Thinking the Clojure Way

Christophe Grand

GOTO Cph, May 13th 2011

Rejoice, Clojure is simple

Coding along/against
Singing with

à la carte

Rejoice, Clojure is simple

Small regular syntax

Coding along/against
Singing with

à la carte

Rejoice, Clojure is simple

Small regular syntax

Simple does ***not*** mean familiar

Coding along/against
Singing with

à la carte

Rejoice, Clojure is simple

Small regular syntax

Simple does *not* mean familiar

Simple means *not compound*

Coding along/against
Singing with

à la carte

Rejoice, Clojure is simple

Small regular syntax

Simple does *not* mean familiar

Simple means *not compound*

Clojure is made of simple things

Coding along/against
Singing with

à la carte

Rejoice, Clojure is simple

Small regular syntax

Simple does <u>*not*</u> mean familiar

Simple means *not compound*

Clojure is made of simple things

Small set of independent concepts

Coding along/against
Singing with

à la carte

Rejoice, Clojure is simple

Coding along/against
Singing with

à la carte

Small regular syntax

Simple does *not* mean familiar

Simple means *not compound*

Clojure is made of simple things

Small set of independent concepts

One concept at a time

Rejoice, Clojure is simple

Coding along/against
Singing with

à la carte

- Small regular syntax
- Simple does *not* mean familiar
- Simple means *not compound*
- Clojure is made of simple things
- Small set of independent concepts
- One concept at a time

One bite at a time

faire 4 groupes :
Syntax+FP
Rec+lazy seqs
polymorphism+types
interop+mutation

les deux derniers blocs devraient être animés pour être mis au même niveau

- Syntax
- Core Functional Programming
- Recursion and loops
- Lazy seqs (creating your owns)
- Polymorphism
- Types
- Macros
- Interop
- State management

Syntax

Literals

Numbers	`42 3.14 3/4 5.01M 43N`
Strings	`"Hello GOTO Cph"`
Characters	`\c \newline`
Keywords	`:a-key`
Symbols	`foo clojure.core/map`
Vectors	`[1 "two" :three]`
Maps	`{:key "val", :key2 42}`
Sets	`#{1 "two" :three}`
Regex	`#"a.*b"`
null	`nil`
booleans*	`true false`

*anything but *false* and *nil* is true

What about lists?

What about lists?

Quoted lists are too literal: `'(1 (+ 1 1))`

What about lists?

Quoted lists are too literal: `'(1 (+ 1 1))`

Displaced as literals by vectors

What about lists?

Quoted lists are too literal: `'(1 (+ 1 1))`

Displaced as literals by vectors

Abstracted away by sequences

What about lists?

Quoted lists are too literal: `'(1 (+ 1 1))`

Displaced as literals by vectors

Abstracted away by sequences

Survive mostly for code representation

Lists are for code

```
(defn abs [n]
  (if (neg? n)
    (- n)
    n))
```

Lists are for code

```
(defn abs [n]
  (if (neg? n)
    (- n)
    n))
```

Pro tip: Lisp code is a stereogram

Lists are for code

```
(defn abs [n]
  (if (neg? n)
    (- n)
   n))
```

Pro tip: Lisp code is a stereogram
Cross your eyes to see parens in the right place

Lists are for code

```
defn abs [n](
  if  neg?(n)(
    -(n)
   n))
```

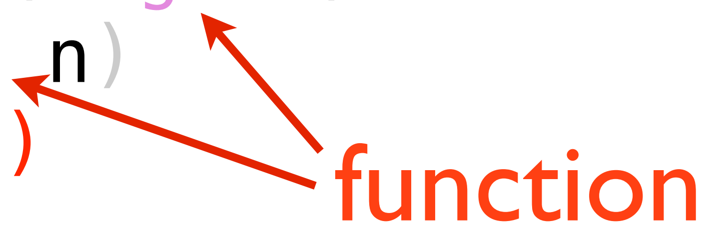
Pro tip: Lisp code is a stereogram
Cross your eyes to see parens in the right place

Lists are for code

```
(defn abs [n]
  (if (neg? n)
    (- n)
    n))
```

Lists are for code

```
(defn abs [n]
  (if (neg? n)
    (- n)
    n))
```



Lists are for code

```
(defn abs [n]
  (if (neg? n)
    (- n)
   n))
```

macro

function

Lists are for code

```
(defn abs [n]
  (if (neg? n)
    (- n)
   n))
```

macro

special form

function

That's all about syntax!

Functional Programming

Clojure’s FP

Clojure’s FP

Impure

Clojure’s FP

Impure

Persistent collections

Clojure's FP

Impure

Persistent collections

Strictly evaluated

Clojure’s FP

Impure

Persistent collections

Strictly evaluated

But lazy sequences

Clojure's FP

Impure

Persistent collections

Strictly evaluated

But lazy sequences

- Not strictly lazy though!

Clojure's FP

Impure

Persistent collections

Strictly evaluated

But lazy sequences

- Not strictly lazy though!

Persistent collections

Persistent collections

Vectors, maps and sets

Persistent collections

Vectors, maps and sets

Common usecases:

Persistent collections

Vectors, maps and sets

Common usecases:

- Vectors as tuples

Persistent collections

- Vectors, maps and sets
- Common usecases:
  - Vectors as tuples
  - Vectors as stacks

Persistent collections

Vectors, maps and sets

Common usecases:

- Vectors as tuples
- Vectors as stacks
- Maps as data

Persistent collections

Vectors, maps and sets

Common usecases:

- Vectors as tuples
- Vectors as stacks
- Maps as data
- Map as index, summary

Persistent collections

Vectors, maps and sets

Common usecases:

- Vectors as tuples
- Vectors as stacks
- Maps as data
- Map as index, summary
- Sets as containers, relations

Sequences

Sequences

First, what are sequences?

Sequences

First, what are sequences?

Abstraction over linked lists

Sequences

First, what are sequences?

Abstraction over linked lists

List-like views over data

Sequences

First, what are sequences?

Abstraction over linked lists

List-like views over data

- Support `first` and `rest`

Sequences

First, what are sequences?

Abstraction over linked lists

List-like views over data

- Support `first` and `rest`
- Replace iterators

Sequences

First, what are sequences?

Abstraction over linked lists

List-like views over data

- Support `first` and `rest`
- Replace iterators
- Replace indices

Lazy sequences

Lazy sequences

Sequences evaluated (realized) on demand

Lazy sequences

Sequences evaluated (realized) on demand

Allow to process big data

Lazy sequences

Sequences evaluated (realized) on demand

Allow to process big data

  or big intermediate values

Lazy sequences

Sequences evaluated (realized) on demand

Allow to process big data

or big intermediate values

```
(->> (slurp "access.log") split-lines (map count)
  (filter odd?))
```

Lazy sequences

Doesn't matter w/ strict evaluation

Lazy sequences

Realization can go ahead of consumption

Doesn't matter w/ strict evaluation

Lazy sequences

Realization can go ahead of consumption

Not suitable for control flow

Doesn't matter w/ strict evaluation

Lazy sequences

Realization can go ahead of consumption

- Not suitable for control flow
- Better locality, less churn

Doesn't matter w/ strict evaluation

That’s all about FP!

Clojure spirit

Clojure spirit

Clojure spirit

Pragmatism

Clojure spirit

Pragmatism

Correctness

Clojure spirit

Pragmatism

Correctness

Uniform interfaces

Clojure spirit

Pragmatism

Correctness

Uniform interfaces

Data over functions

Clojure spirit

Pragmatism

Correctness

Uniform interfaces

Data over functions

Sequences as computation media

Clojure spirit

Pragmatism

Correctness

Uniform interfaces

Data over functions

Sequences as computation media

Reftypes as mutation patterns

Pragmatism

Hosted on the JVM

Embrace the host limitations

 to be a better guest

Excellent Java interop

Performance over purity

LISP

Correctness

- No silent error
- Correct result or failure
- Non-negotiable
  - See 1.2 -> 1.3 numerics changes
  - Unless the user opts in

Uniform interfaces

Uniform interfaces

Widespread small interfaces

Uniform interfaces

Widespread small interfaces

Tons of helpers fns built upon

Uniform interfaces

Widespread small interfaces

Tons of helpers fns built upon

It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures.

– Alan Perlis

Uniform interfaces

Widespread small interfaces

Tons of helpers fns built upon

It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures.

– Alan Perlis

*It is better to have 100 functions operate on one **abstraction** than 10 functions on 10 data structures.*

– Rich Hickey

Uniform interfaces

```
=> (-> {:product-id "ACME123", :description "Powder
water"} keys sort)
(:description :product-id)
```

```
=> (-> (javax.swing.JFrame.) bean keys sort)
(:JMenuBar :accessibleContext :active :alignmentX
:alignmentY :alwaysOnTop :alwaysOnTopSupported
:background :bufferStrategy :componentCount :components
:containerListeners :contentPane :cursorType
:defaultCloseOperation ...)
```

Data over functions

- Large reuse of core collection fns
  - Less specific code
- Data go out of the process
  - Don’t be too clever
- A schema is a good API

Data over functions

Data over functions

How to enforce invariants

Data over functions

- How to enforce invariants
  - Write a validator function

Data over functions

- How to enforce invariants
  - Write a validator function
  - Use it in fns pre- and post-conditions

Data over functions

- How to enforce invariants
  - Write a validator function
  - Use it in fns pre- and post-conditions
  - Use it in reftypes validators

Sequences as pipes

Sequences as pipes

Sequences as ephemeral media of computation

Sequences as pipes

Sequences as ephemeral media of computation

Each stage of a pipeline yields its own seq

Sequences as pipes

Sequences as ephemeral media of computation

Each stage of a pipeline yields its own seq

The ends of the pipeline are not seqs:

Sequences as pipes

Sequences as ephemeral media of computation

Each stage of a pipeline yields its own seq

The ends of the pipeline are not seqs:

db, network, persistent collection etc.

Sequences as pipes

Sequences as ephemeral media of computation

Each stage of a pipeline yields its own seq

The ends of the pipeline are not seqs:

- db, network, persistent collection etc.

Mutation patterns

Mutation patterns

Reftypes embody mutation patterns

Mutation patterns

Reftypes embody mutation patterns

Application state management:

Mutation patterns

Reftypes embody mutation patterns

Application state management:

- Refs, atoms and agents

Mutation patterns

Reftypes embody mutation patterns

Application state management:

- Refs, atoms and agents

Program state management:

Mutation patterns

Reftypes embody mutation patterns

Application state management:

- Refs, atoms and agents

Program state management:

- Vars

Mutation patterns

Reftypes embody mutation patterns

Application state management:

- Refs, atoms and agents

Program state management:

- Vars

Execution or dataflow management:

Mutation patterns

Reftypes embody mutation patterns

Application state management:

- Refs, atoms and agents

Program state management:

- Vars

Execution or dataflow management:

- Promises, delays and futures

How to think functionally?

Break your habits

Tie your imperative hand behind your back!

Tie your OO hand too!

Features

Syntax

Core Functional Programming

Recursion and loops

Lazy seqs (creating your owns)

Polymorphism

Types

Macros

Interop

Mutation

Features

Syntax

Core Functional Programming

Recursion and loops

Lazy seqs (creating your owns)

Polymorphism

Types

Macros

Interop

~~Mutation~~

Features

Syntax

Core Functional Programming

Recursion and loops

Lazy seqs (creating your owns)

Polymorphism

Types

Macros

~~Interop~~

~~Mutation~~

Features

Syntax

Core Functional Programming

Recursion and loops

Lazy seqs (creating your owns)

Polymorphism

Types

~~Macros~~

~~Interop~~

~~Mutation~~

Features

Syntax

Core Functional Programming

Recursion and loops

Lazy seqs (creating your owns)

Polymorphism

~~Types~~

~~Macros~~

~~Interop~~

~~Mutation~~

Features

Syntax

Core Functional Programming

Recursion and loops

Lazy seqs (creating your owns)

~~Polymorphism~~

~~Types~~

~~Macros~~

~~Interop~~

~~Mutation~~

Features

Syntax

Core Functional Programming

Recursion and loops

~~Lazy seqs (creating your owns)~~

~~Polymorphism~~

~~Types~~

~~Macros~~

~~Interop~~

~~Mutation~~

Features

Syntax

Core Functional Programming

~~Recursion and loops~~

~~Lazy seqs (creating your owns)~~

~~Polymorphism~~

~~Types~~

~~Macros~~

~~Interop~~

~~Mutation~~

Allowed subset

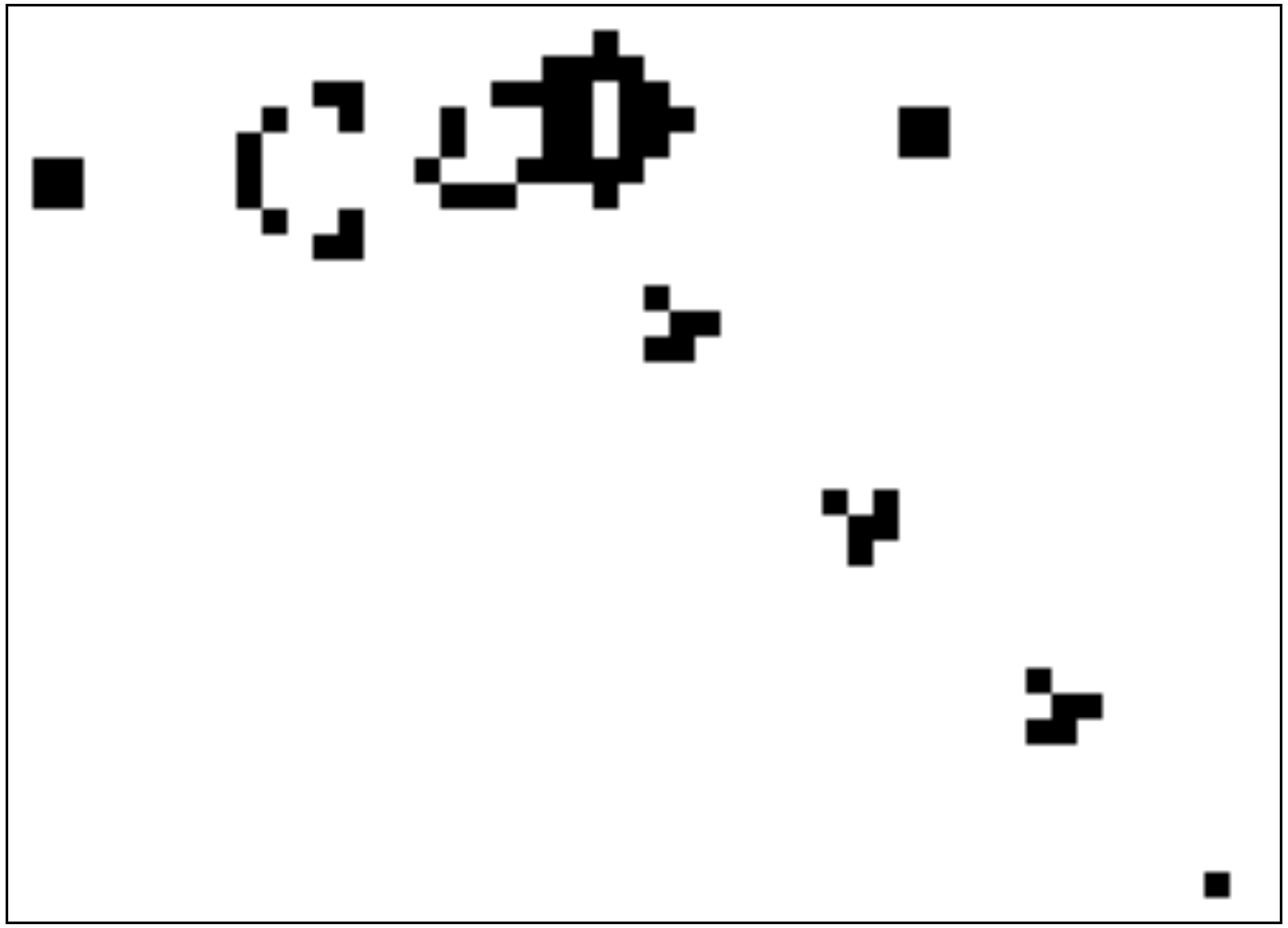
Pure Functional Programming

- without recursion nor loops
- without `lazy-seq`
- without indices

Do it until it hurts!
(and works)

Do it especially for ill-suited problems!

Example:
Conway's game of life



Rules

At each step in time, the following transitions occur:

- Any live cell with fewer than two live neighbours dies, as if caused by under-population.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overcrowding.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

(wikipedia)

Conway's game of life

Typical implementation is full of indices and loops

```
(defn step
 "Takes a vector of vectors of 0 and 1, and
  returns the next iteration of the automaton."
 [board]
  (let [w (count board)
        h (count (first board))]
    (loop [i 0 j 0 new-board board]
      (cond
        (>= i w) new-board
        (>= j h) (recur (inc i) 0 new-board)
        :else
          (let [n (neighbours-count board i j)
                nb (cond
                     (= 3 n) (assoc-in new-board [i j] 1)
                     (not= 2 n) (assoc-in new-board [i j] 0)
                     :else new-board)]
            (recur i (inc j) new-board))))))
```

Conway's game of life

Typical implementation is full of indices and loops

```
(defn step
 "Takes a vector of vectors of 0 and 1, and
  returns the next iteration of the automaton."
 [board]
  (let [w (count board)
        h (count (first board))]
    (loop [i 0 j 0 new-board board]
      (cond
        (>= i w) new-board
        (>= j h) (recur (inc i) 0 new-board)
        :else
          (let [n (neighbours-count board i j)
                nb (cond
                     (= 3 n) (assoc-in new-board [i j] 1)
                     (not= 2 n) (assoc-in new-board [i j] 0)
                     :else new-board)]
            (recur i (inc j) new-board))))))
```

Conway’s game of life

And there’s more!

```
(defn neighbours-count [board i j]
  (let [i+1 (inc i) j+1 (inc j)]
    (loop [cnt 0 x (dec i) y (dec j)]
      (cond
        (> x i+1) cnt
        (> y j+1) (recur cnt (inc x) (dec j))
        (= [x y] [i j]) (recur cnt x (inc y))
        :else (recur (+ cnt (get-in board [x y] 0))
                     x (inc y))))))
```

Conway's game of life

And there's more!

```clojure
(defn neighbours-count [board i j]
  (let [i+1 (inc i) j+1 (inc j)]
    (loop [cnt 0 x (dec i) y (dec j)]
      (cond
        (> x i+1) cnt
        (> y j+1) (recur cnt (inc x) (dec j))
        (= [x y] [i j]) (recur cnt x (inc y))
        :else (recur (+ cnt (get-in board [x y] 0))
                     x (inc y))))))
```

Look, no indices

At each step in time, the following transitions occur:

- Any live cell with fewer than two live neighbours dies, as if caused by under-population.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overcrowding.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

(wikipedia)

Take a step back!

Look, no indices

At each step in time, the following transitions occur:

- Any live cell with fewer than two live **neighbours** dies, as if caused by under-population.
- Any live cell with two or three live **neighbours** lives on to the next generation.
- Any live cell with more than three live **neighbours** dies, as if by overcrowding.
- Any dead cell with exactly three live **neighbours** becomes a live cell, as if by reproduction.

(wikipedia)

Neighbours!

Neighbours!

Try to express the rules in code using the **neighbours** concept

Neighbours!

Try to express the rules in code using the **neighbours** concept

Don't worry about the implementation of **neighbours**

Neighbours!

Neighbours!

For each living cell or neighbour of a living cell, compute the number of neighbours.

Neighbours!

For each living cell or neighbour of a living cell, compute the number of neighbours.

Then apply generation rules.

Neighbours!

Neighbours!

Compute all neighbours of the living cells.

Neighbours!

Compute all neighbours of the living cells.

Occurences count is neighbour count!

Neighbours!

Compute all neighbours of the living cells.

Occurences count is neighbour count!

Then apply generation rules.

Neighbours!

```
(defn step
  "Takes a set of living cells and returns the next
generation (as a set too)."
  [living-cells]
  (letfn [(alive [[cell cnt]]
            (when (or (= cnt 3)
                      (and (= cnt 2) (living-cells cell)))
              cell))]
    (->> living-cells (mapcat neighbours) frequencies
      (keep alive) set)))
```

Neighbours!

```
(defn step
  "Takes a set of living cells and returns the next
generation (as a set too)."
  [living-cells]
  (letfn [(alive [[cell cnt]]
            (when (or (= cnt 3)
                      (and (= cnt 2) (living-cells cell)))
              cell))]
    (->> living-cells (mapcat neighbours) frequencies
      (keep alive) set)))
```

Neighbours!

```
(defn neighbours [[x y]]
  (for [dx [-1 0 1]
        dy (if (zero? dx) [-1 1] [-1 0 1])]
    [(+ x dx) (+ y dy)]))


(defn step
  "Takes a set of living cells and returns the next
generation (as a set too)."
  [living-cells]
  (letfn [(alive [[cell cnt]]
            (when (or (= cnt 3)
                      (and (= cnt 2) (living-cells cell)))
              cell))]
    (->> living-cells (mapcat neighbours) frequencies
      (keep alive) set)))
```

Drafting code

Drafting code

Don’t go to the details

Drafting code

Don’t go to the details

Draft high-level code you’d like to be able to write **to solve the problem**

Drafting code

Don’t go to the details

Draft high-level code you’d like to be able to write **to solve the problem**

Try to implement it

Drafting code

Don’t go to the details

Draft high-level code you’d like to be able to write **to solve the problem**

Try to implement it

Negociate between practicality of implementation and draft code

But it really hurts...

But it really hurts...

Ask for help

But it really hurts...

- Ask for help
  - #clojure on IRC

But it really hurts...

- Ask for help
  - #clojure on IRC
  - Stackoverflow

But it really hurts...

- Ask for help
  - #clojure on IRC
  - Stackoverflow
  - clojure google group

But it really hurts...

- Ask for help
  - #clojure on IRC
  - Stackoverflow
  - clojure google group
  - reach your local user group

But it really hurts...

- Ask for help
  - #clojure on IRC
  - Stackoverflow
  - clojure google group
  - reach your local user group
  - create your local user group

But it really hurts...

Ask for help

- #clojure on IRC
- Stackoverflow
- clojure google group
- reach your local user group
- create your local user group
- mail me christophe@cgrand.net