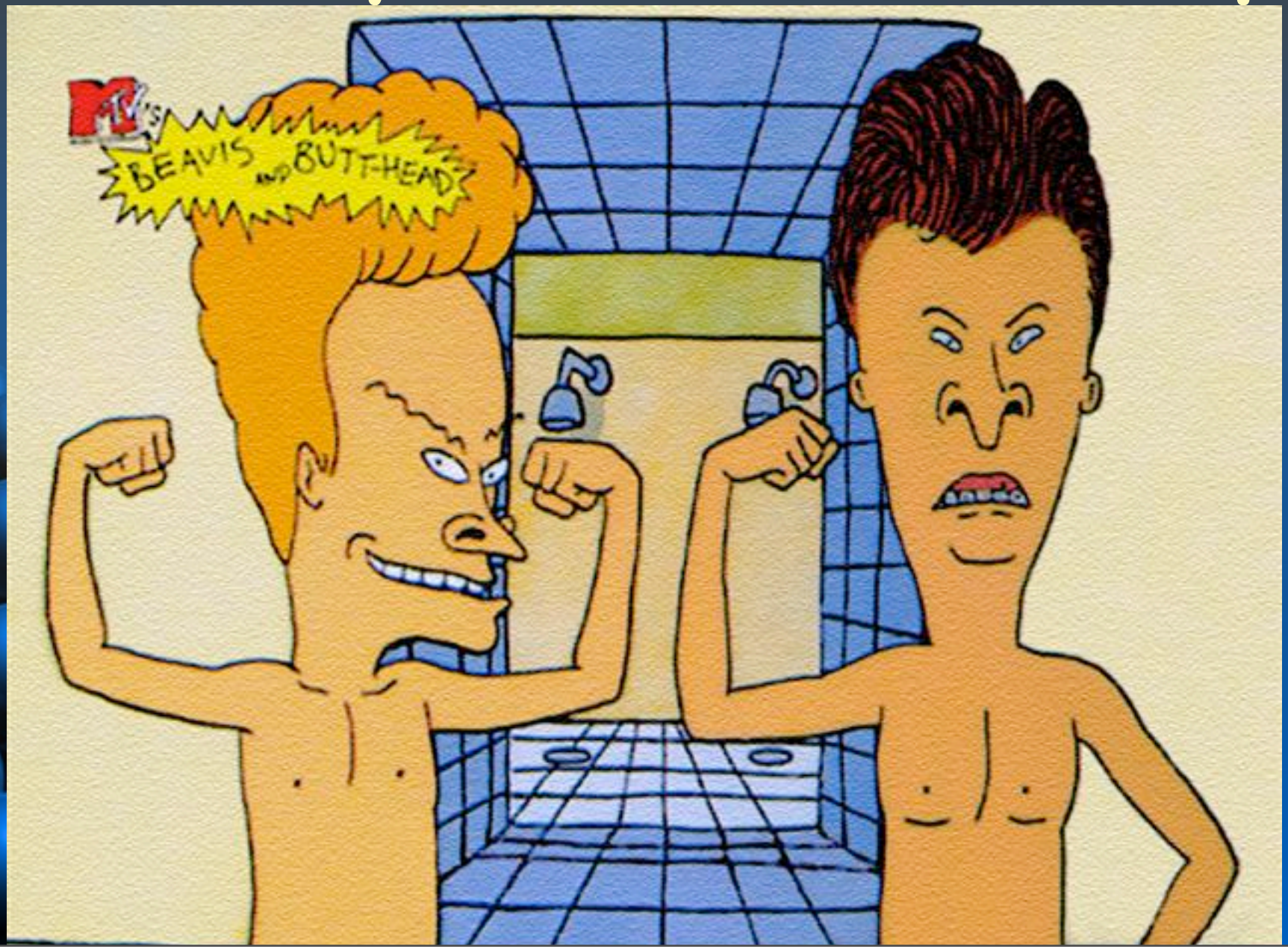
WTF is a Monad?

An exploration in Clojure

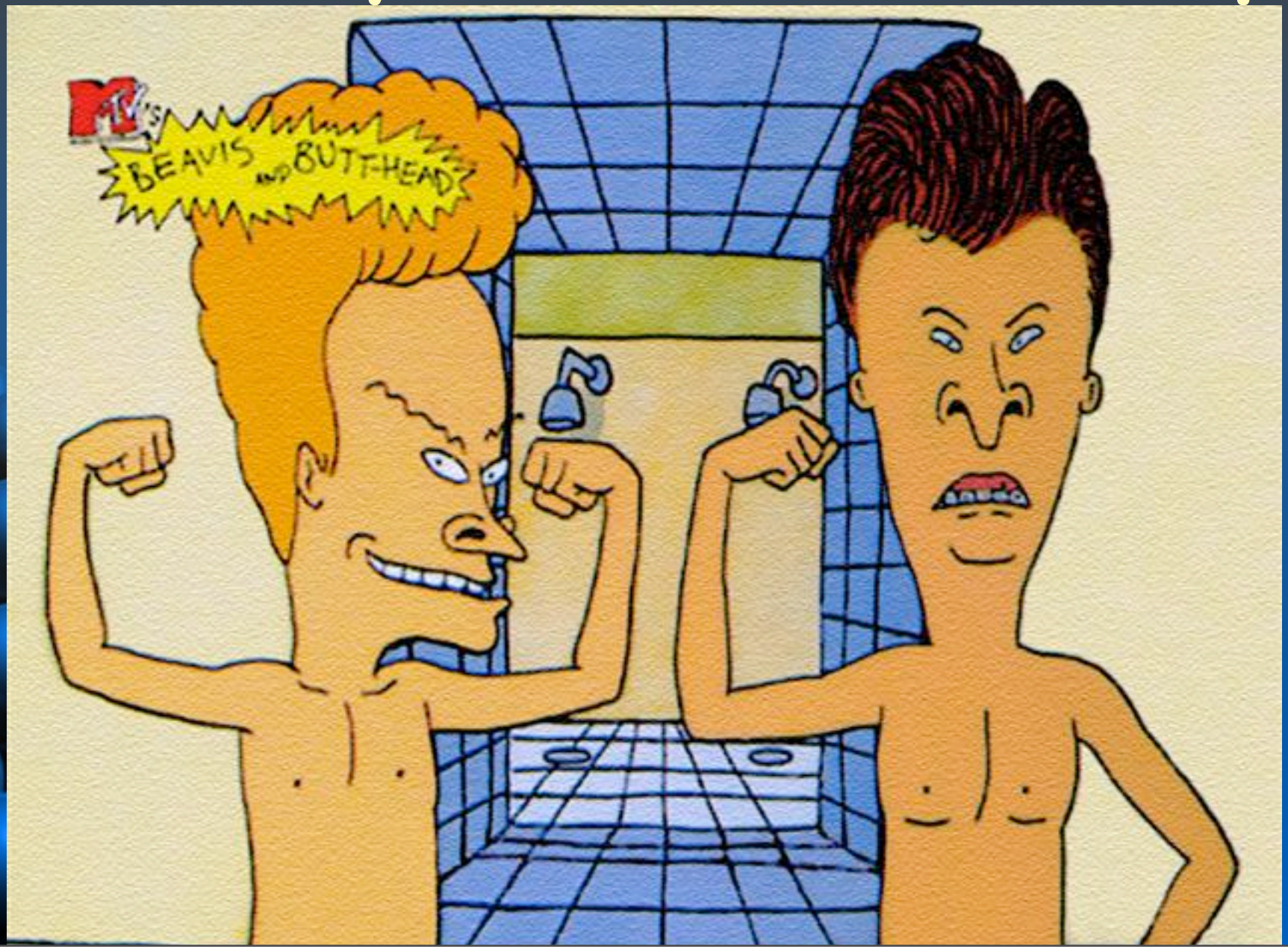
Robert C. Martin
Object Mentor Inc.

The Monads are Coming to EAT your Brains!

In Reality, Monads are Wimps



In Reality, Monads are Wimps



Upper and Lower

- Mondads are adapters.
  - Given a “lower” form (e.g. integers)
    - and functions that manipulate it (e.g. +, -, *, /, etc.)
  - Given an “upper” form (e.g. a string of dots)
    - Where there is a mapping between the lower and upper forms
      - e.g. 3 maps to “...” and vice-versa
  - Then a monad of the upper form allows data in the upper form to be manipulated by functions of the lower form.
    - e.g. You can apply the monad to the + operation to create a new function (dot-add) that takes upper form data.
      - applying dot-add to “...” and “..” yeilds “.....”
  - To summarize: A monad is an adapter between a lower form and an upper form.
    - That is not _all_ monads are, but it’s a good first step.

Dots

Non-Monadic Dots in Ruby And Closure

```ruby
def dotsToN(d)
  d.size()
end

def nToDots(n)
  "."*n
end

def addDots(da, db)
  a = dotsToN(da)
  b = dotsToN(db)
  nToDots(a + b)
end
```

```clojure
(defn dots-to-n [dots]
  (count dots))

(defn n-to-dots [n]
  (apply str
    (repeat n ".")))

(defn add-dots [da db]
  (let [a (dots-to-n da)
        b (dots-to-n db)]
    (n-to-dots (+ a b))))
```

Monadic Dots in Ruby And Closure

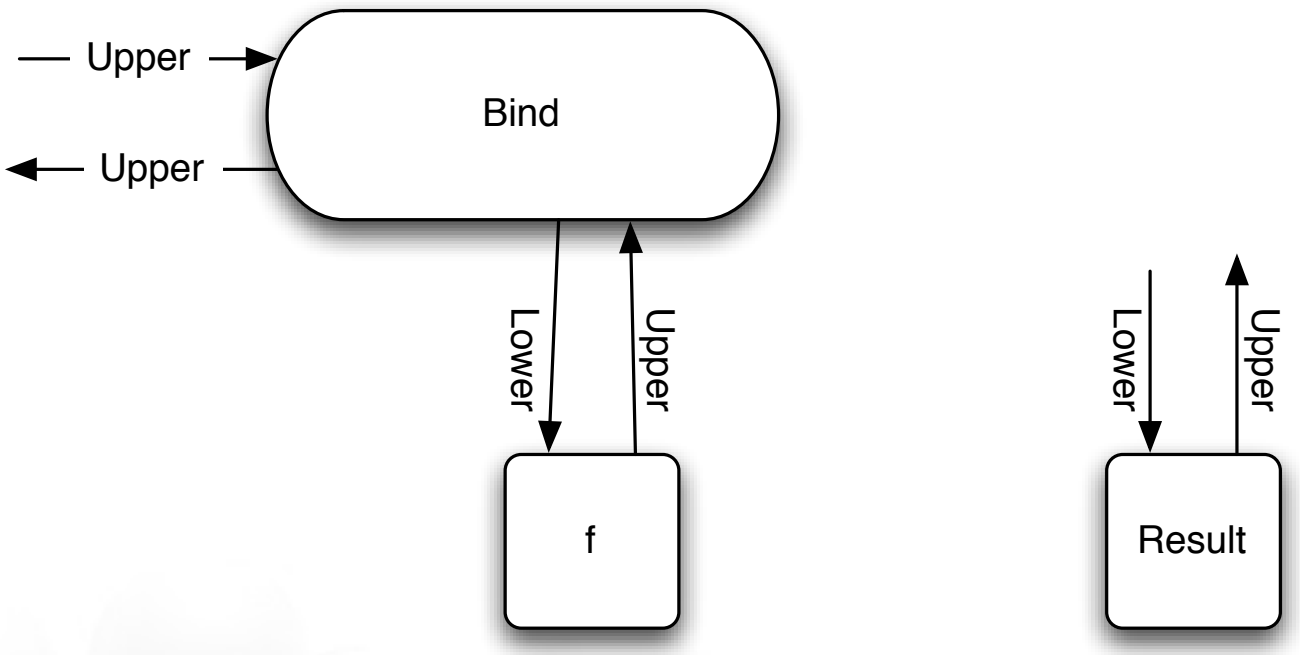
```ruby
def dot_result(n)
  "."*n
end

def dot_bind(d, &f)
  f.call(d.size())
end

def dot_add(da, db)
  dot_bind(da) do
    |a| dot_bind(db) do
      |b| dot_result(a+b)
    end
  end
end
```

```clojure
(defn dot-result [n]
  (apply str (repeat n ".")))

(defn dot-bind [d f]
  (f (count d)))

(defn add-dots [da db]
  (dot-bind da
    (fn [a] (dot-bind db
      (fn [b] (dot-result
        (+ a b)))))))
```

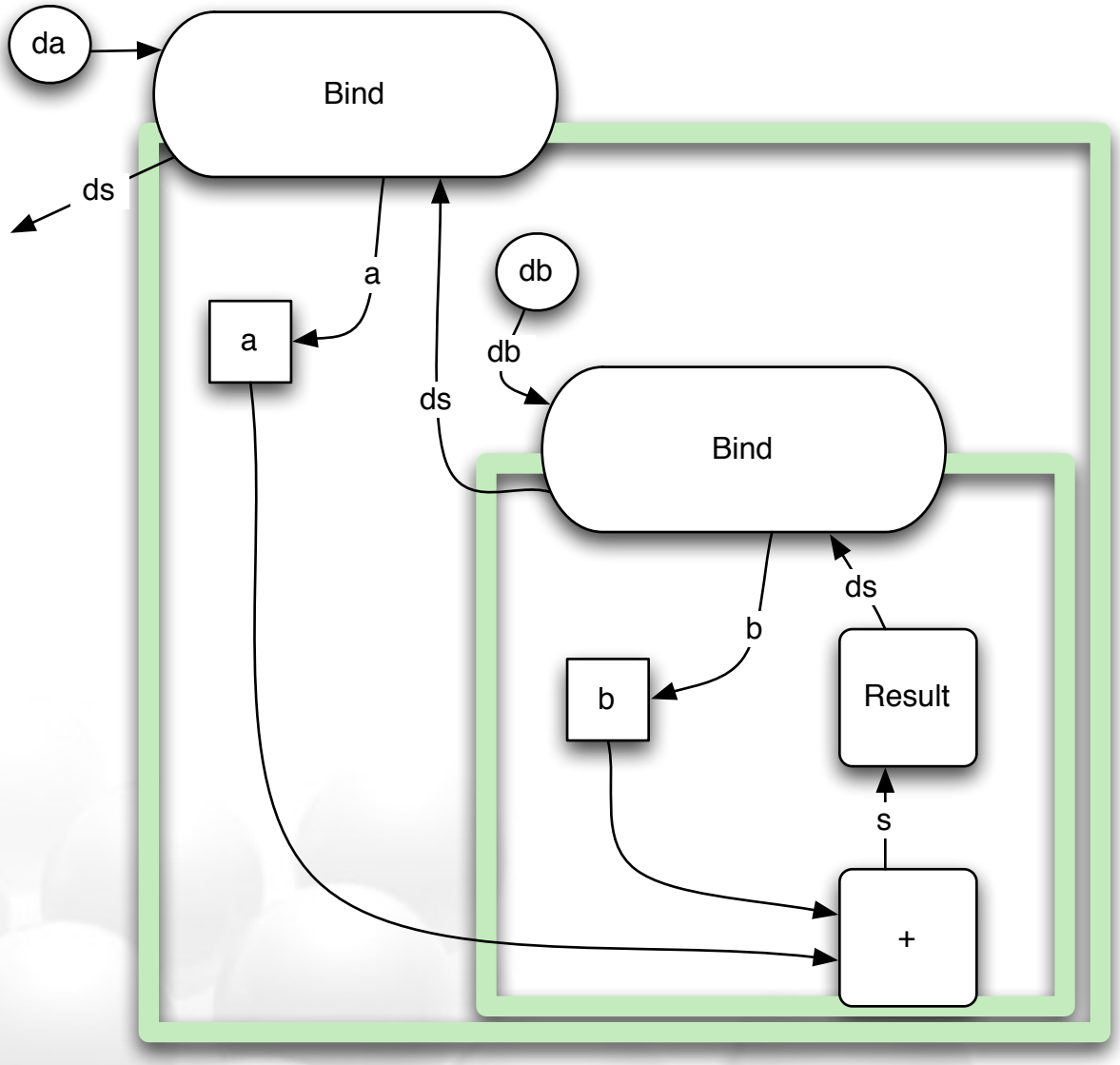
Series vs Parallel

- In the first (non-monadic) example the conversions took place in parallel.
  - ```
    (defn add-dots [da db]
      (let [a (dots-to-n da)
            b (dots-to-n db)]
        (n-to-dots (+ a b))))
    ```
- In the monadic example, the conversions took place in *series*.
  - ```
    (defn add-dots [da db]
      (dot-bind da
        (fn [a] (dot-bind db
          (fn [b] (dot-result
            (+ a b)))))))
    ```
- This will be important later, so keep it in mind.

Bind and Result



Monadic Dots. (ok, that's clear.)



Monads are inspectors

- Since the conversion from lower to upper takes place in series, the conversions have the ability to abort the operation.
  - e.g. if there are values in the upper form that cannot be handled by the lower form, then the series nature of the monad allows the conversions to abort before the lower functions get called.

- For example, let's say we have a lower form of "real number", and an upper form of "complex number".
  - Our monad converts complex numbers to real and invokes real operators.
  - But if the complex number has a non-zero imaginary component, then the monad must abort, because the real-number functions can't deal with imaginary numbers.

Complex

Nil checks.

- Many functions cannot deal with nil arguments. They throw NPEs.
  - So we define a lower form that cannot accept nil,
  - An upper form that can.
  - And a monad that adapts the two and aborts the operation if there is a nil.
  - This is the Maybe-Monad.

No Nil : Maybe Monad

The real power of being in series.

- Since the conversion takes place in series...
  - Each converter can determine how many *times* it is applied.
  - The maybe-monad applies one or zero times.
  - But what about 2, 3, 4?

- Imagine two forms:
  - a lower form of “number
  - An upper form of “list of numbers”.
  - What does [1 2 3] + [4 5 6] mean?
    - it could mean: [5 6 7 6 7 8 7 8 9]
  - In order for our monad to convert from a list to a number, it must invoke the conversion n times.
    - Given two arguments in list form, the lower function will be called n*m times.
- This is the sequence monad.

Lists: The Sequence Monad

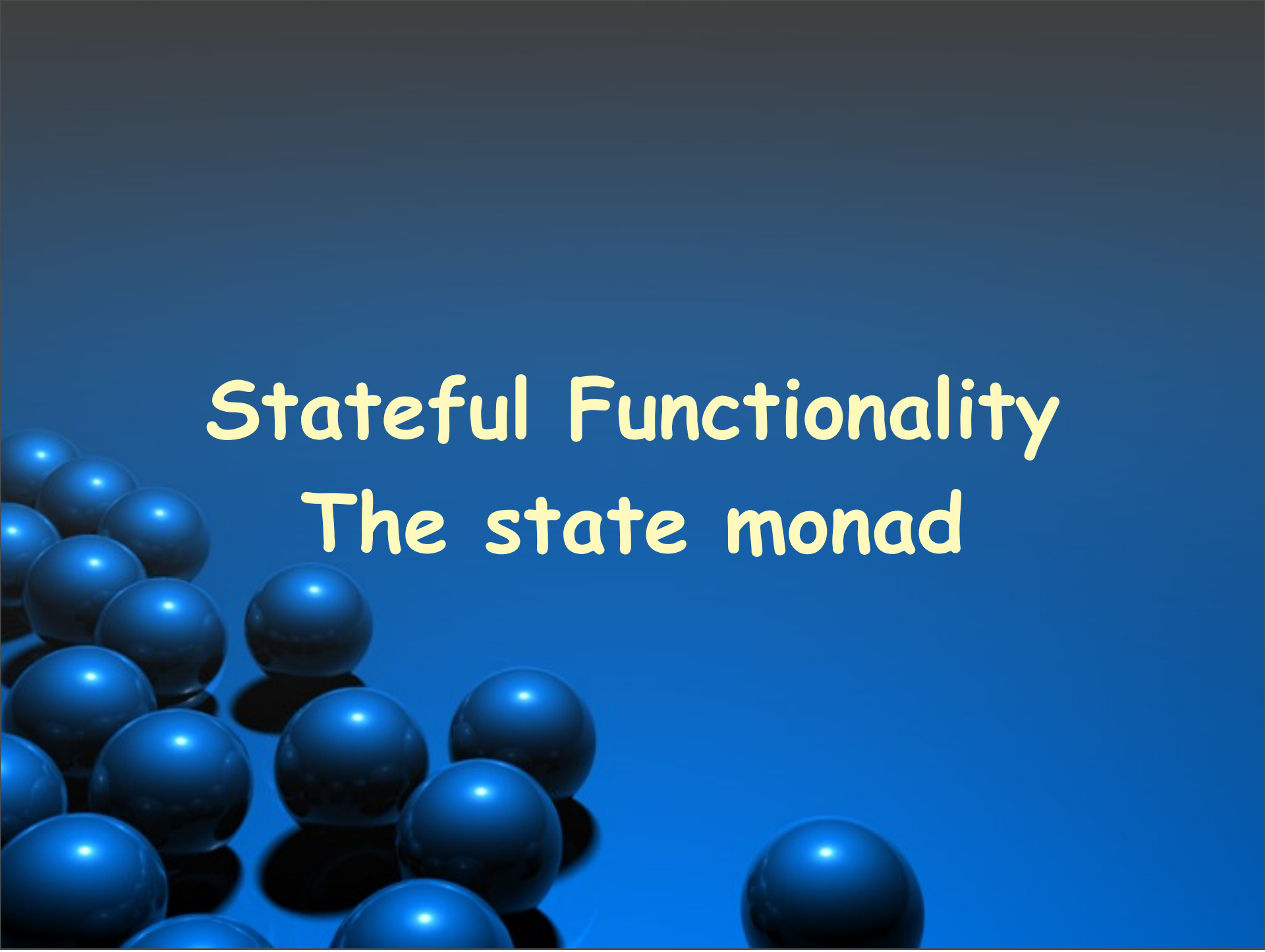
Probabilities

- Lower form: An event.
  - Represented by a token: e.g. 3, “bob”, :tic.
- Upper form: A probability distribution
  - Represented as a map: {:heads 1/2, :tails 1/2}
- The monad
  - generates events from distributions,
  - applies functions to the events to generate other events.
  - generates new distributions from those events.
- e.g. What is the probability distribution for rolling two dice?
  - use the distribution for one die.
  - use the monad to roll the die twice and generate the events.
  - add the two events together.
  - the monad will generate a new distribution.

Dice: The Distribution Monad

State

- Functional programs are stateless.
  - How do you simulate state in stateless code?
  - You pass the state to each function and have it return the new state.
  - You call the functions in *series*! because the output state of one is the input state of the next.
- Lower form: A function.
- Upper form: A function that takes and returns state.
- Result:
  - In the context of the monad our functions appear stateful.
  - Outside the monad we see that the operation was functional and not stateful.


Stateful Functionality
The state monad

fin

- unclebob @ objectmentor.com
- fitnesse.org
- cleancodeproject.com