CS492: Probabilistic Programming

Basics of Clojure and tiny bit of Anglican

Hongseok Yang
KAIST

Does anyone use Clojure, Scheme, or Lisp?

Does anyone use Clojure, Scheme, or Lisp?

What are the cons and pros of such a lang.?

Clojure

- Re-design of Scheme for Java virtual machine, with concurrency in mind.
- Untyped.
- Highly expressive.
- Cousin language for Anglican, the probabilistic programming language used in this course.

Learning outcome

- Can write simple Clojure programs with recursion, loop, sequence, and map.
- Can write simple Anglican programs with no conditioning, and perform inference.
- All by copy-past-modify programming.

Clojure in a nutshell

1. Prefix instead of infix notation:

   `(+ 3 3)`, not 3+3

2. Use let to bind variables to values.

```
(let [x (* 3 3) y (* 4 4)] (+ x y))
```

3. Anonymous function using fn:

```
(let [f (fn [x] (* x x))] (+ (f 3) (f 4)))
```

Clojure in a nutshell

1. Prefix instead of infix notation:

   `(+ 3 3)`, not 3+3

2. Use let to bind variables to values.

```
(let [x (* 3 3) y (* 4 4)] (+ x y))
```

3. Anonymous function using fn:

```
(let [f (fn [x] (* x x))] (+ (f 3) (f 4)))
```

[Q] Write a program that computes $1^2 + 2^2 + \ldots + 5^2$.

Clojure in a nutshell

1. Prefix instead of infix notation:

```
(let [f (fn [x] (* x x))]
  (+ (f 1)
     (f 2)
     (f 3)
     (f 4)
     (f 5)))
```

2. Use s.

```
(le - x y))
```

3. Anonymous function using fn:

```
(let [f (fn [x] (* x x))] (+ (f 3) (f 4)))
```

[Q] Write a program that computes $1^2 + 2^2 + \ldots + 5^2$.

Clojure in a nutshell

1. Prefix instead of infix notation:

```
(let [f (fn [x] (* x x))]
  (+ (f 1)
     (f 2)
     (f 3)
     (f 4)
     (f 5)))
```

2. Use s.

```
(le - x y))
```

3. Anonymous function using fn:

```
(let [f (fn [x] (* x x))] (+ (f 3) (f 4)))
```

[Q] Write a program that computes $1^2 + 2^2 + \ldots + \overset{n^2}{5^2}$.

Clojure in a nutshell

4. Separate function definition:

```
(defn f [n] …)
```

E.g.

```
(defn f [n]
  (if (= n 0)
    0
    (+ n (f (- n 1)))))

(println (f 10))
```

[Q] Write a program that computes $1^2 + 2^2 + \ldots + \overset{n^2}{\cancel{5^2}}$.

Clojure in a nutshell

4. Separate function definition:

```
(defn f [n] ...)
```

E.g.

```
(defn f [n]
  (if (= n 0)
    0
    (+ n (f (- n 1)))))

(println (f 10))
```

Recursion allowed

[Q] Write a program that computes $1^2 + 2^2 + \ldots + \overset{n^2}{\cancel{5^2}}$.

Clojure in a nutshell

4. Separate function definition:

```
(defn f [n] …)
```

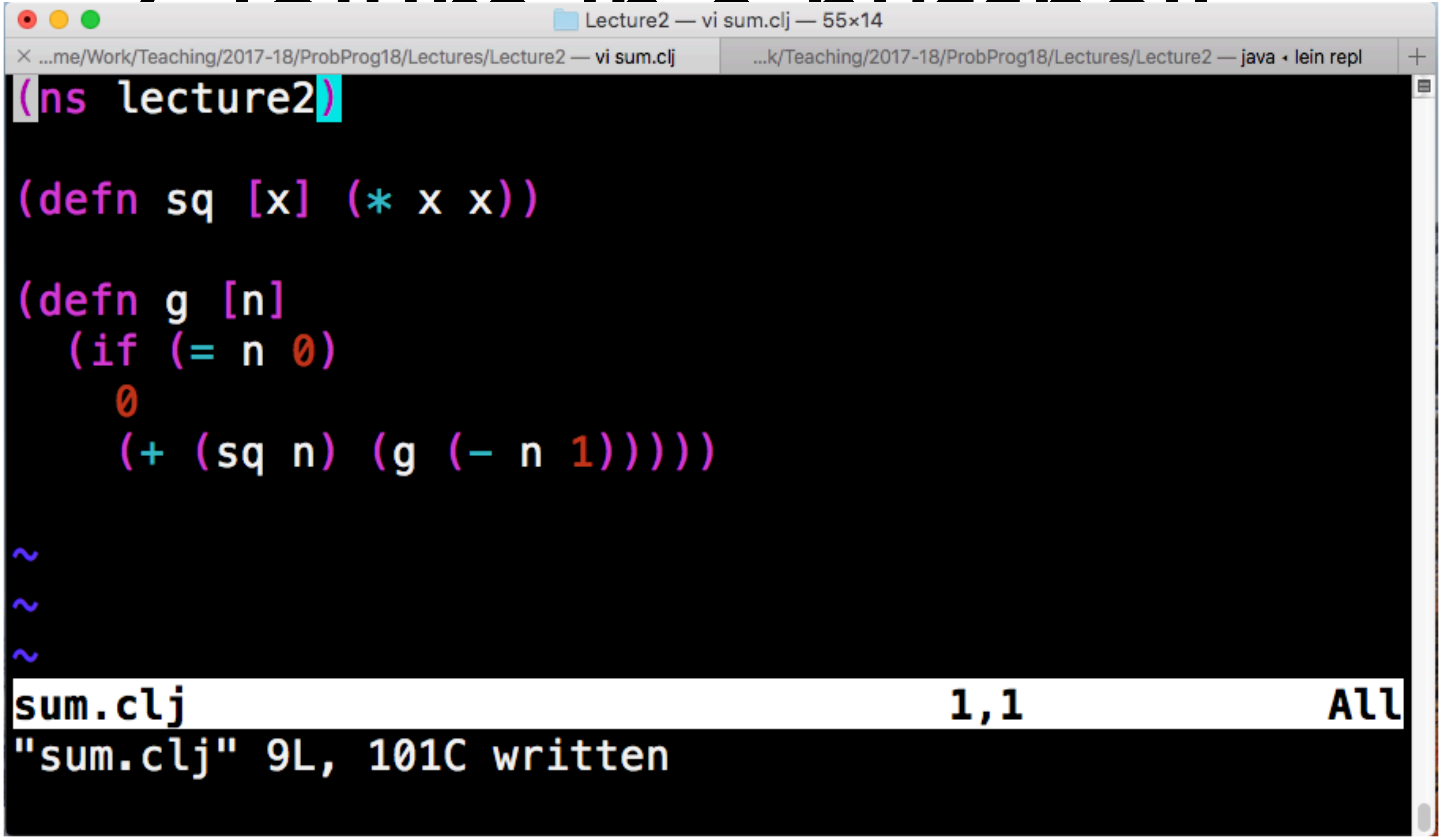
E.g.

```
(defn f [n]
  (if (= n 0)
    0
    (+ n (f (- n 1)))))

(println (f 10))
```

Recursion allowed

[Q] Write a program that computes $1^2 + 2^2 + \ldots + \overset{n^2}{\cancel{5^2}}$.

Clojure in a nutshell



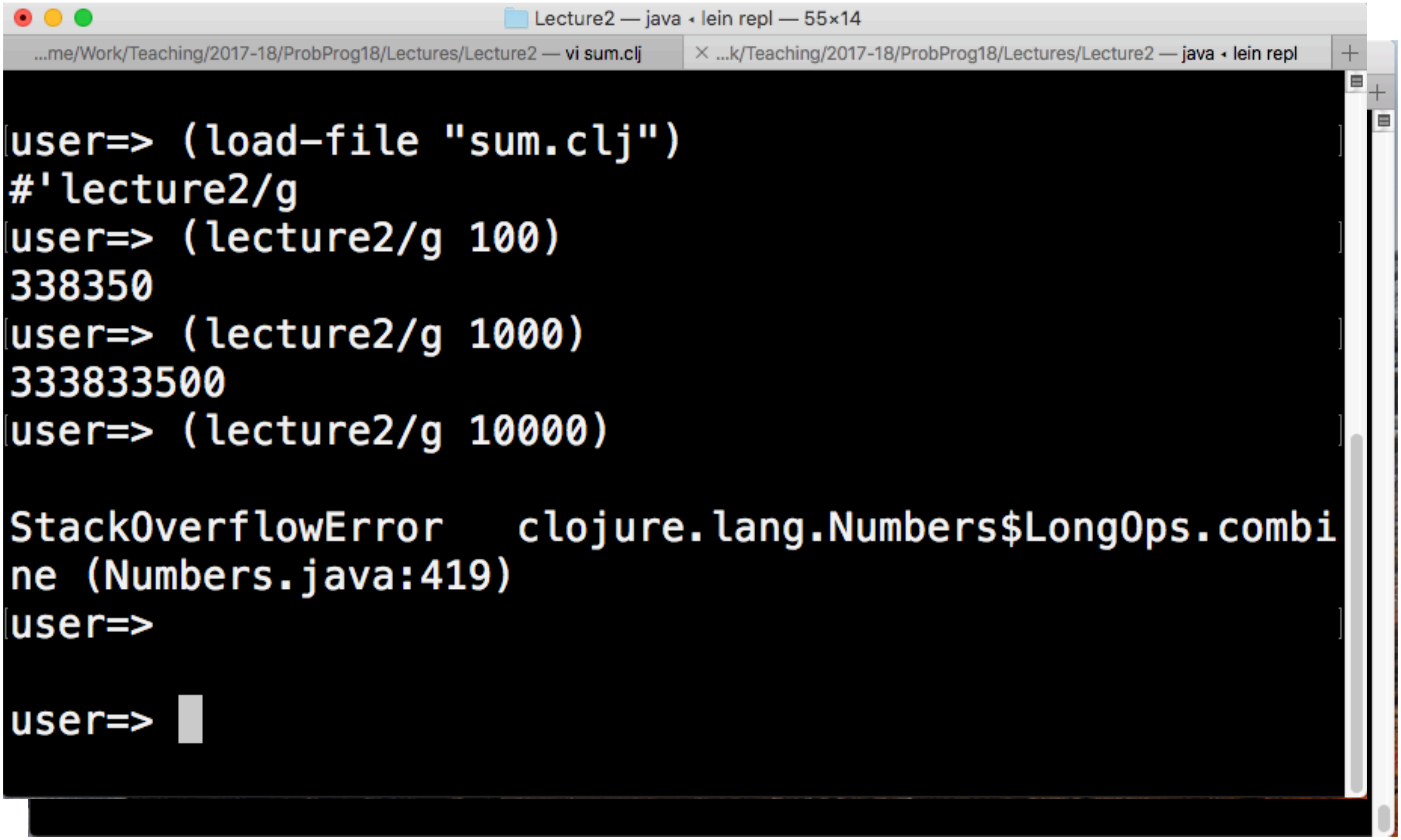
Lecture2 — vi sum.clj — 55×14

...me/Work/Teaching/2017-18/ProbProg18/Lectures/Lecture2 — vi sum.clj | ...k/Teaching/2017-18/ProbProg18/Lectures/Lecture2 — java ◂ lein repl

```
(ns lecture2)

(defn sq [x] (* x x))

(defn g [n]
  (if (= n 0)
    0
    (+ (sq n) (g (- n 1)))))

~
~
~
sum.clj                                     1,1            All
"sum.clj" 9L, 101C written
```

[Q] Write a program that computes $1^2 + 2^2 + \ldots + \overset{n^2}{\cancel{5^2}}$.

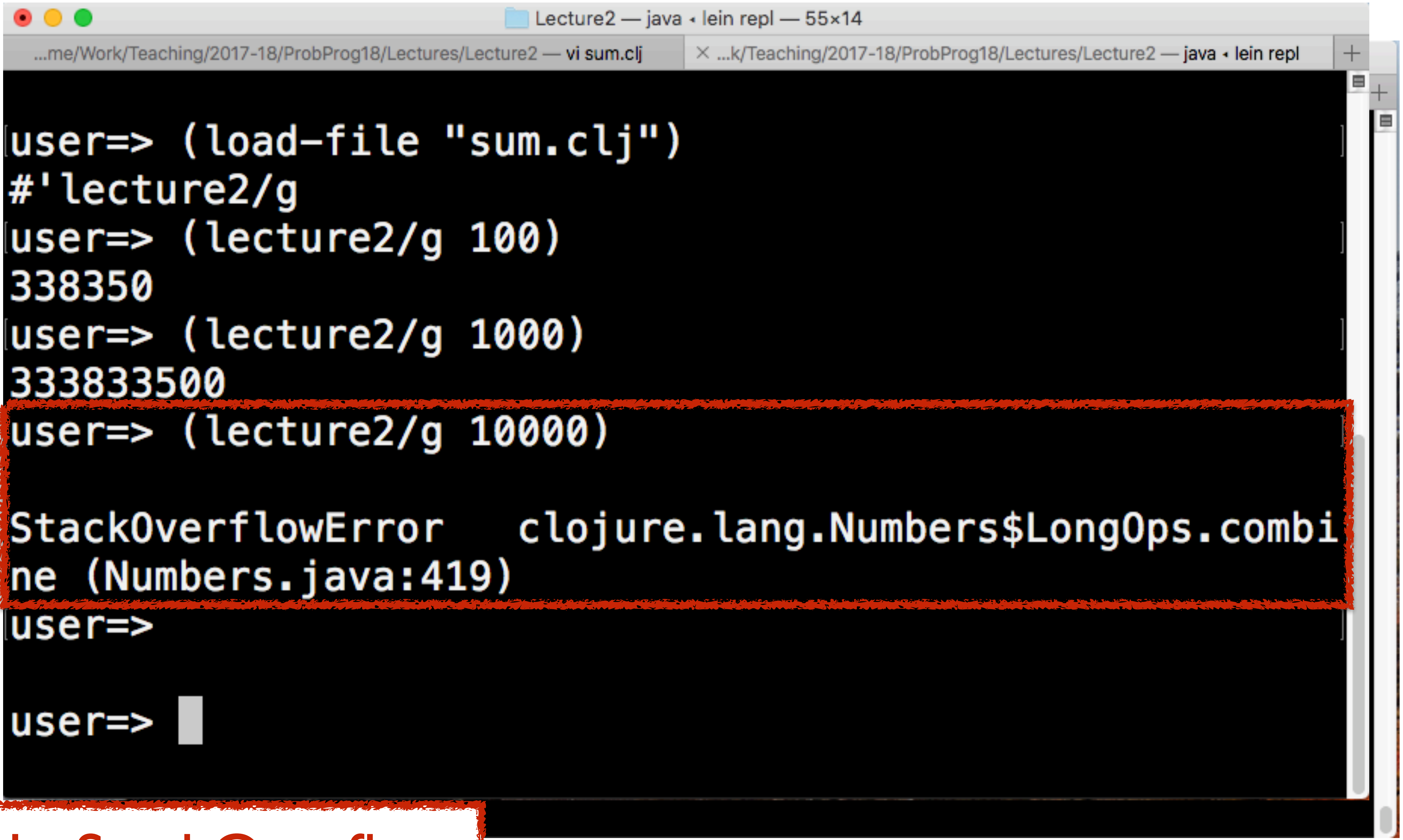
NB: I installed leiningen. Then, I ran "lein repl".


Lecture2 — java ◂ lein repl — 55×14
...me/Work/Teaching/2017-18/ProbProg18/Lectures/Lecture2 — vi sum.clj
...k/Teaching/2017-18/ProbProg18/Lectures/Lecture2 — java ◂ lein repl

```
user=> (load-file "sum.clj")
#'lecture2/g
user=> (lecture2/g 100)
338350
user=> (lecture2/g 1000)
333833500
user=> (lecture2/g 10000)

StackOverflowError   clojure.lang.Numbers$LongOps.combi
ne (Numbers.java:419)
user=>


user=> 
```

[Q] Write a program that computes $1^2 + 2^2 + \ldots + \overset{n^2}{\cancel{5^2}}$.

NB: I installed leiningen. Then, I ran “lein repl”.



Lecture2 — java ◂ lein repl — 55×14
...me/Work/Teaching/2017-18/ProbProg18/Lectures/Lecture2 — vi sum.clj
...k/Teaching/2017-18/ProbProg18/Lectures/Lecture2 — java ◂ lein repl

```
user=> (load-file "sum.clj")
#'lecture2/g
user=> (lecture2/g 100)
338350
user=> (lecture2/g 1000)
333833500
user=> (lecture2/g 10000)

StackOverflowError   clojure.lang.Numbers$LongOps.combi
ne (Numbers.java:419)
user=>

user=> 
```

No StackOverflow.

[Q] Write a program that computes $1^2 + 2^2 + \ldots + \cancel{5^2}$ n^2.

Clojure in a nutshell

5. Tail recursion.

[Q] Write a program that computes $1^2 + 2^2 + \ldots + \overset{n^2}{\cancel{5^2}}$.

Clojure in

No further work after recursive call.

5. Tail recursion.

[Q] Write a program that computes $1^2 + 2^2 + \ldots + \overset{n^2}{\cancel{5^2}}$.

Clojure in

No further work
after recursive call.

5. Tail recursion.

```
(defn f [n]
  (if (= n 0)
    0
    (+ n
       (f (- n 1)))))
```

Not tail recursive

[Q] Write a program that computes $1^2 + 2^2 + \ldots + \overset{n^2}{\cancel{5^2}}$.

Clojure in

No further work after recursive call.

5. Tail recursion. 1) Use accumulator.

```
(defn f [n]
  (if (= n 0)
    0
    (+ n
       (f (- n 1)))))
```

Not tail recursive

[Q] Write a program that computes $1^2 + 2^2 + \ldots + \overset{n^2}{\cancel{5^2}}$.

Clojure in

No further work
after recursive call.

5. Tail recursion. 1) Use accumulator.

```
(defn f [n]
  (if (= n 0)
    0
    (+ n
       (f (- n 1)))))
```

Not tail recursive

```
(defn g [n r]
  (if (= n 0)
    r
    (g (- n 1)
       (+ n r))))

(defn f [N] (g N 0))
```

accumulator r

[Q] Write a program that computes $1^2 + 2^2 + \ldots + \overset{n^2}{\cancel{5^2}}$.

Clojure in

No further work after recursive call.

5. Tail recursion. 1) Use accumulator. 2) Inform the compiler using loop and recur.

```
(defn f [n]
  (if (= n 0)
    0
    (+ n
       (f (- n 1)))))
```

Not tail recursive

```
(defn g [n r]
  (if (= n 0)
    r
    (g (- n 1)
       (+ n r))))

(defn f [N] (g N 0))
```

accumulator r

[Q] Write a program that computes $1^2 + 2^2 + \dots + \overset{n^2}{\cancel{5^2}}$.

Clojure in

No further work after recursive call.

5. Tail recursion. 1) Use accumulator. 2) Inform the compiler using loop and recur.

```
(defn f [n]
  (if (= n 0)
```

```
(defn f [N]
  (loop [n N r 0]
    (if (= n 0)
      r
      (recur (- n 1)
             (+ n r)))))
```

```
(defn g [n r]
  (if (= n 0)
    r
    (g (- n 1)
       (+ n r))))

(defn f [N] (g N 0))
```

accumulator r

[Q] Write a program that computes $1^2 + 2^2 + \ldots + \overset{n^2}{\cancel{5^2}}$.

Clojure in

No further work after recursive call.

5. Tail recursion. 1) Use accumulator. 2) Inform the compiler using loop and recur.

```
(defn f [n]
  (if (= n 0)
```

```
(defn f [N]
  (loop [n N r 0]
    (if (= n 0)
      r
      (recur (- n 1)
             (+ n r)))))
```

```
(defn g [n r]
  (if (= n 0)
    r
    (g (- n 1)
       (+ n r))))

(defn f [N] (g N 0))
```

accumulator r

[Q] Write a program that computes $1^2 + 2^2 + \ldots + \overset{n^2}{\cancel{5^2}}$.

Clojure in

No further work after recursive call.

5. Tail recursion. 1) Use accumulator. 2) Inform the compiler using loop and recur.

```
(defn f [n]
  (if (= n 0)
```

```
(defn f [N]
  (loop [n N r 0]
    (if (= n 0)
      r
      (recur (- n 1)
             (+ n r)))))
```

```
(defn g [n r]
  (if (= n 0)
    r
    (g (- n 1)
       (+ n r))))

(defn f [N] (g N 0))
```

accumulator r

No StackOverflow.

[Q] Write a program that computes $1^2 + 2^2 + \ldots + \overset{n^2}{\cancel{5^2}}$.

Clojure in

No further work
after recursive call.

5. Tail recursion. 1) Use accumulator. 2) Inform the compiler using loop and recur.

```
(defn f [n]
  (if (= n 0)
```

```
(defn f [N]
  (loop [n N r 0]
    (if (= n 0)
      r
      (recur (- n 1)
             (+ n r)))))
```

```
(defn g [n r]
  (if (= n 0)
    r
    (g (- n 1)
       (+ n r))))

(defn f [N] (g N 0))
```

accumulator r

[Q] Write a program that computes $1^2 + 2^2 + \ldots + \overset{n^2}{\cancel{5^2}}$.

Clojure in

No further work after recursive call.

5. Tail recursion. 1) Use accumulator. 2) Inform the compiler using loop and recur.

```
(defn f [n]
  (if (= n 0)
```

```
(defn f [N]
  (loop [n N r 0]
    (if (= n 0)
      r
      (recur (- n 1)
             (+ n r))))))
```

```
(defn g [n r]
  (if (= n 0)
    r
    (g (- n 1)
       (+ n r))))

(defn f [N] (g N 0))
```

accumulator r

[Q] Write a program that computes $1^2 + 2^2 + \ldots + \cancel{5^2}\overset{n^2}{}$.

Clojure in

No further work after recursive call.

5. Tail recursion. 1) Use accumulator. 2) Inform the compiler using loop and recur.

```
(defn f [n]
  (if (= n 0)
```

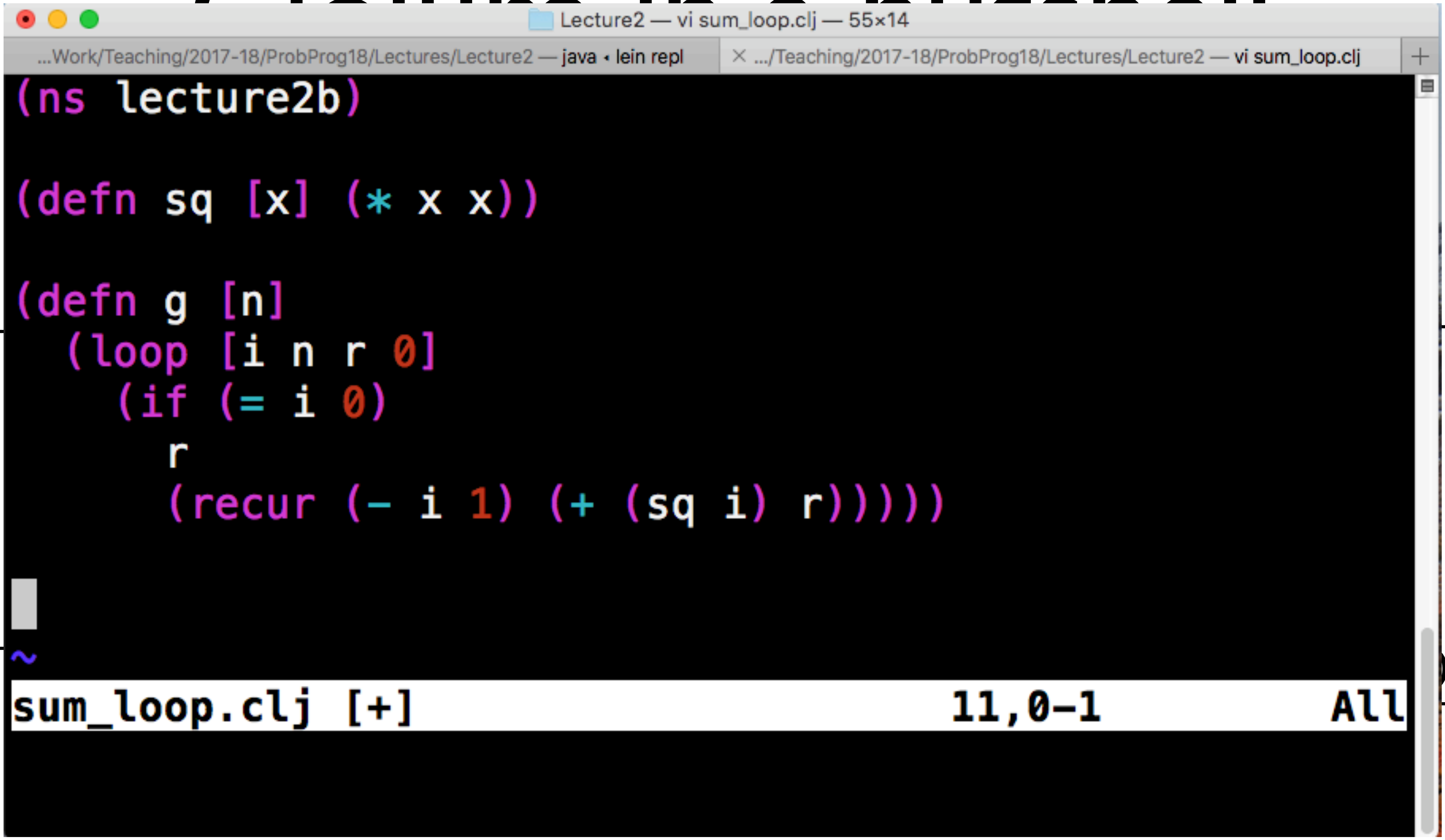
```
(defn f [N]
  (loop [n N r 0]
    (if (= n 0)
      r
      (recur (- n 1)
             (+ n r)))))
```

```
(defn g [n r]
  (if (= n 0)
    r
    (g (- n 1)
       (+ n r))))

(defn f [N] (g N 0))
```

accumulator r

[Q] Write a program that computes $1^2 + 2^2 + \ldots + \overset{n^2}{\cancel{5^2}}$.

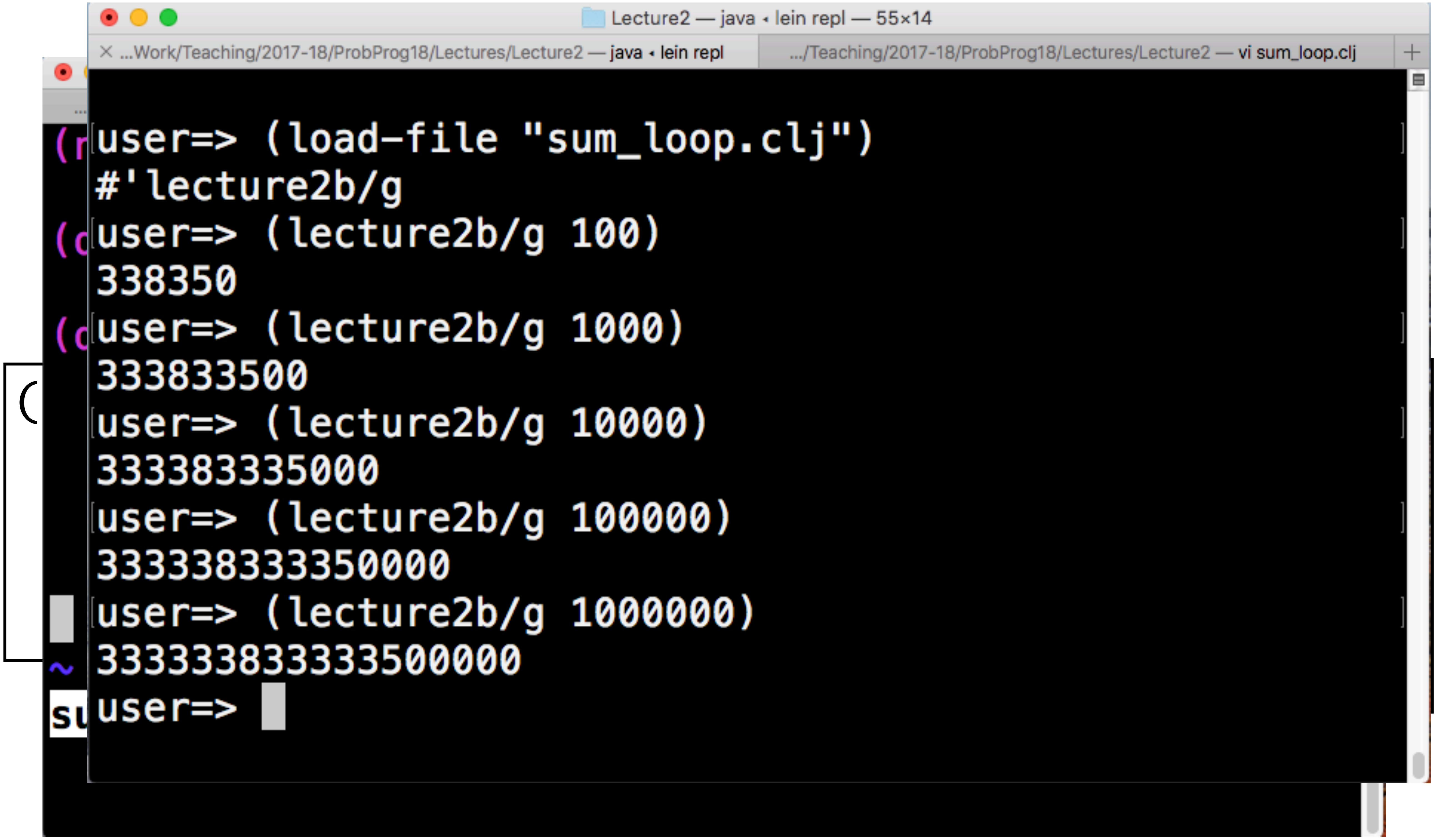
Clojure in a nutshell


Lecture2 — vi sum_loop.clj — 55×14
...Work/Teaching/2017-18/ProbProg18/Lectures/Lecture2 — java ◂ lein repl
.../Teaching/2017-18/ProbProg18/Lectures/Lecture2 — vi sum_loop.clj

```
(ns lecture2b)

(defn sq [x] (* x x))

(defn g [n]
  (loop [i n r 0]
    (if (= i 0)
      r
      (recur (- i 1) (+ (sq i) r)))))


~
sum_loop.clj [+]                          11,0-1            All
```

[Q] Write a program that computes $1^2 + 2^2 + \ldots + \overset{n^2}{\cancel{5^2}}$.

NB: I installed leiningen. Then, I ran “lein repl”.



Lecture2 — java ◂ lein repl — 55×14

...Work/Teaching/2017-18/ProbProg18/Lectures/Lecture2 — java ◂ lein repl

.../Teaching/2017-18/ProbProg18/Lectures/Lecture2 — vi sum_loop.clj

```
user=> (load-file "sum_loop.clj")
#'lecture2b/g
user=> (lecture2b/g 100)
338350
user=> (lecture2b/g 1000)
333833500
user=> (lecture2b/g 10000)
333383335000
user=> (lecture2b/g 100000)
333338333350000
user=> (lecture2b/g 1000000)
333333833333500000
user=>
```

[Q] Write a program that computes $1^2 + 2^2 + \ldots + \overset{n^2}{\cancel{5^2}}$.

Exercise 1: Fibonacci sequence

[Q] Write a Clojure function that takes $n \geq 2$ and computes the n-th Fibonacci number F_n:

$$F_1 = 1, \quad F_2 = 1, \quad F_{n+2} = F_n + F_{n+1}$$

Exercise 1: Fibonacci sequence

[Q] Write a Clojure function that takes n≥2 and computes the n-th Fibonacci number F_n:

$$F_1 = 1, \quad F_2 = 1, \quad F_{n+2} = F_n + F_{n+1}$$

[Hint]

```
(defn fib [n]
  (loop [i . . . . .   ]
    (if (= i n)
      . . .
      (recur (+ i 1)
             . . . .
             . . . .  ))))
```

Exercise 1: Fibonacci sequence

[Q] Write a Clojure function that takes $n \geq 2$ and computes the n-th Fibonacci number F_n:

$$F_1 = 1, \quad F_2 = 1, \quad F_{n+2} = F_n + F_{n+1}$$

[Hint]

```
(defn fib [n]
  (loop [i 2 r0 1 r1 1]
    (if (= i n)
      . . .
      (recur (+ i 1)
             . . . .
             . . . .  ))))
```

Exercise 1: Fibonacci sequence

[Q] Write a Clojure function that takes n≥2 and computes the n-th Fibonacci number F_n:

$$F_1 = 1, \quad F_2 = 1, \quad F_{n+2} = F_n + F_{n+1}$$

[Hint]

```
(defn fib [n]
  (loop [i 2 r0 1 r1 1]
    (if (= i n)
      r1
      (recur (+ i 1)
                 . . . .
                 . . . .  ))))
```

Exercise 1: Fibonacci sequence

[Q] Write a Clojure function that takes $n \geq 2$ and computes the n-th Fibonacci number F_n:

$$F_1 = 1, \quad F_2 = 1, \quad F_{n+2} = F_n + F_{n+1}$$

[Hint]

```
(defn fib [n]
  (loop [i 2 r0 1 r1 1]
    (if (= i n)
      r1
      (recur (+ i 1)
             r1
             (+ r0 r1)))))
```

Exercise 2: Random Fibonacci sequence R_n

$R_1 = 1, \quad R_2 = 1,$

$R_{n+2} = R_n + R_{n+1}$ or $R_{n+1} - R_n$, each with prob. 1/2

[Q] What does the distribution of R_n look like?

Anglican in a nutshell

1. Define an Anglican query using defquery.

```
(defquery baby-rfib [n]
  (let [b (sample (flip 0.5))
        new-n (if b n (+ n 1))]
    (loop [i 2 r0 1 r1 1]
      (if (= i new-n)
          r1
          (recur (+ i 1)
                 r1
                 (+ r0 r1)))))))
```

Anglican in a nutshell

1. Define an Anglican query using defquery.

```
(defquery baby-rfib [n]
  (let [b (sample (flip 0.5))
        new-n (if b n (+ n 1))]
    (loop [i 2 r0 1 r1 1]
      (if (= i new-n)
          r1
          (recur (+ i 1)
                 r1
                 (+ r0 r1))))))
```

Anglican in a nutshell

1. Define an Anglican query using defquery.

query name

```
(defquery baby-rfib [n]
  (let [b (sample (flip 0.5))
        new-n (if b n (+ n 1))]
    (loop [i 2 r0 1 r1 1]
      (if (= i new-n)
          r1
          (recur (+ i 1)
                 r1
                 (+ r0 r1))))))
```

Anglican in a nutshell

1. Define an Anglican query using defquery.

query name

arguments

```
(defquery baby-rfib [n]
  (let [b (sample (flip 0.5))
        new-n (if b n (+ n 1))]
    (loop [i 2 r0 1 r1 1]
      (if (= i new-n)
          r1
          (recur (+ i 1)
                 r1
                 (+ r0 r1))))))
```

Anglican in a nutshell

1. Define an Anglican query using defquery.

```
(defquery baby-rfib [n]
  (let [b (sample (flip 0.5))
        new-n (if b n (+ n 1))]
    (loop [i 2 r0 1 r1 1]
      (if (= i new-n)
          r1
          (recur (+ i 1)
                 r1
                 (+ r0 r1))))))
```

query name

arguments

query body

Anglican in a nutshell

Creating and sampling from distribution object

ery using defquery.

query name

arguments

query body

```
(defquery baby-rfib [n]
  (let [b (sample (flip 0.5))
        new-n (if b n (+ n 1))]
    (loop [i 2 r0 1 r1 1]
      (if (= i new-n)
        r1
        (recur (+ i 1)
               r1
               (+ r0 r1))))))
```

Anglican in a nutshell

1. Define a Anglican query using defquery.

query name

arguments

```
(defquery baby-rfib [n]
  (let [b (sample (flip 0.5))
        new-n (if b n (+ n 1))]
    (loop [i 2 r0 1 r1 1]
      (if (= i new-n)
          r1
          (recur (+ i 1)
                 r1
                 (+ r0 r1))))))
```

query body

Anglican in a nutshell

1. Define an Anglican query using defquery.

query name

arguments

```
(defquery baby-rfib [n]
  (let [b (sample (flip 0.5))
        new-n (if b n (+ n 1))]
    (loop [i 2 r0 1 r1 1]
      (if (= i new-n)
          r1
          (recur (+ i 1)
                 r1
                 (+ r0 r1))))))
```

query body

[Q] Write an Anglican query for generating R_n for given n.

Anglican in a nutshell

1. Define an Anglican query using defquery.

```
(defquery rfib [n]
  (loop [i 2 r0 1 r1 1]
    (if (= i n)
      r1

      ● ● ●

                 ))))
```

```
(def
  (l

                       r1
                       (+ r0 r1)))))))
```

[Q] Write an Anglican query for generating R_n for given n.

Anglican in a nutshell

1. Define an Anglican query using defquery.

```
(defquery rfib [n]
  (loop [i 2 r0 1 r1 1]
    (if (= i n)
      r1
      (let [b  (sample (flip 0.5))
                                         ]
            ...


          ...      ))))
```

```
(def
  (l
           r1
           (+ r0 r1))))))
```

[Q] Write an Anglican query for generating R_n for given n.

Anglican in a nutshell

1. Define an Anglican query using defquery.

```
(defquery rfib [n]
  (loop [i 2 r0 1 r1 1]
    (if (= i n)
      r1
      (let [b  (sample (flip 0.5))
            r2 (if b (+ r1 r0) (- r1 r0))]
        (recur (+ i 1)
               r1
               r2))))))
```

```
(def
  (l
           r1
           (+ r0 r1)))))))
```

[Q] Write an Anglican query for generating R_n for given n.

Anglican in a nutshell

2. Perform inference using doquery.

```
(doquery :importance rfib [20])
```

Anglican in a nutshell

2. Perform inference using doquery.

```
(doquery :importance rfib [20])
```

Anglican in a nutshell

2. Perform inference using doquery.

```
(doquery :importance rfib [20])
```

$\uparrow$

Clojure keyword.
Chooses an inference algorithm.

Anglican in a nutshell

query name,
arguments

2. Perform inference using doquery.

```
(doquery :importance rfib [20])
```

Clojure keyword.
Chooses an inference algorithm.

Anglican in a nutshell

query name,
arguments

2. Perform inference using doquery.

```
(doquery :importance rfib [20])
```

Clojure keyword.
Chooses an inference algorithm.

Returns a lazy infinite
sequence of samples.
Only a finite prefix used.

Anglican in a nutshell

2. Perform inference using doquery.

```
(let [s (doquery :importance rfib [20])]
  (take 2 s))
```

Anglican in a nutshell

2. Perform inference using doquery.

```
(let [s (doquery :importance rfib [20])]
  (take 2 s))
```

```
( {:log-weight 0.0,
   :result 1,
   :predict []}

  {:log-weight 0.0,
   :result -17,
   :predict []} )
```

Anglican in a nutshell

2. Perform inference using doquery.

```
(let [s (doquery :importance rfib [20])]
  (take 2 s))
```

```
( {:log-weight 0.0,
   :result 1,
   :predict []}

  {:log-weight 0.0,
   :result -17,
   :predict []} )
```

List

Anglican in a nutshell

2. Perform inference using doquery.

```
(let [s (doquery :importance rfib [20])]
  (take 2 s))
```

```
( {:log-weight 0.0,
   :result 1,
   :predict []}

  {:log-weight 0.0,
   :result -17,
   :predict []} )
```

List
of maps

Anglican in a nutshell

2. Perform inference using doquery.

```
(let [s (doquery :importance rfib [20])]
  (take 2 s))
```

```
( {:log-weight 0.0,
   :result 1,
   :predict []}

  {:log-weight 0.0,
   :result -17,
   :predict []} )
```

List
of maps
with three keys

Anglican in a nutshell

2. Perform inference using doquery.

```
(let [s (doquery :importance rfib [20])]
  (take 2 s))
```

```
( {:log-weight 0.0,
   :result 1,
   :predict []}

  {:log-weight 0.0,
   :result -17,
   :predict []} )
```

List
of maps
with three keys

- To move on, we need to understand map and list datatypes of Clojure.

- To move on, we need to understand map and list datatypes of Clojure.
- Two key questions:

1. How to construct a datatype?
2. How to destruct (or decompose) it?

Map in Clojure

1. Constructed using {..} or assoc typically.

```
{:a 0, :b 1, 3 10},
(assoc {:a 0, :b 1} 3 10)
```

2. Accessed (or destructed) by get & keyword.

```
(get {:a 0, :b 1, 3 10} 3)
(get {:a 0, :b 1, 3 10} :a)
(:a {:a 0, :b 1, 3 10})
```

List in Clojure

1. Created using list and conj typically.

```
(list 1 2 3), (conj (list 2 3) 1)
```

2. Destructed by first, rest, and take.

```
(first (list 1 2 3)),
(rest (list 1 2 3)),
(take 2 (list 1 2 3))
```

List in Clojure

3. Changed using map and filter.

```
(map inc (list 1 2 3)),
(map + (list 1 2 3) (list 10 11 12))
(filter (fn [x] (>= x 2)) (list 1 2 3))
```

4. reduce.

```
(reduce + 0.0 (list 1 2 3 4))
```

5. drop, empty?, many other functions. Google.

Summary

- Map: {..}, assoc, get, and access by keyword.
- List: list, conj, first, rest, take, map, filter, reduce, drop, empty?, etc.

Summary

- Map: {..}, assoc, get, and access by keyword.
- List: list, conj, first, rest, take, map, filter, reduce, drop, empty?, etc.

Core functions

Summary

- Map: {..}, assoc, get, and access by keyword.
- List: list, conj, first, rest, take, map, filter, reduce, drop, empty?, etc.

Masters' tools.

Core functions

Summary

- Map: {..}, assoc, get, and access by keyword.
- List: list, conj, first, rest, take, map, filter, reduce, drop, empty?, etc.

[Q1] Write a fun. `rev` that reverses a list.

[Q2] Write a fun. `conc` that concatenates two lists.

```
(defn rev [l]
  (reduce conj (list) l))
```

y

- Map: {..}, assoc, get, and access by keyword.
- List: list, conj, first, rest, take, map, filter, reduce, drop, empty?, etc.

[Q1] Write a fun. `rev` that reverses a list.

[Q2] Write a fun. `conc` that concatenates two lists.

```
(defn rev [l]
  (reduce conj (list) l))

(defn conc [l1 l2]
  (reduce conj l2 (rev l1))
```

y

- Map: {..}, assoc, get, and access by keyword.
- List: list, conj, first, rest, take, map, filter, reduce, drop, empty?, etc.

[Q1] Write a fun. `rev` that reverses a list.

[Q2] Write a fun. `conc` that concatenates two lists.

Anglican in a nutshell

1. Define a Anglican query using defquery.
2. Perform inference using doquery.

```
(let [s (doquery :importance rfib [20])]
  (take 2 s))
```

```
( {:log-weight 0.0,
    :result 1,
    :predict []}
 {:log-weight 0.0,
    :result -17,
    :predict []} )
```

List
of maps
with three keys

Anglican in a nutshell

3. Pick :result entries and analyse them.

```
(doquery :importance rfib [20])
```

Anglican in a nutshell

3. Pick :result entries and analyse them.

```
(let [s (doquery :importance rfib [20])
      r (map :result (take 1000 s))]




                                                     )
```

Anglican in a nutshell

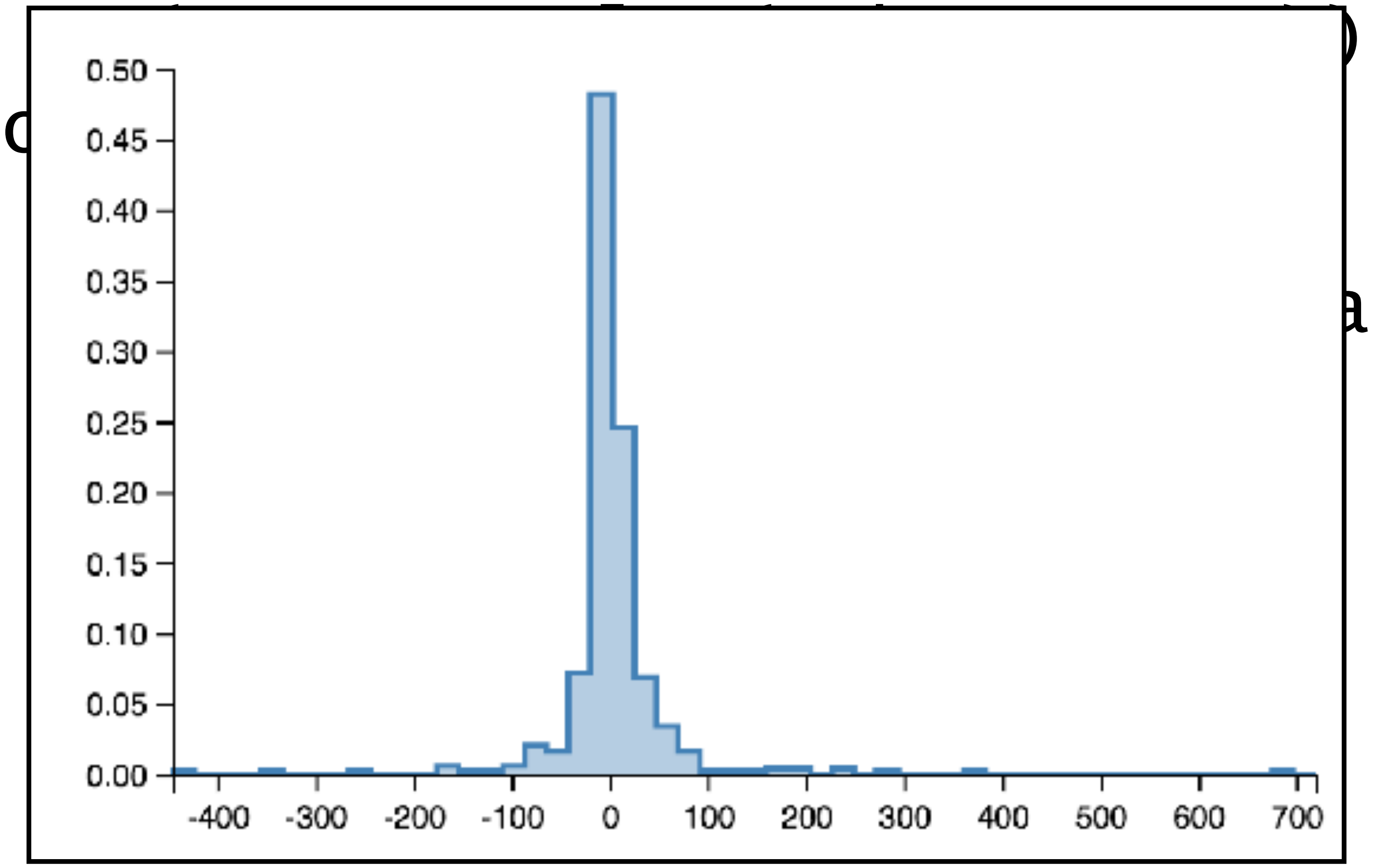
3. Pick :result entries and analyse them.

```
(let [s (doquery :importance rfib [20])
      r (map :result (take 1000 s))]
  (plot/histogram r
                  :bins 50
                  :normalize :probability))
```

Anglican in a nutshell

3. Pick :result entries and analyse them.

```
(let [s (doquery :importance rfib [20])
                                              )]
  (pl

                                              ability))
```



Anglican in a nutshell

3. Pick :result entries and analyse them.

```
(let [s (doquery :importance rfib [20])
      r (map :result (take 1000 s))]
  (plot/histogram r
                  :bins 50
                  :normalize :probability))
```

[Q1] Compute the average of generated R_{20} using 1000 samples. This is called empirical mean.

Anglican in a nutshell

3. Pick :result entries and analyse them.

```
(let [s (doquery :importance rfib [20])
      r (map :result (take 1000 s))]
  (/ (reduce + 0.0 r) 1000.0))
```

[Q1] Compute the average of generated R_{20} using 1000 samples. This is called empirical mean.

Anglican in a nutshell

3. Pick :result entries and analyse them.

```
(let [s (doquery :importance rfib [20])
      r (map :result (take 1000 s))]
  (/ (reduce + 0.0 r) 1000.0))
```

[Q1] Compute the average of generated R_{20} using 1000 samples. This is called empirical mean.
[Q2] Compute the variance of generated R_{20}.

Anglican in a nutshell

3. Pick :result entries and analyse them.

```
(let [s (doquery :importance rfib [20])
      r (map :result (take 1000 s))
      m (/ (reduce + 0.0 r) 1000.0)
      f (fn [x] (Math/pow (- x m) 2))]
  (/ (reduce + 0.0 (map f r)) 1000.0))
```

[Q1] Compute the average of generated R_{20} using 1000 samples. This is called empirical mean.
[Q2] Compute the variance of generated R_{20}.

Exercise 3:

[Part 1] Generate a list of indexed Fibonacci numbers (list F_1 F_2 … F_n) for a given n.

[Part 2] Generate a list of indices $0<i\leq10000$ such that i is the sum of digits of F_i.

Assume get-digits for converting a number to a list of its digits is given. (get-digits 23)=(list 2 3).

Assume range. (range 1 6) = (list 1 2 3 4 5).

Exercise 3:

[Part 1] Generate a list of indexed Fibonacci numbers (list F_1 F_2 … F_n) for a given n.

[Part 2] Generate a list of indices $0<i\leq 10000$ such that i is the sum of digits of F_i.

Assume get-digits for converting a number to a list of its digits is given. (get-digits 23)=(list 2 3).

Assume range. (range 1 6) = (list 1 2 3 4 5).

Part 1

```
(defn fib [n]
  (loop [i 2 s (list 1 1)]
    (if (= i n)
      (rev s)
      (recur (+ i 1)
             (conj s (+ (first s)
                        (second s)))))))
```

Part 2

```
(defn ck [fibn-and-i]
  (let [fibn (first fibn-and-i)
        i    (second fibn-and-i)]
    (= (reduce + 0 (get-digits fibn))
       i)))

(def idx-fib-seq
  (map (fn [s i] (list s i))
       (fib 10000)
       (range 1 10001)))

(def indices
  (map second (filter ck idx-fib-seq)))
```

Topics covered

- Functions, recursion, loop, list, and map in Clojure.
- defquery, and doquery in Anglican.

Announcement

1. Homework 0 in the course webpage.
    - It will teach you how to use Gorilla and to try examples in the web browser.
2. Form a group and tell us by the midnight of 13 March 2018 (Tuesday).