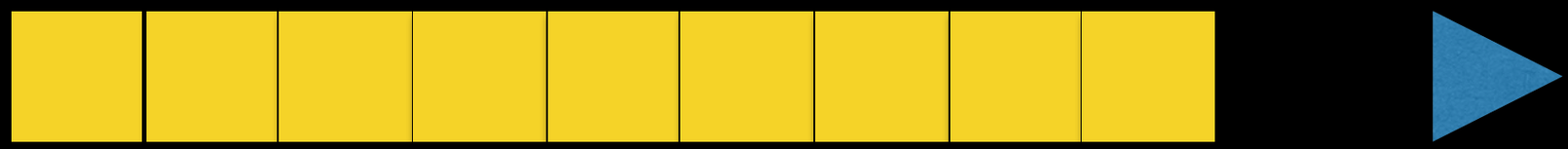
direction

YUM YUM YUM!



direction

+1 segment



direction

```
import Graphics.Element (..)
import Keyboard
import Signal
import Signal (..)
import Text
import Time
import Window

type alias UserInput = {}

userInput : Signal UserInput
userInput = Signal.constant {}

type alias Input =
    { timeDelta : Float
    , userInput : UserInput
    }

type alias GameState = {}

defaultGame : GameState
defaultGame = {}

stepGame : Input -> GameState -> GameState
stepGame {timeDelta,userInput} gameState = gameState

display : (Int,Int) -> GameState -> Element
display (w,h) gameState = Text.asText gameState

delta : Signal Float
delta = Time.fps 30

input : Signal Input
input = Signal.sampleOn delta (Input<~delta~userInput)

gameState : Signal GameState
gameState = Signal.foldp stepGame defaultGame input

main : Signal Element
main = display<~Window.dimensions~gameState
```

```elm
import Graphics.Element (..)
import Keyboard
import Signal
import Signal (..)
import Text
import Time
import Window

type alias UserInput = {}

userInput : Signal UserInput
userInput = Signal.constant {}

type alias Input =
    { timeDelta : Float
    , userInput : UserInput
    }

type alias GameState = {}

defaultGame : GameState
defaultGame = {}

stepGame : Input -> GameState -> GameState
stepGame {timeDelta,userInput} gameState = gameState

display : (Int,Int) -> GameState -> Element
display (w,h) gameState = Text.asText gameState

delta : Signal Float
delta = Time.fps 30

input : Signal Input
input = Signal.sampleOn delta (Input<~delta~userInput)

gameState : Signal GameState
gameState = Signal.foldp stepGame defaultGame input

main : Signal Element
main = display<~Window.dimensions~gameState
```

```elm
import Graphics.Element (..)
import Keyboard
import Signal
import Signal (..)
import Text
import Time
import Window

type alias UserInput = {}

userInput : Signal UserInput
userInput = Signal.constant {}

type alias Input =
    { timeDelta : Float
    , userInput : UserInput
    }

type alias GameState = {}

defaultGame : GameState
defaultGame = {}

stepGame : Input -> GameState -> GameState
stepGame {timeDelta,userInput} gameState = gameState

display : (Int,Int) -> GameState -> Element
display (w,h) gameState = Text.asText gameState

delta : Signal Float
delta = Time.fps 30

input : Signal Input
input = Signal.sampleOn delta (Input<~delta~userInput)

gameState : Signal GameState
gameState = Signal.foldp stepGame defaultGame input

main : Signal Element
main = display<~Window.dimensions~gameState
```

```elm
import Graphics.Element (..)
import Keyboard
import Signal
import Signal (..)
import Text
import Time
import Window

type alias UserInput = {}

userInput : Signal UserInput
userInput = Signal.constant {}

type alias Input =
    { timeDelta : Float
    , userInput : UserInput
    }

type alias GameState = {}

defaultGame : GameState
defaultGame = {}

stepGame : Input -> GameState -> GameState
stepGame {timeDelta,userInput} gameState = gameState

display : (Int,Int) -> GameState -> Element
display (w,h) gameState = Text.asText gameState

delta : Signal Float
delta = Time.fps 30

input : Signal Input
input = Signal.sampleOn delta (Input<~delta~userInput)

gameState : Signal GameState
gameState = Signal.foldp stepGame defaultGame input

main : Signal Element
main = display<~Window.dimensions~gameState
```

```elm
import Graphics.Element (..)
import Keyboard
import Signal
import Signal (..)
import Text
import Time
import Window

type alias UserInput = {}

userInput : Signal UserInput
userInput = Signal.constant {}

type alias Input =
    { timeDelta : Float
    , userInput : UserInput
    }

type alias GameState = {}

defaultGame : GameState
defaultGame = {}

stepGame : Input -> GameState -> GameState
stepGame {timeDelta,userInput} gameState = gameState

display : (Int,Int) -> GameState -> Element
display (w,h) gameState = Text.asText gameState

delta : Signal Float
delta = Time.fps 30

input : Signal Input
input = Signal.sampleOn delta (Input<~delta~userInput)

gameState : Signal GameState
gameState = Signal.foldp stepGame defaultGame input

main : Signal Element
main = display<~Window.dimensions~gameState
```

```elm
import Graphics.Element (..)
import Keyboard
import Signal
import Signal (..)
import Text
import Time
import Window

type alias UserInput = {}

userInput : Signal UserInput
userInput = Signal.constant {}

type alias Input =
    { timeDelta : Float
    , userInput : UserInput
    }

type alias GameState = {}

defaultGame : GameState
defaultGame = {}

stepGame : Input -> GameState -> GameState
stepGame {timeDelta,userInput} gameState = gameState

display : (Int,Int) -> GameState -> Element
display (w,h) gameState = Text.asText gameState

delta : Signal Float
delta = Time.fps 30

input : Signal Input
input = Signal.sampleOn delta (Input<~delta~userInput)

gameState : Signal GameState
gameState = Signal.foldp stepGame defaultGame input

main : Signal Element
main = display<~Window.dimensions~gameState
```

Demo

Elm Debugger: Stamps

debug.elm-lang.org/edit/Stamps.elm

```
1  module Snake where
2
3  import Window
4
5  type UserInput = {}
6
7  userInput : Signal UserInput
8  userInput = constant {}
9
10 type Input = { timeDelta:Float, userInput:UserInput }
11
12 type GameState = {}
13
14 defaultGame : GameState
15 defaultGame = {}
16
17
18 stepGame : Input -> GameState -> GameState
19 stepGame {timeDelta,userInput} gameState = gameState
20
21
22 display : (Int,Int) -> GameState -> Element
```

```
Runtime error in module Snake (on line 23, column 27 to 43):
    Non-exhaustive pattern match in case-expression.
    Make sure your patterns cover every case!
    Open the developer console for more details.
```

Hints: ☑ Options: ☐ Auto-update: ☑ Hot Swap Compile

Elm Debugger: Stamps

debug.elm-lang.org/edit/Stamps.elm

```
1  module Snake where
2
3  import Window
4
5  type UserInput = {}
6
7  userInput : Signal UserInput
8  userInput = constant {}
9
10 type Input = { timeDelta:Float, userInput:UserInput }
11
12 type GameState = {}
13
14 defaultGame : GameState
15 defaultGame = {}
16
17
18 stepGame : Input -> GameState -> GameState
19 stepGame {timeDelta,userInput} gameState = gameState
20
21
22 display : (Int,Int) -> GameState -> Element
```

```
Runtime error in module Snake (on line 23, column 27 to 43):
    Non-exhaustive pattern match in case-expression.
    Make sure your patterns cover every case!
    Open the developer console for more details.
```

Hints: ☑ Options: ☐ Auto-update: ☑ Hot Swap Compile

Elm Debugger: Stamps

debug.elm-lang.org/edit/Stamps.elm

```
20 type Input = { timeDelta:Float, userInput:UserInput }
21
22 type GameState = {}
23
24 defaultGame : GameState
25 defaultGame = {}
26
27
28 stepGame : Input -> GameState -> GameState
29 stepGame {timeDelta,userInput} gameState = gameState
30
31
32 display : (Int,Int) -> GameState -> Element
33 display (w,h) gameState = asText gameState
34
35
36 delta = fps 20
37 input = sampleOn delta (Input<~delta~userInput)
38
39 gameState = foldp stepGame defaultGame input
40
```

Hints: ☑ Options: ☐ Auto-update: ☑ Hot Swap Compile

```
Runtime error in module
Sn
co
    Error on line 9, col
    Could not find varia
    Non-exhaustive
pattern match in case-
expression.
    Make sure your
patterns cover every
case!
    Open the developer
console for more
details.
```

Elm Debugger: Stamps

debug.elm-lang.org/edit/Stamps.elm

```
33        |> List.map (\x -> (x * 15.0, 0)) }
34 type Cherry    = Maybe (Float, Float)
35 data GameState = NotStarted | Started (Snake, Cherry)
36
37 defaultGame : GameState
38 defaultGame = NotStarted
39
40
41 stepGame : Input -> GameState -> GameState
42 stepGame {timeDelta,userInput} gameState = gameState
43
44
45 display : (Int,Int) -> GameState -> Element
46 display (w,h) gameState = asText gameState
47
48
49 delta = fps 20
50 input = sampleOn delta (Input<~delta~userInput)
51
52 gameState = foldp stepGame defaultGame input
53
54 main = display<~Window.dimensions~gameState
```

```
Runtime error in module
Snake (on line 23,
column 27 to 43):
    Non-exhaustive
pattern match in case-
expression.
    Make sure your
patterns cover every
case!
    Open the developer
console for more
details.
```

Hints: ☑ Options: ☐ Auto-update: ☑ Hot Swap Compile

Elm Debugger: Stamps

debug.elm-lang.org/edit/Stamps.elm

```
1  module Snake where
2
3  import Window
4  import List
5  import Keyboard
6  import Text
7
8  data UserInput = Arrow { x:Int, y:Int } | Space
9  defaultUserInput = Arrow {x=0, y=0}
10
11 arrows : Signal UserInput
12 arrows = Arrow <~ Keyboard.arrows
13
14 spaces : Signal UserInput
15 spaces =
16   (\pressed ->
17     if pressed then Space
18     else defaultUserInput) <~ Keyboard.space
19
20 userInput : Signal UserInput
21 userInput = merge arrows spaces
22
```

Press SPACE to

data Text.Text more

Hints: Options: Auto-update: Hot Swap Compile

Elm Debugger: Stamps

debug.elm-lang.org/edit/Stamps.elm

```
35 type Cherry    = Maybe (Float, Float)
36 data GameState = NotStarted | Started (Snake, Cherry)
37
38 defaultGame : GameState
39 defaultGame = NotStarted
40
41
42 stepGame : Input -> GameState -> GameState
43 stepGame {timeDelta,userInput} gameState = gameState
44
45 txt msg = msg
46           |> toText
47           |> Text.color white
48           |> Text.monospace
49           |> leftAligned
50           |> toForm
51
52 display : (Int,Int) -> GameState -> Element
53 display (w,h) gameState =
54   let background =
55         rect (toFloat w) (toFloat h)
56         |> filled (rgb 0 0 0)
```

Press SPACE to start.

Hints: ☑ Options: ☐ Auto-update: ☑ Hot Swap Compile

Elm Debugger: Stamps

debug.elm-lang.org/edit/Stamps.elm

```
22 userInput = merge arrows spaces
23
24 type Input = { timeDelta:Float, userInput:UserInput }
25
26 data Direction = Left | Right | Up | Down
27 type Snake     = { segments  : [(Float, Float)],
28                    direction : Direction }
29 defaultSnake   =
30   { direction = Right,
31     segments =
32       [ 0.0..8.0 ]
33         |> List.reverse
34         |> List.map (\x -> (x * 15.0, 0)) }
35 type Cherry    = Maybe (Float, Float)
36 data GameState = NotStarted | Started (Snake, Cherry)
37
38 defaultGame : GameState
39 defaultGame = NotStarted
40
41
42 stepGame : Input -> GameState -> GameState
43 stepGame {timeDelta,userInput} gameState =
```

Press SPACE to start.

Hints: ☑ Options: ☐ Auto-update: ☑ Hot Swap Compile

Elm Debugger: Stamps

debug.elm-lang.org/edit/Stamps.elm

```
36 data GameState = NotStarted | Started (Snake, Cherry)
37
38 defaultGame : GameState
39 defaultGame = NotStarted
40
41
42 stepGame : Input -> GameState -> GameState
43 stepGame {windowDim,userInput} gameState =
44   case gameState of
45     NotStarted ->
46       if userInput == Space
47       then Started (defaultSnake, Nothing)
48       else gameState
49
50
51 txt msg = msg
52           |> toText
53           |> Text.color white
54           |> Text.monospace
55           |> leftAligned
56           |> toForm
```

Press SPACE to start.

Hints: ☑ Options: ☐ Auto-update: ☑ Hot Swap Compile

Elm Debugger: Stamps

debug.elm-lang.org/edit/Stamps.elm

```
38 defaultGame : GameState
39 defaultGame = NotStarted
40
41
42 stepGame : Input -> GameState -> GameState
43 stepGame {windowDim,userInput} gameState =
44   case gameState of
45     NotStarted ->
46       if userInput == Space
47       then Started (defaultSnake, Nothing)
48       else gameState
49     Started ({segments, direction}, cherry) ->
50       let arrow = case userInput of
51                     Arrow arrow -> arrow
52                     _           -> { x=0, y=0 }
53           newDirection = getNewDirection arrow direction
54           newHead = getNewSegment (List.head segments) n
55           newTail = List.take (List.length segments-1) s
56           (w, h)  = windowDim
57           isGameOver =
58             List.any (\t -> t == newHead) newTail -- eat
59             || fst newHead > (toFloat w / 2)  -- hit bot
```

```
Error on line 53, columr
Could not find variable

Press SPACE to start.
```

Hints: ☑ Options: ☐ Auto-update: ☑ Hot Swap Compile

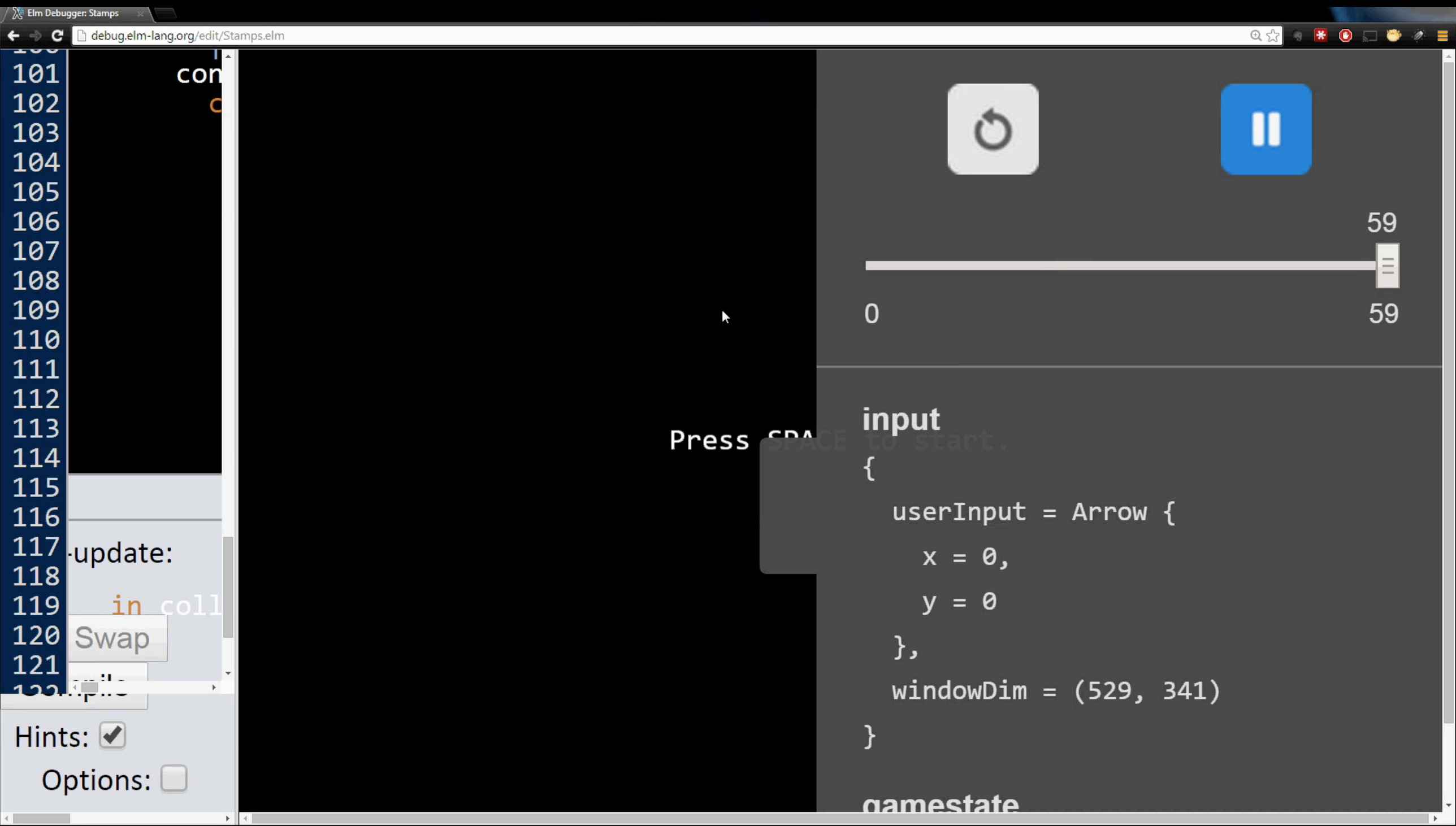
Elm Debugger: Stamps

debug.elm-lang.org/edit/Stamps.elm

```
90              |> toText
91              |> Text.color white
92              |> Text.monospace
93              |> leftAligned
94              |> toForm
95
96  display : (Int,Int) -> GameState -> Element
97  display (w,h) gameState =
98    let background =
99          rect (toFloat w) (toFloat h)
100         |> filled (rgb 0 0 0)
101       content =
102         case Debug.watch "gamestate" gameState of
103           NotStarted ->
104             [ txt "Press SPACE to start." ]
105
106   in collage w h (background::content)
107
108
109 delta = fps 20
110 input = sampleOn delta (Input<~Window.dimensions~userInp
111
112
```

Press SPACE to start.

Hints: ☑ Options: ☐ Auto-update: ☑ Hot Swap Compile


Elm Debugger: Stamps
debug.elm-lang.org/edit/Stamps.elm
update:
in coll
Swap
Hints:
Options:
Press SPACE to start.
59
0
59
input
{
userInput = Arrow {
x = 0,
y = 0
},
windowDim = (529, 341)
}
gamestate

Elm Debugger: Stamps

debug.elm-lang.org/edit/Stamps.elm

```
11 cherryRadius = 7.5
12
13 data UserInput = Arrow { x:Int, y:Int } | Space
14 defaultUserInput = Arrow {x=0, y=0}
15
16 arrows : Signal UserInput
17 arrows = Arrow <~ Keyboard.arrows
18
19 spaces : Signal UserInput
20 spaces =
21   (\pressed ->
22     if pressed then Space
23     else defaultUserInput) <~ Keyboard.space
24
25 userInput : Signal UserInput
26 userInput = merge arrows spaces
27
28 type Input = { windowDim:(Int, Int), userInput:UserInput }
29
30 data Direction = Left | Right | Up | Down
31 type Snake     = { segments  : [(Float, Float)],
32                   direction : Direction }
```

Press SPACE

Hints: ☑ Options: ☐ Auto-update: ☑ Hot Swap Compile

Elm Debugger: Stamps

debug.elm-lang.org/edit/Stamps.elm

```
69    case gameState of
70      NotStarted ->
71        if userInput == Space
72        then Started (defaultSnake, Nothing)
73        else gameState
74      Started ({segments, direction}, cherry) ->
75        let arrow = case userInput of
76                      Arrow arrow -> arrow
77                      _           -> { x=0, y=0 }
78            newDirection = getNewDirection arrow direction
79            newHead = getNewSegment (List.head segments) newDi
80            newTail = List.take (List.length segments-1) segme
81            (w, h)  = windowDim
82            isGameOver =
83              List.any (\t -> t == newHead) newTail -- eat its
84              || fst newHead > (toFloat w / 2)  -- hit bottom
85              || snd newHead > (toFloat h / 2)  -- hit top
86              || fst newHead < (toFloat -w / 2) -- hit left
87              || snd newHead < (toFloat -h / 2) -- hit right
88        in if isGameOver then NotStarted
89           else Started ({ segments = newHead::newTail,
90                           direction = newDirection },
91
```

Press SPACE

Hints: ☑ Options: ☐ Auto-update: ☑ Hot Swap Compile

Elm Debugger: Stamps

debug.elm-lang.org/edit/Stamps.elm

```
54          | y == 1  && currentDi
55          | x == -1 -> Left
56          | x == 1  -> Right
57          | y == -1 -> Down
58          | y == 1  -> Up
59
60  getNewSegment (x, y) direct
61    case direction of
62      Up    -> (x, y+segmentD
63      Down  -> (x, y-segmentD
64      Left  -> (x-segmentDim,
65      Right -> (x+segmentDim,
66
67  isOverlap (snakeX, snakeY)
68    let (xd, yd) = (cherryX -
69        distance = sqrt(xd *
70    in distance <= (cherryRad
71
72
```

Auto-update: ☑ Hot Swap

Compile

Hints: ☑ Options: ☐

Press SPACE to start.

Elm Debugger: Stamps

debug.elm-lang.org/edit/Stamps.elm

```
83    newDirection = getNewDirection arrow direction
84    newHead = getNewSegment (List.head segments) newDirection
85    newTail = List.take (List.length segments-1) segments
86    (w, h)  = windowDim
87    [spawn, rand1, rand2] = randoms
88    ateCherry =
89      case cherry of
90        Nothing -> False
91        Just cherryCentre -> isOverlap newHead cherryCentre
92    newCherry =
93      if ateCherry then Nothing
94      else if cherry == Nothing && spawn <= 0.5
95           then
96             let cherryX = (rand1 * toFloat w) - (toFloat w /2
97                 cherryY = (rand2 * toFloat h) - (toFloat h /
98             in Just (cherryX, cherryY)
99           else cherry
100
101   isGameOver =
102     List.any (\t -> t == newHead) newTail -- eat itself
103     || fst newHead > (toFloat w / 2)  -- hit bottom
104     || snd newHead > (toFloat h / 2)  -- hit top
```

Hints: ☑ Options: ☐ Auto-update: ☑ Hot Swap Compile

Elm Debugger: Stamps

debug.elm-lang.org/edit/Stamps.elm

```
91          Just cherryCentre ->
92      newCherry =
93        if ateCherry then Noth
94        else if cherry == Noth
95             then
96               let cherryX = (
97                   cherryY = (
98               in Just (cherry
99             else cherry
100     newTail =
101       if ateCherry then segm
102       else List.take (List.l
103     isGameOver =
104       List.any (\t -> t == n
105         || fst newHead > (toFl
106         || snd newHead > (toFl
107         || fst newHead < (toFl
108         || snd newHead < (toFl
```

Auto-update: ☑ Hot Swap

Compile

Hints: ☑ Options: ☐

Press SPACE to start.

Snake

github.com/theburningmonk/elm-snake

Missile Command

github.com/theburningmonk/elm-missile-command

elm-lang.org/try

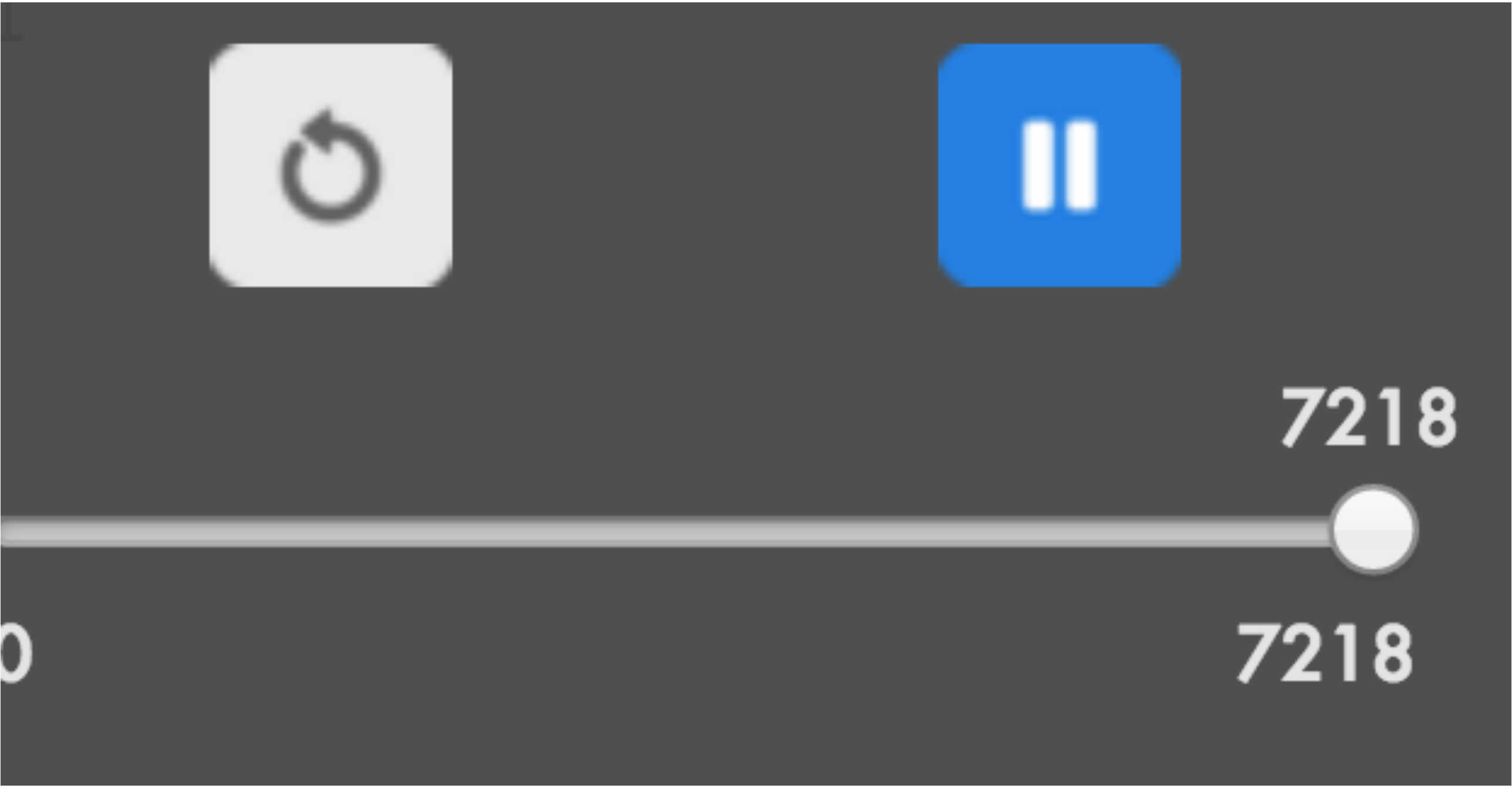
debug.elm-lang.org/try

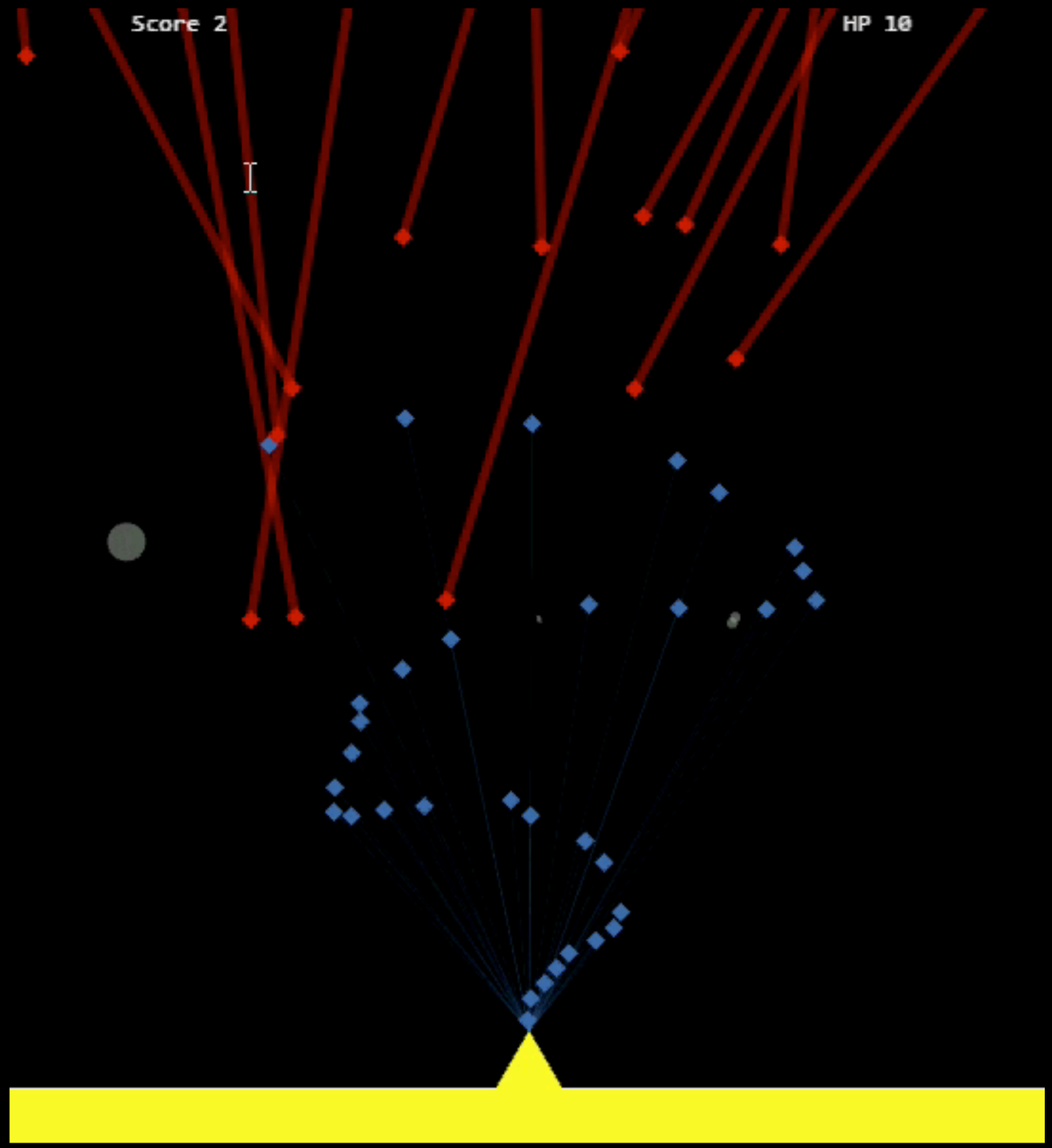
the not so great things

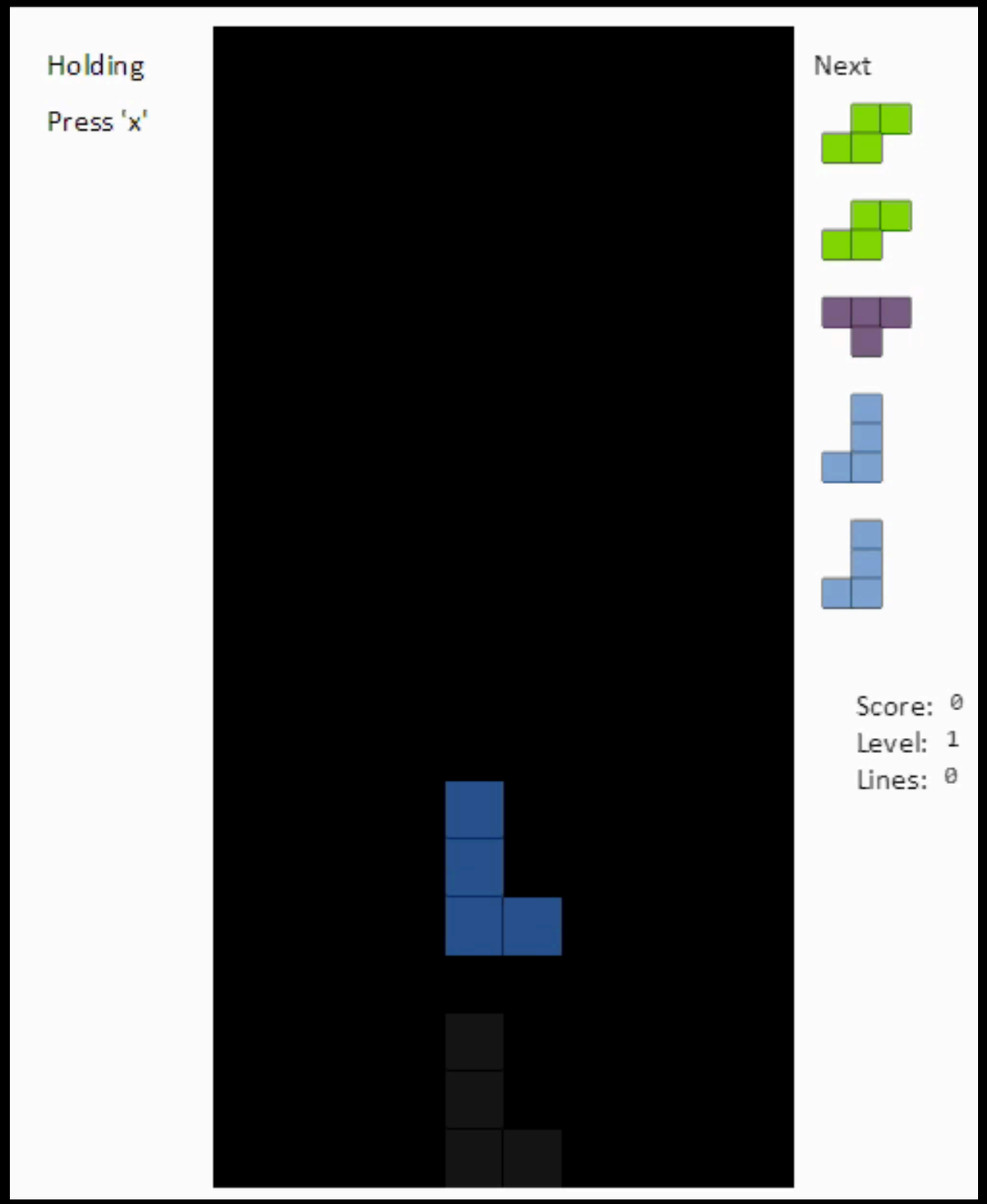
cryptic error messages

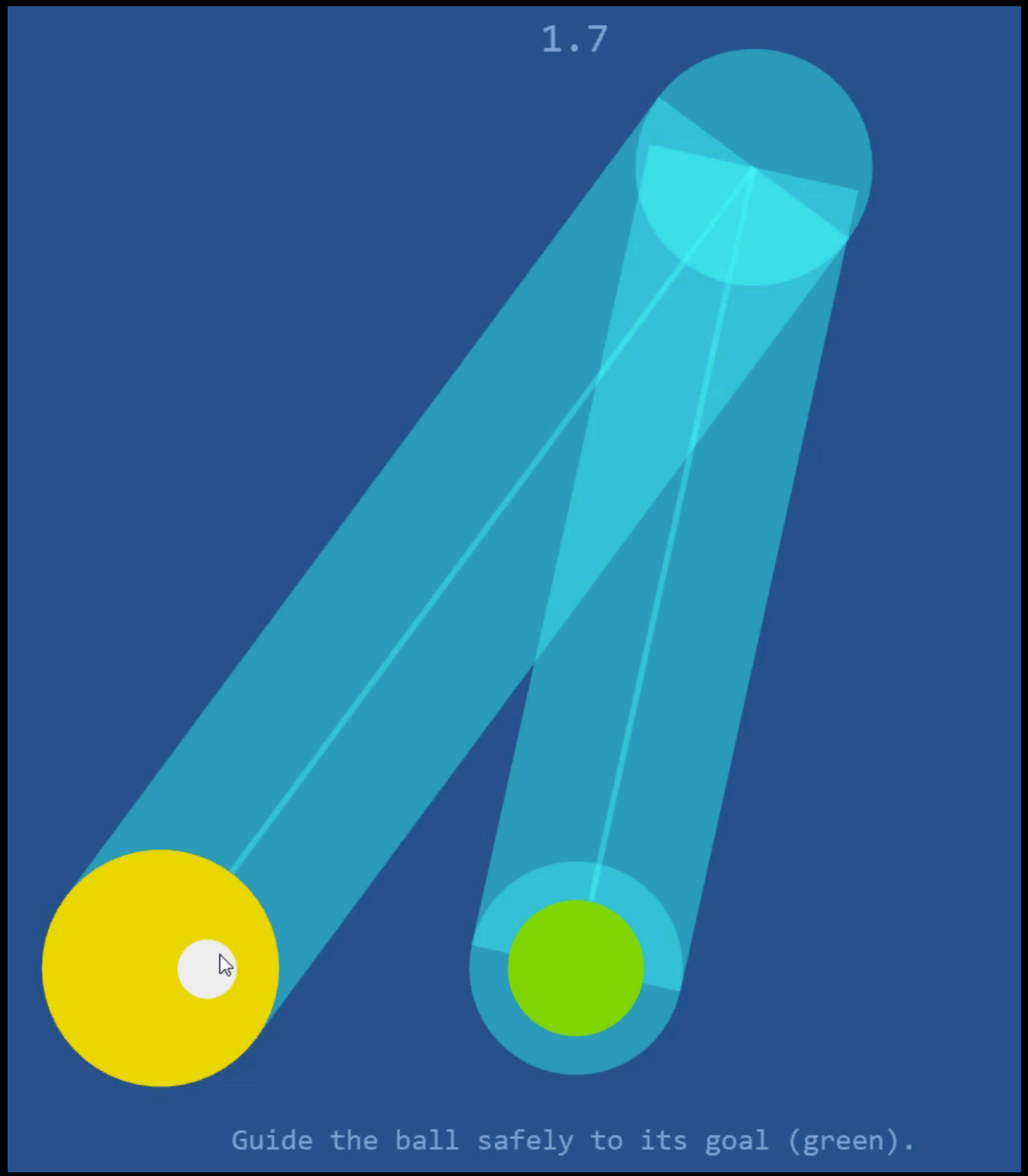
```
Type error between lines 63 and 85:
                case gameState of
    NotStarted -> if | userInput == Space ->
                         Started (defaultSnake,Nothing)
                     | True -> gameState
Started ({segments,direction},cherry) -> let arrow = case userInput
                                                     of
                                           Arrow arrow -> arrow
                                           _ -> {x = 0, y = 0}
                                         newDirection = getNewDirection
                                                        arrow direction
                                         newHead = getNewSegment
                                          (List.head segments) newDirection
                                             newTail = List.take
                                              ((List.length segments) - 1)
                                                      segments
                                           (w,h) = windowDims
                                         isGameOver = (List.any
                                                 (\t -> t == newHead)
                                                      newTail) ||
                                                (((fst newHead) >
                                                 ((toFloat w) / 2)) ||
                                                (((snd newHead) >
                                                 ((toFloat h) / 2)) ||
                                                 (((fst newHead) <
                                                ((toFloat (-w)) / 2)) ||
                                                 ((snd newHead) <
                                                ((toFloat (-h)) / 2)))))
                                       in if | isGameOver -> NotStarted
                                             | True ->
                                                 Started
                                           ({segments = newHead :: newTail,
                                              direction = newDirection},
                                             cherry)

    Expected Type: {}
Actual Type: Snake.Input
```

breaking changes


7218
7218
0


Score 2
HP 10


Holding
Press 'x'
Next
Score: 0
Level: 1
Lines: 0


1.7
Guide the ball safely to its goal (green).


SCORE: 0


THANK
YOU
GRACIAS
ARIGATO
SHUKURIA
DANKSCHEEN
BÏYAN
SHUKRIA
TINGKI
YAQHANYELAY
TASHAKKUR ATU
SUKSAMA
EKHMET
SPASIBO
DENKAUJA
NENACHALHYA
SPASSIBO
SNACHALHUYA
NUHUN
CHALTU
WABEEJA
MAITEKA
YUSPAGARÀTAM
HUI
GUI
HÀTUR
EKOJU
SIKOMO
UNALCHÉESH
DHANYABAAD
AHHIA
ÀTTO
MERSI
MAAKE
GRAZIE
MEHRBA
PALDIES
LAH
KOMAPSUMNIDA
SANCO
MERASTAWHY
GAEJTHO
AGUYJE
FAKAAUE
GOZAIMASHITA
EFCHARISTO
TAVTAPUCH
MEDAWAGSE
BAIKA
JUSPAXAR
MAKETAI
MINMONCHAR
BOLZÏN
MERCI

theburningmonk.com

@theburningmonk

github.com/theburningmonk

Leave your feedback on Joind.in!
https://joind.in/13665