1

1 2

1 2 3

1 2 3 4

1 2 3 4 5

1 2 3 4 5 6

LOG OF JENNY E

From LORD HOWE to RAPA ITI Date TUES 16/4 1963

HOUR	LOG	DISTANCE MADE	COURSE GOOD	WIND DIRECTION AND FORCE	BAROMETER	LEEWAY	REMARKS
A.M. 1	299.3	5.0	T56	SW F5-6	29.9	NIL	HEAVIEY SEAS But Vessel stayed
2	305.	5.7	T56	S.W.F 5-6	29.9	NIL.	On yet as we are running before
3	308	3.0	T56	S.W. 5-6	29.9	NIL	Which corresponds with Your Log
4	311	3.0	T56	S.W. 5-6	29.9	NIL	20 Knots of Breaking over First 100
5	319.5	8.5	T56	SW 5-6	29.9	NIL	Very uncomfutbal down below
6	330	8.5	T56	SW 8.	29.9	NIL.	decks.
7	337.4	7.4	T56	SW 7	29.95	NIL	SEAS HEAVY HUGE SWELLS
8	344.8	7.4	T56	SW 7	29.95	NIL	
9	350.4	6.4	T56	SW 7	29.95	NIL	30 FT SWELLS CREVIS TO PEAK
10	357.4	6.4	T56	SW 7	29.95	NIL	
11	404.4	7.0	T56	SW 7	29.95	NIL	
N. 12	411.2	6.8	T56	SW 7	29.95	NIL.	
P.M. 1	416.9	5.7	T56	SW 6	29.9	NIL	SEAS RECEDING
2	422.7	5.7	T56	S.W 4-5	29.9	NIL	" "
3	427.7	5.0	T56	S S 6	29.9	NIL	
4	432.7	5.0	T56	SES 6	29.9	NIL	
5	437.7	5.0	T56	SES 6	29.95	NIL	
6	441.1	3.4	T56	SES 6	30.0	NIL	VERY NEARLY TREACHING
7	440	2.0	T56	SES 6	30.1	NIL	HER MAXIMUM HULL SPEED
8	449	9.0	T56	SES 6	30.1	NIL	THIS IS HER BEST POINT OF
9	458	9.0	T56	SES-6	30.1	NIL	SAILING WITH THE WIND JUST
10	467.9	9.9	T56	SE 6	30.1	NIL	ABAFT OF BEAM 9 KNOTS 1-OR
11	476.2	8.3	T56	SE 6	30.1	NIL	THE HOUR
81 12	482.6	7.3	T56	SE 6	30.1	NIL	MODERATE SEAS

	LATITUDE	LONGITUDE
A.M.		
NOON	28° 65' S	171° 40'
12 MN. D.R. P.M.	28° 35' S	169° 50

D.R. POSITION

DAY'S RUN 140.2 Nautical Miles

CRUISE RUN 341.2 Nautical Miles

ENGINE USED ..... Hrs. ..... Mins.

FUEL—IN GALLONS

IN HAND	RECEIVED	CONSUMED	REMAINING
37	120	NIL	37

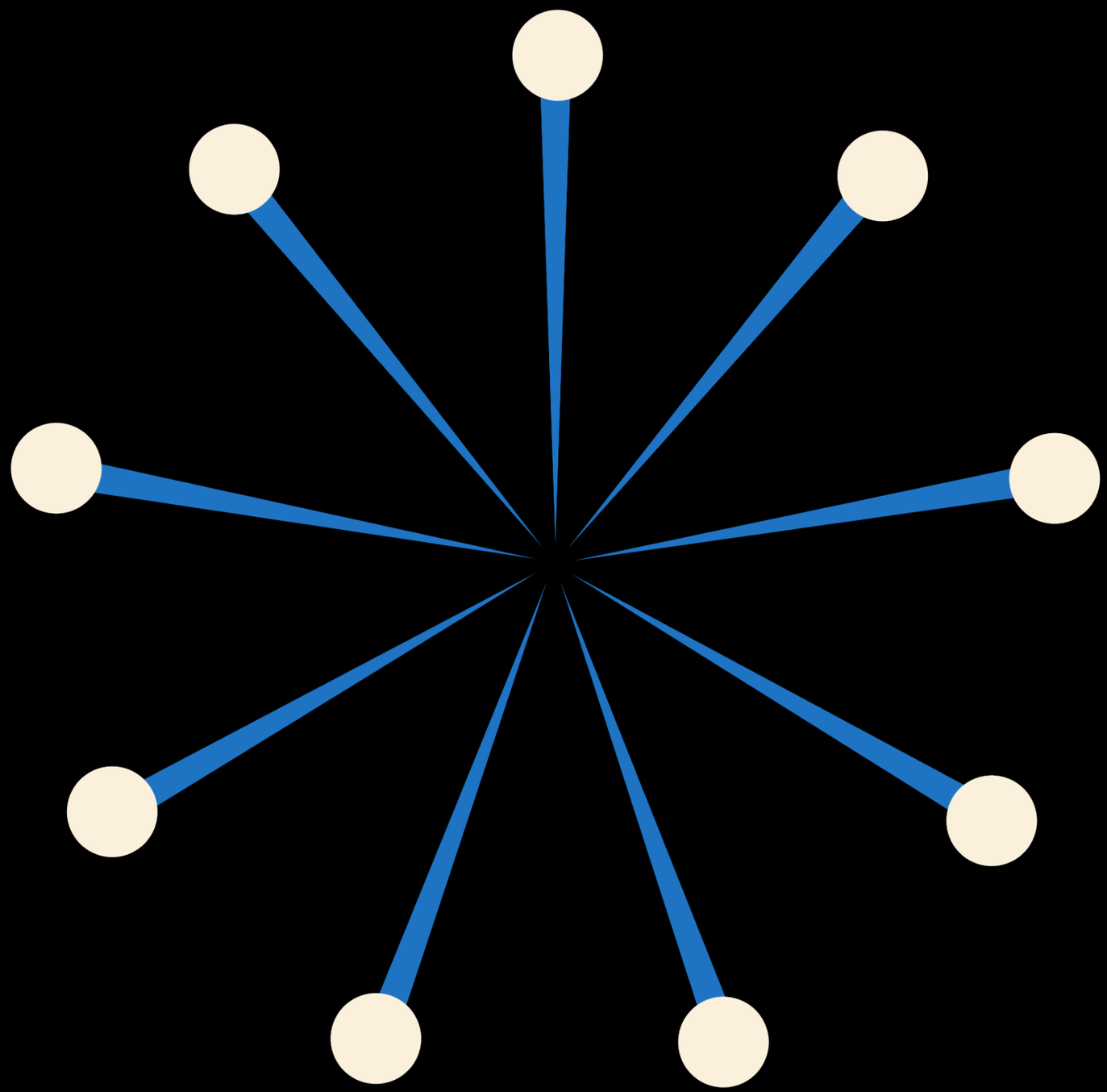
- Sum up all the rocks for the year
- Average # of rocks per day
- Biggest week
- Smallest month

What makes numbers so useful for modeling piles of rocks?

All I needed for FP I learned in High School Algebra

Eric Normand

PurelyFunctional.tv



Clojure SYNC

4

a

4

a

+

2

b

6

a + b

2

b

2

b

+

4

a

6

b + a

For combining piles of rocks, *order doesn't matter*

```
(+ a b)
```

```
(+ a b)
(+ b a)
```

```
(= (+ a b)
   (+ b a))
```

```
(= (f a b)
   (f b a))
```

Commutativity

Commutativity

Order doesn't matter

```
(= (f a b)
   (f b a))
```

```
(= (f a b)
   (f b a))
```

```
(prop/for-all [a gen
               b gen]
  (= (f a b)
     (f b a)))
```

```
(prop/for-all [a gen/map
               b gen/map]
  (= (merge a b)
     (merge b a)))
```

```
(prop/for-all [a gen-map1
               b gen-map2]
  (= (merge a b)
     (merge b a)))
```

5 1 3

a + b + c

6

(a + b)

+

3

c

9

a + b + c

5 1 3

a + b + c

5

a

+

4

(b + c)

9

a + b + c

Grouping doesn’t matter

a b c

a b c

(+ a b) c

a b c

```
(+ a   b)  c
a (+ b   c)
```

```
(+ (+ a   b)  c)
(+ a (+ b   c))
```

```
(+ (+ a b) c)
```

```
(+ a (+ b c))
```

```
(= (+ (+ a b) c)
   (+ a (+ b c)))
```

```
(= (f (f a b) c)
   (f a (f b c)))
```

Associativity

Associativity

Grouping doesn't **matter**

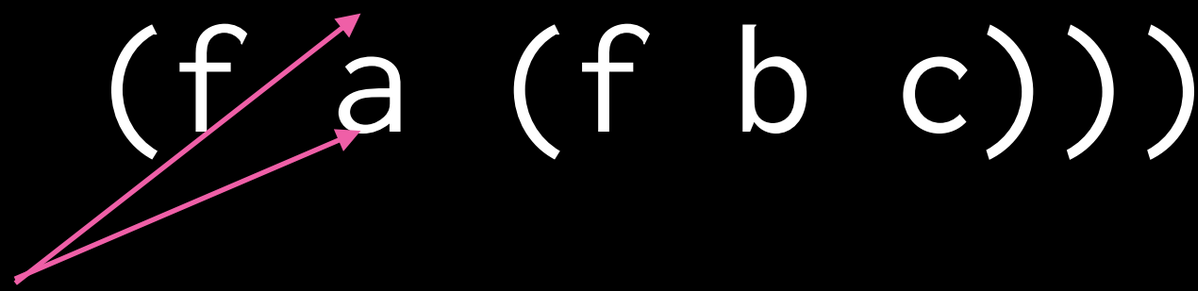
```
(= (f (f a b) c)
   (f a (f b c)))
```

Types

```
(= (f (f a b) c)
   (f a (f b c)))
```

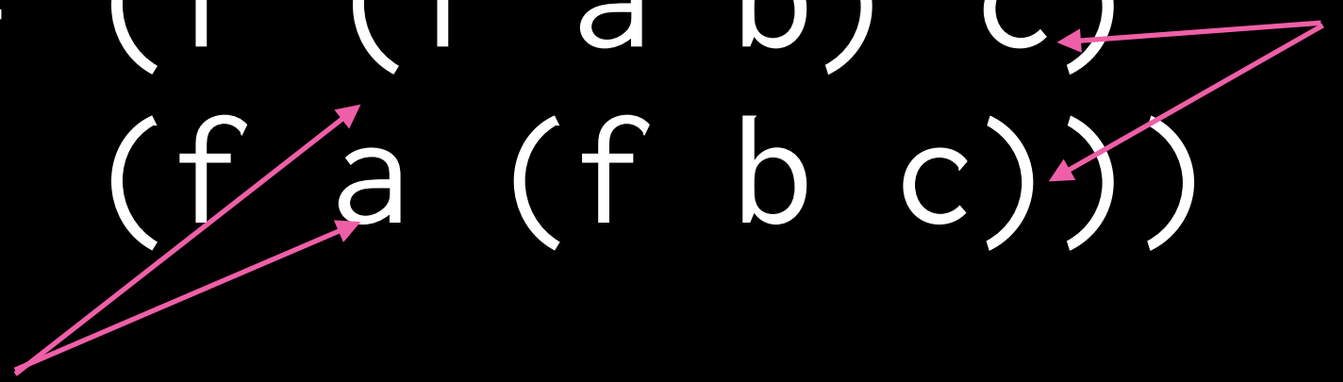
Types

```
(= (f (f a b) c)
   (f a (f b c)))
```



Types

```
(= (f (f a b) c)
   (f a (f b c)))
```



Types

```
(= (f (f a b) c)
   (f a (f b c)))
```

return value of f and its two
arguments need to be the same type

Whole Values

Whole Values

Combining two piles makes a new pile

Whole Values

Combining two piles makes a new pile

Concatenating two lists makes a new list

Whole Values

Combining two piles makes a new pile

Concatenating two lists makes a new list

Merging two hash maps makes a new hash map

Commutativity	Associativity
Order doesn’t matter	Grouping doesn’t matter
`(= (f a b)` `   (f b a))`	`(= (f (f a b) c)` `   (f a (f b c)))`

```clojure
(defn average [a b]
  (/ (+ a b) 2))
```

Order doesn't matter

```
(= (average a b) (average b a))
a = 10, b = 4
```

Order doesn't matter

```
(= (average a b) (average b a))
a = 10, b = 4

(average 10 4)  => 7
```

Order doesn't matter

```
(= (average a b) (average b a))
a = 10, b = 4

(average 10 4)   => 7
(average 4 10)   => 7
```

Does grouping matter?

Does grouping matter?

```
(= (average (average a b) c)
   (average a (average b c)))
a = 10, b = 4, c = 6
```

Does grouping matter?

```
(= (average (average a b) c)
   (average a (average b c)))
a = 10, b = 4, c = 6

(average 10 4)  => 7
```

Does grouping matter?

```
(= (average (average a b) c)
   (average a (average b c)))
a = 10, b = 4, c = 6

(average 10 4)  => 7
(average 7  6)  => 6.5
```

Does grouping matter?

```
(= (average (average a b) c)
   (average a (average b c)))
a = 10, b = 4, c = 6

(average 10 4)   => 7
(average 7  6)   => 6.5

(average 4  6)   => 5
```

Does grouping matter?

```
(= (average (average a b) c)
   (average a (average b c)))
a = 10, b = 4, c = 6

(average 10 4)   => 7
(average 7  6)   => 6.5

(average 4  6)   => 5
(average 10 5)   => 7.5
```

Does grouping matter?

```
(= (average (average a b) c)
   (average a (average b c)))
a = 10, b = 4, c = 6

(average 10 4)   => 7
(average 7  6)   => 6.5

(average 4  6)   => 5
(average 10 5)   => 7.5
```

```clojure
(defn average [a b]
  (/ (+ a b) 2))
```

```
function average(numbers) {
  var sum   = 0;
  var count = 0;
  for(i = 0; i < numbers.length; i++) {
    sum   += numbers[i];
    count += 1;
  }
  if(count === 0) {
    return null;
  }
  return sum / count;
}
```

```
function average(numbers) {
  var sum   = 0;
  var count = 0;
  for(i = 0; i < numbers.length; i++) {
    sum   += numbers[i];
    count += 1;
  }
  if(count === 0) {
    return null;
  }
  return sum / count;
}
```

```
(defn combine [                                  ]
                                    )
```

```
(defn combine [                                  ]
                                    )
```

```
(defn combine [[sum1 count1] [sum2 count2]]
                                        )
```

```
(defn combine [[sum1 count1] [sum2 count2]]
  [                                 ])
```

```
(defn combine [[sum1 count1] [sum2 count2]]
  [(+ sum1 sum2) (+ count1 count2)])
```

```
(defn combine [[sum1 count1] [sum2 count2]]
  [(+ sum1 sum2) (+ count1 count2)])
```

```clojure
(defn combine [[sum1 count1] [sum2 count2]]
  [(+ sum1 sum2) (+ count1 count2)])

(defn ->average [number]
  [number 1])
```

```
(defn combine [[sum1 count1] [sum2 count2]]
  [(+ sum1 sum2) (+ count1 count2)])

(defn ->average [number]
  [number 1])
```

```
(defn average [numbers]
  (reduce combine   (map ->average numbers)))
```

```
(defn combine [[sum1 count1] [sum2 count2]]
  [(+ sum1 sum2) (+ count1 count2)])

(defn ->average [number]
  [number 1])
```

```
(defn average [numbers]
  (reduce combine ? (map ->average numbers)))
```

Where do you start a computation?

```
(+ a 0)
```

```
(= (+ a 0) a)
```

$$(= (f\ a\ i_f)\ a)$$

Identity value

Identity value

Where to start

$$(= (f\ a\ i_f)\ a)$$

```
(defn combine [[sum1 count1] [sum2 count2]]
  [(+ sum1 sum2) (+ count1 count2)])

(defn ->average [number]
  [number 1])
```

```
(defn average [numbers]
  (reduce combine ? (map ->average numbers)))
```

```
function average(numbers) {
  var sum   = 0;
  var count = 0;
  for(i = 0; i < numbers.length; i++) {
    sum   += numbers[i];
    count += 1;
  }
  if(count === 0) {
    return null;
  }
  return sum / count;
}
```

```clojure
(defn combine [[sum1 count1] [sum2 count2]]
  [(+ sum1 sum2) (+ count1 count2)])

(defn ->average [number]
  [number 1])

(defn average [numbers]
  (reduce combine [0 0] (map ->average numbers)))
```

Monoid

Monoid

Associative and has identity value

Commutative	Order doesn't matter	`(= (f a b)` `  (f b a))`
Associative	Grouping doesn't matter	`(= (f (f a b) c)` `  (f a (f b c)))`
Identity value	Where to start	`(= (f a i) a)`
Monoid	Whole value with starting/empty value	`Associative + Identity value`


EXIT
ENTRANCE

Moving into a new space, doing a calculation, then moving back

Going back and forth matters

send

"{:name \"Eric\",…}"

"{:name \"Eric\",…}"

pr-str

read-string

{:name “Eric”
 :birthday #inst “1981-07-18”}

{:name “Eric”
 :birthday #inst “1981-07-18”}

```
(pr-str a)
```

```
(read-string
 (pr-str a))
```

```
(= (read-string
     (pr-str a))
   a)
```

```
(= (g (f a)) a)
```

Inverse

g is the inverse of f

Inverse

Going back and forth

matters

```
(= (g (f a)) a)
```

```
(defn combine [[sum1 count1] [sum2 count2]]
  [(+ sum1 sum2) (+ count1 count2)])

(defn ->average [number]
  [number 1])

(defn average [numbers]
  (reduce combine [0 0] (map ->average numbers)))
```

```clojure
(defn combine [[sum1 count1] [sum2 count2]]
  [(+ sum1 sum2) (+ count1 count2)])

(defn ->average [number]
  [number 1])

(defn average-> [[sum count]]
  (/ sum count))

(defn average [numbers]
  (->> numbers
    (map ->average)
    (reduce combine [0 0])
    average->))
```

```clojure
(defn combine [[sum1 count1] [sum2 count2]]
  [(+ sum1 sum2) (+ count1 count2)])

(defn ->average [number]
  [number 1])

(defn average-> [[sum count]]
  (/ sum count))

(defn average [numbers]
  (->> numbers
    (map ->average)
    (reduce combine [0 0])
    average->))
```

Duplicates don’t matter

```
(-> m
 (assoc :a “hello”))
```

```
(-> m
 (assoc :a “hello”)
 (assoc :a “hello”))
```

```
(= (-> m
       (assoc :a “hello”)
       (assoc :a “hello”))
   (-> m
       (assoc :a “hello”)))
```

```
(=     (f a)
       (f a))
```

```
(= (f (f a))
   (f a))
```

Idempotence

Idempotence

Duplicates don't matter

```
(= (f (f a))
   (f a))
```

```
(def button-state (atom {}))
```

```clojure
(def button-state (atom {}))

(defn press! [button-id]
  (swap! button-state assoc button-id true))
```

```clojure
(def button-state (atom {}))

(defn press! [button-id]
  (swap! button-state assoc button-id true))
```

…

```clojure
(def button-state (atom {}))

(defn press! [button-id]
  (swap! button-state assoc button-id true))

…

(press! :3rd-floor-north-up)
```

```
(def button-state (atom {}))

(defn press! [button-id]
  (swap! button-state assoc button-id true))

…
(press! :3rd-floor-north-up)

(press! :3rd-floor-north-up)
```

```
(def button-state (atom {}))

(defn press! [button-id]
  (swap! button-state assoc button-id true))

…
(press! :3rd-floor-north-up)

(press! :3rd-floor-north-up)

(press! :3rd-floor-north-up)
```

Commutative	Order doesn’t matter	`(= (f a b) (f b a))`
Associative	Grouping doesn’t matter	`(= (f (f a b) c)` `(f a (f b c)))`
Identity value	Where to start	`(= (f a i) a)`
Inverse	Going back and forth	`(= (g (f a)) a)`
Idempotence	Duplicates don’t matter	`(= (f (f a)) (f a))`
Zero value	When to stop	`(= (f a z) z)`

Structure Preservation	Rearranging work	`(= (map identity a) a)`
		
		`(= (map (comp f g) a)`
		`(map f (map g a)))`

Translating properties is what allows us to program

You can add these properties to operations that don't have them naturally

You can discover your own properties or tweak the well-known ones to suit your needs

Challenges of distributed systems

Challenges of distributed systems

- Messages are delivered out of order

Challenges of distributed systems

- Messages are delivered out of order
- Messages are delivered one or more times

Challenges of distributed systems

- Messages are delivered out of order
- Messages are delivered one or more times
- Sending tasks and combining answers

Challenges of distributed systems

- Messages are delivered out of order
- Messages are delivered one or more times
- Sending tasks and combining answers
- Where do workers start?

Challenges of distributed systems

- Messages are delivered out of order
- Messages are delivered one or more times
- Sending tasks and combining answers
- Where do workers start?
- Serialization/deserialization

Algebraic properties make great test.check properties

```
(= (f a b)
   (f b a))
```

```
(prop/for-all [a gen
               b gen]
  (= (f a b)
     (f b a)))
```

```
(prop/for-all [a gen/int
               b gen/int]
  (= (* a b)
     (* b a)))
```