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:

   `(+ 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 $1^2 + 2^2 + \ldots + \overset{n^2}{\cancel{5^2}}$.

# Clojure in a nutshell

4. Separate function definition:

```
(defn g [n] …)
```

E.g.

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

(println (f 10))
```

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

# Clojure in a nutshell

4. Separate function definition:

```
(defn g [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 $1^2 + 2^2 + \ldots + \underbrace{5^2}_{}$. $n^2$

# Clojure in a nutshell

4. Separate function definition:

```
(defn g [n] …)
```

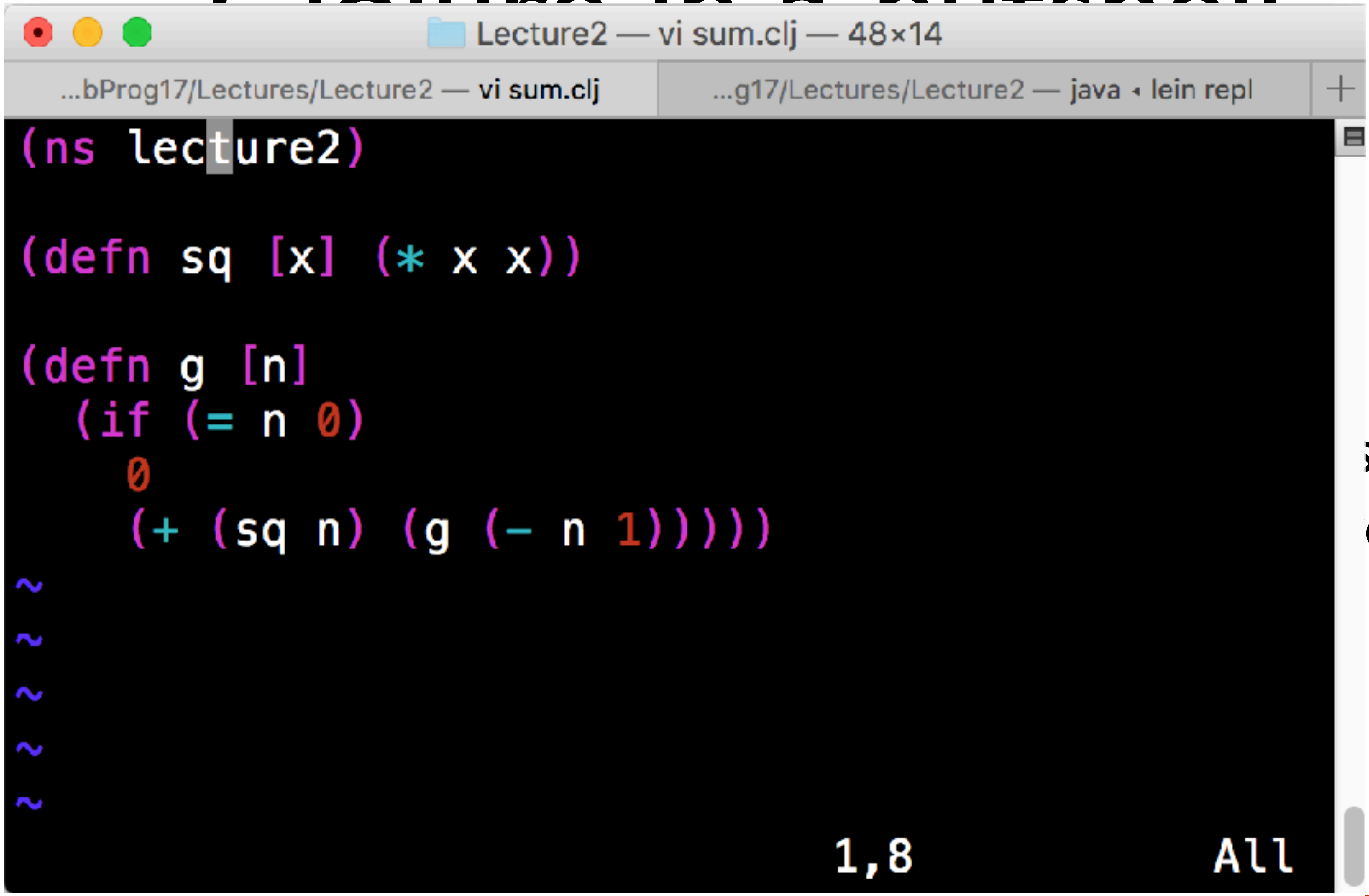
E.g.

```
(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 — 48×14

...bProg17/Lectures/Lecture2 — vi sum.clj | ...g17/Lectures/Lecture2 — java ◂ lein repl

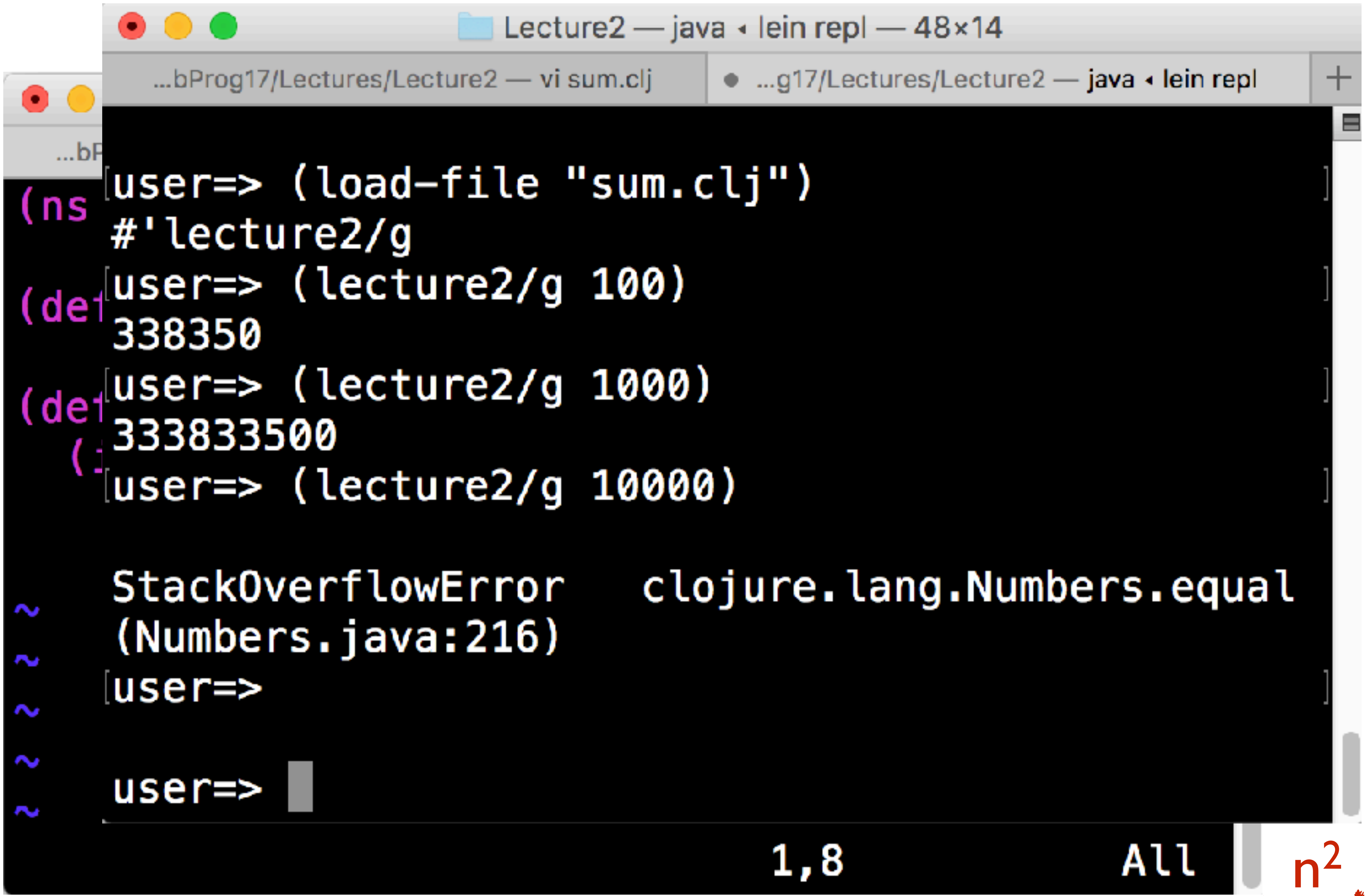
```
(ns lecture2)

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

(defn g [n]
  (if (= n 0)
    0
    (+ (sq n) (g (- n 1)))))
~
~
~
~
~
                                1,8           All
```

sion

ed

[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 — 48×14
...bProg17/Lectures/Lecture2 — vi sum.clj
...g17/Lectures/Lecture2 — java ◂ lein repl
1,8
All

```
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.equal
(Numbers.java:216)
user=>

user=> 
```

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

# Clojure in a nutshell

5. Tail recursion using loop and recur.

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

```
(defn f [n]
  (loop [n0 n r0 0]
    (if (= n0 0)
      r0
      (recur (- n0 1)
             (+ n0 r0)))))
```

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

# Clojure in a nutshell

5. Tail recursion using loop and recur.

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

```
(defn f [n]
  (loop [n0 n r0 0]
    (if (= n0 0)
      r0
      (recur (- n0 1)
             (+ n0 r0)))))
```

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

# Clojure in a nutshell

5. Tail recursion using loop and recur.

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

```
(defn f [n]
  (loop [n0 n r0 0]
    (if (= n0 0)
      r0
      (recur (- n0 1)
             (+ n0 r0)))))
```

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

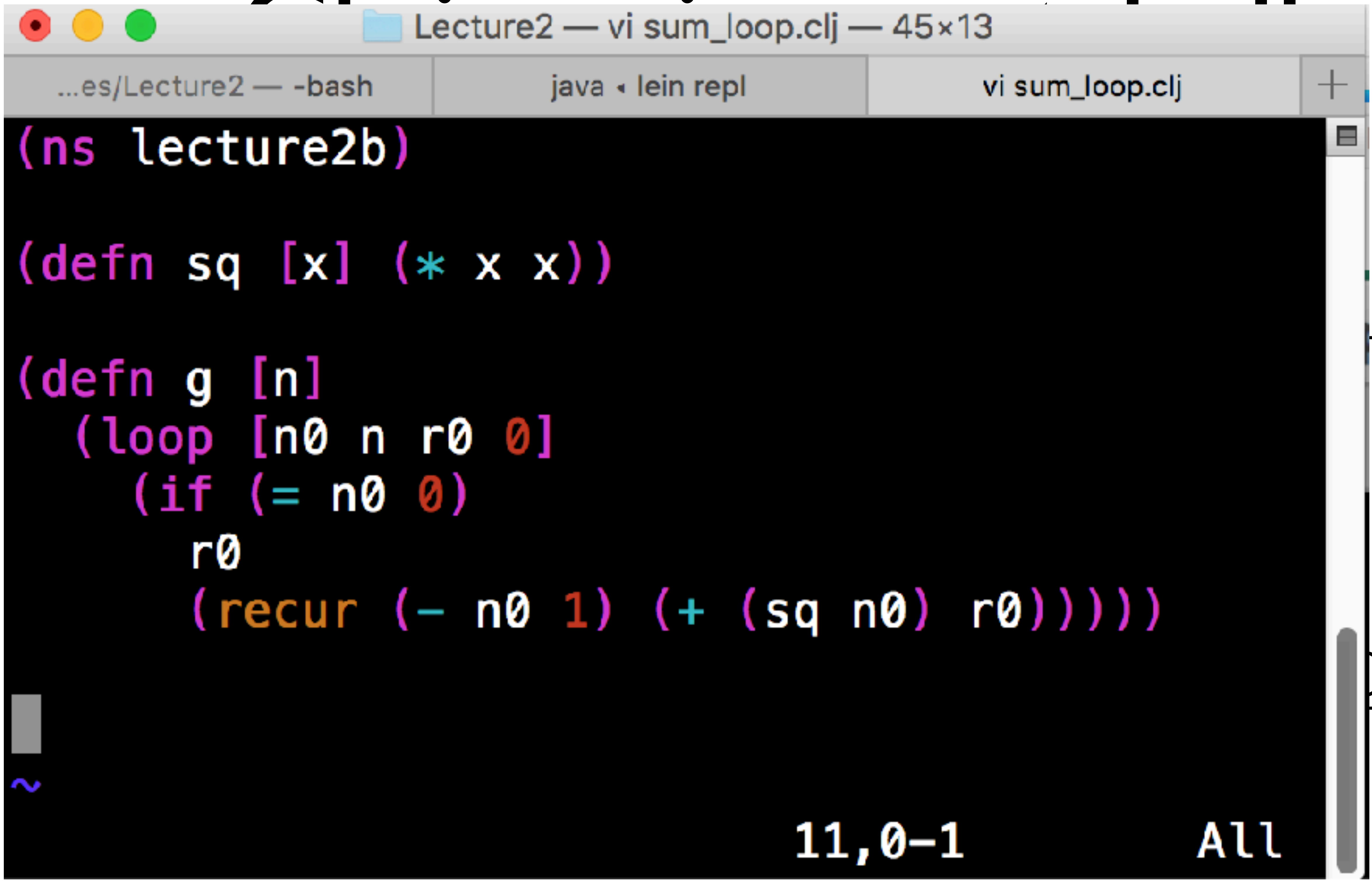
# Clojure in a nutshell

5. Tail recursion using loop and recur.

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

```
(defn f [n]
  (loop [n0 n r0 0]
    (if (= n0 0)
      r0
      (recur (- n0 1)
             (+ n0 r0)))))
```

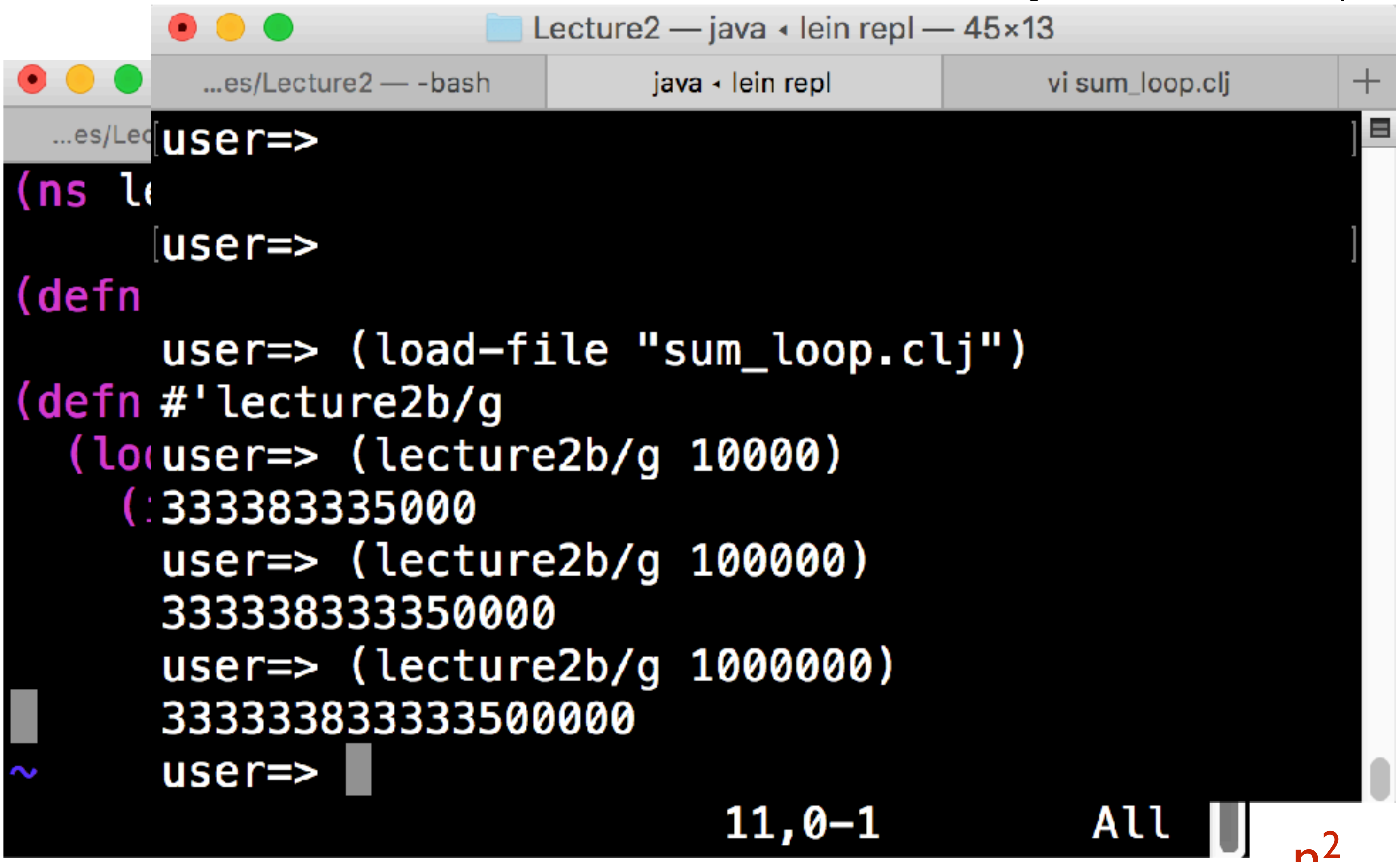
[Q] Write a program that computes $1^2 + 2^2 + \dots + \overset{n^2}{\cancel{5^2}}$.



```
(ns lecture2b)

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

(defn g [n]
  (loop [n0 n r0 0]
    (if (= n0 0)
      r0
      (recur (- n0 1) (+ (sq n0) r0)))))


~
                             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".



```
(ns le
(defn
(defn
  (lo
    (:
~
```

```
user=>

user=>

user=> (load-file "sum_loop.clj")
#'lecture2b/g
user=> (lecture2b/g 10000)
333383335000
user=> (lecture2b/g 100000)
333338333350000
user=> (lecture2b/g 1000000)
333333833333500000
user=> 
                    11,0-1          All
```

[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 $n \geq 2$ 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 [n0  . . . . .  ]
    (if (= n0 n)

      . . .
        (recur (+ n0 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 [n0 2 r0 1 r1 1]
    (if (= n0 n)
      . . .
      (recur (+ n0 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 [n0 2 r0 1 r1 1]
    (if (= n0 n)
      r1
      (recur (+ n0 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 [n0 2 r0 1 r1 1]
    (if (= n0 n)
      r1
      (recur (+ n0 1)
             r1
             (+ r0 r1)))))
```

# Exercise 2: Random Fibonacci sequence $R_n$

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

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

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

## Anglican in a nutshell

1. Define a Anglican query using defquery.

```
(defquery baby-rfib [n]
  (let [b (sample (flip 0.5))
        new-n (if b n (+ n 1))]
    (loop [n0 2 r0 1 r1 1]
      (if (= n0 new-n)
          r1
          (recur (+ n0 1)
                 r1
                 (+ r0 r1))))))
```

# Anglican in a nutshell

1. Define a Anglican query using defquery.

```
(defquery baby-rfib [n]
  (let [b (sample (flip 0.5))
        new-n (if b n (+ n 1))]
    (loop [n0 2 r0 1 r1 1]
      (if (= n0 new-n)
          r1
          (recur (+ n0 1)
                 r1
                 (+ r0 r1))))))
```

# Anglican in a nutshell

1. Define a Anglican query using defquery.

query name

```
(defquery baby-rfib [n]
  (let [b (sample (flip 0.5))
        new-n (if b n (+ n 1))]
    (loop [n0 2 r0 1 r1 1]
      (if (= n0 new-n)
          r1
          (recur (+ n0 1)
                 r1
                 (+ r0 r1)))))))
```

# Anglican in a nutshell

1. Define a Anglican query using defquery.

```
(defquery baby-rfib [n]
  (let [b (sample (flip 0.5))
        new-n (if b n (+ n 1))]
    (loop [n0 2 r0 1 r1 1]
      (if (= n0 new-n)
          r1
          (recur (+ n0 1)
                 r1
                 (+ r0 r1))))))
```

query name

arguments

# Anglican in a nutshell

1. Define a Anglican query using defquery.

```
(defquery baby-rfib [n]
  (let [b (sample (flip 0.5))
        new-n (if b n (+ n 1))]
    (loop [n0 2 r0 1 r1 1]
      (if (= n0 new-n)
          r1
          (recur (+ n0 1)
                 r1
                 (+ r0 r1))))))
```

query name

arguments

query body

# Anglican in a nutshell

Creating and sampling from distribution object

ry 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 [n0 2 r0 1 r1 1]
      (if (= n0 new-n)
          r1
          (recur (+ n0 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 [n0 2 r0 1 r1 1]
      (if (= n0 new-n)
          r1
          (recur (+ n0 1)
                 r1
                 (+ r0 r1))))))
```

query body

# 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 [n0 2 r0 1 r1 1]
      (if (= n0 new-n)
          r1
          (recur (+ n0 1)
                 r1
                 (+ r0 r1))))))
```

query body

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

# Anglican in a nutshell

1. Define a Anglican query using defquery.

```
(defquery rfib [n]
  (loop [n0 2 r0 1 r1 1]
    (if (= n0 n)
      r1
      (let [b (sample (flip 0.5))
            r2 (if b (+ r1 r0) (- r1 r0))]
        (recur (+ n0 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 []} )
```

Sequence

# 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 []} )
```

Sequence
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 []} )
```

Sequence
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 []} )
```

Sequence
of maps
with three keys

- To move on, we need to understand map and sequence 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})
```

# Sequence in Clojure

1. Created using list and conj typically.

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

2. Destructed by first, rest and reduce.

```
(first (list 1 2 3)),
(rest (list 1 2 3)),
(reduce + (list 1 2 3))
```

## Sequence in Clojure

3. Changed using map and filter.

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

## 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 []} )
```

Sequence
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

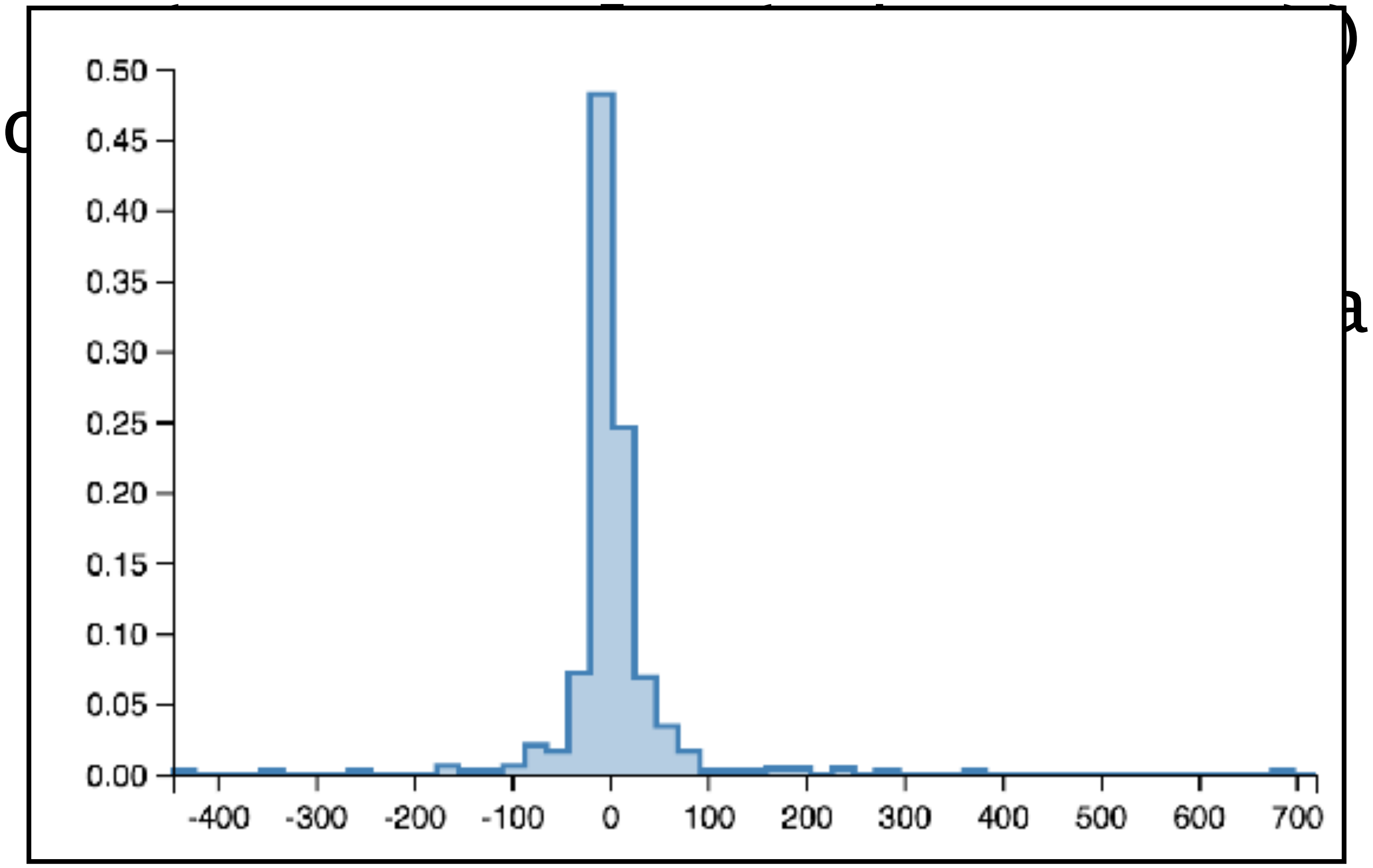
3. Pick :result entries and analyse them.

```
(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 + r) 1000))
```

[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 + r) 1000))
```

[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))]
  (/ (reduce + r) 1000))
```

[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 + r) 1000)
      f (fn [x] (Math/pow (- x m) 2))]
  (/ (reduce + (map f r)) 1000))
```

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

[Q2] Compute the variance of generated $R_{20}$.

## Topics covered

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

# Announcement

1. Homework 1 in the course web page.
    - It will teach you how to use Gorilla and to try examples in the web browser.
2. No lectures on 4 & 6 September 2017.
3. Form a group and tell us by the midnight of 11 September 2017.