

Improving Clojure Usability for Introductory Course

Henry Fellows, Thomas Hagen, Sean Stockholm, Ryan McArthur,
and Elena Machkasova

*Minnesota Clojure Users Group Meeting
July 8, 2015*

Table of contents

- 1 Overview of the Project
- 2 Error Handling
- 3 Clojure's Graphical Library

Goals

- Integrate Clojure into an introductory CS class
- Currently use Racket
 - Limited teaching language
 - Difficult to make complex projects
 - Students hitting performance issues (!)

Why use Clojure?

- Used in industry
- Better on resume
- Support for concurrency
- Large community and excellent resources
- Excellent libraries (data processing, image recognition, graphical, musical)

Issues with Clojure

- Confusing error messages
- Lack of beginner-friendly graphics libraries in functional style
- Some misleading or confusing core functions (`conj`, `some`, `for`, string functions)

Error Messages

- Computers are literal
- Error messages are the only way to communicate when something goes wrong
- By the time an error is detected, a lot of the context is lost

Error Messages for Beginners

- Use terminology that beginners haven't been introduced to
- Beginner mistakes can lead to very complex errors
- Beginners don't read error messages
- Line number reporting needs to be accurate
- Very little (if any) systematic usability study of error messages
- Good error messages should lead to a correct fix (study of Racket error messages)

Clojure Error Messages

- Java exceptions
- Use not only Java types, but Java terminology: "cannot be cast", "null pointer"
- Very verbose
- Stack traces are long
- Compiler messages come with a separate cause

Example Clojure Error

```
Exception in thread "main" clojure.lang.ArityException:  
Wrong number of args (3) passed to: core/cons, compiling:  
(/tmp/form-init3025539740275626138.clj:1:72)  
at clojure.lang.Compiler.load(Compiler.java:7142)  
at clojure.lang.Compiler.loadFile(Compiler.java:7086)  
at clojure.main$load_script.invoke(main.clj:274)  
at clojure.main$init_opt.invoke(main.clj:279)  
at clojure.main$initialize.invoke(main.clj:307)  
at clojure.main$null_opt.invoke(main.clj:342)  
at clojure.main$main.doInvoke(main.clj:420)  
at clojure.lang.RestFn.invoke(RestFn.java:421)  
at clojure.lang.Var.invoke(Var.java:383)  
at clojure.lang.AFn.applyToHelper(AFn.java:156)  
at clojure.lang.Var.applyTo(Var.java:700)  
at clojure.main.main(main.java:37)
```

Our Error Messages

- We are not changing language definition
- A combination of two approaches:
 - Approach 1: overwrite common functions (`map`, `filter`, `+`) to add pre-conditions for precise parameter reporting
 - Approach 2: catch exceptions, match and replace the error message
- Filter the stack trace
- Avoid unfamiliar terminology
- Consistency within error messages
- Readable and short
- Future direction: adding hints for common sources of errors

Iterations of Error Message Phrasing

```
Exception in thread "main" clojure.lang.ArityException:  
Wrong number of args (3) passed to: core/cons, compiling:  
(/tmp/form-init3025539740275626138.clj:1:72)
```

```
Error: Wrong number of arguments (3) passed to a function cons.  
Found in file core.clj on line 108 in function -main.  
intro.core/-main (core.clj line 108)
```

```
Error: You cannot pass three arguments to a function cons, need  
two.  
Found in file core.clj on line 108 in function -main.  
intro.core/-main (core.clj line 108)
```

Unified Phrasing

- Original: `let` requires an even number of forms
- New: Parameters for `let` must come in pairs, but one of them does not have a match.
- Original: Vector arg to `map conj` must be a pair
- New: Each inner vector must be a pair: a key followed by a value.

Printing Arguments

- Asserts for functions:

Error: in function map the first argument 2 must be a function but is a number.

- How about this?

Error: in function map the second argument clojure.core\$_STAR_ must be a sequence, but is a function

Function name processing: the second argument *...

Printing Arguments (cont.)

- But what about `(map (range) inc)`?

In function `+`, the second argument

`clojure.lang.LazySeq@22` must be a number but is a sequence.

In function `+`, the second argument `(0 1 2 3 4 5 6 7 8 9...)` must be a number but is a sequence.

- Challenges (work in progress):
 - `(repeat (range))` - infinite sequence of infinite sequences
 - How many levels do we evaluate?
 - Need to evaluate arguments recursively:
`(clojure.core$_STAR_)` should be `(*)`.

Current Work

- Syntax errors vs. runtime errors
- Reporting line numbers
 - Sometimes a part of the message
 - Sometimes a part of the stacktrace
 - Sometimes neither
 - Different line number reporting for different methods of evaluation
- Common sources of errors (hints)

Future Work

- Integrating with an IDE and REPL
- User feedback loop
- Usability studies

What is Quil?

- Graphical Library for Clojure
- It can:
 - Draw shapes and images
 - Move objects on the screen
 - Make games, pictures, ect..

fun-mode

^

Quil

^

Clojure

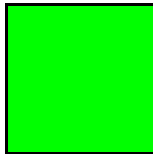
^

Java

Quil's fun-mode isn't enough

- Quil ONLY takes draw commands
- Quil doesn't separate the model from the view
- Quil code can get confusing and long

```
(q/fill 80 255 80)  
(q/rect 100 100 50 50)  
(q/no-fill)  
(q/no-stroke)
```



Designing super-fun-mode

- Built on top of fun-mode
- Gives students functions, colors, images, ect..
- Easy to read and change program code
- Allows for easy complex shapes

```
super-fun-mode  
^  
fun-mode  
^  
Quil  
^  
Clojure  
^  
Java
```

How super-fun-mode works

- You start by creating a shape

```
(def red-square  
  (create-rect 50 50 :red))
```
- Note that creating a shape does not draw it
- From there, you can draw the shape

```
(draw-shape red-square 500 500)
```



How super-fun-mode works technically

- Underneath, super-fun-mode builds a hashmap or a vector of hashmaps (in the case of complex shapes) with holds relevant information including:
 - The shape's width and height
 - The complex shape's width and height
 - The rotation angle of the shape
 - The function to draw the shape

super-fun-mode complex shapes

- You can put shapes together to make complex shapes

```
(def tower  
  (above red-square  
         orange-square  
         yellow-square  
         green-square  
         blue-square  
         violet-square))
```



Six squares

- The difference becomes quite apparent with complexity



Quil code

```
(let [x 100
      numb 6
      dist (+ 100 (* (\ numb 2) 50)))]
  (q/fill 80 255 80)
  (q/rect (- dist (* 1 50)) 100 50 50)
  (q/rect (- dist (* 2 50)) 100 50 50)
  (q/rect (- dist (* 3 50)) 100 50 50)
  (q/rect (- dist (* 4 50)) 100 50 50)
  (q/rect (- dist (* 5 50)) 100 50 50)
  (q/rect (- dist (* 6 50)) 100 50 50))
(q/no-fill)
```


Our code

```
(def lime-rect
  (create-rect 50 50 :lime))

(def lime-rectangles
  (beside
    lime-rect lime-rect lime-rect
    lime-rect lime-rect lime-rect))
```

Rotation and scaling

- You can modify the size and orientation of the shape

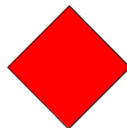
```
(rotate-shape red-square 45)
```



```
(scale-shape red-square 2 2)
```



```
(rotate-shape  
  (scale-shape red-square 2 2)  
  45)
```



Overlaying and complex shapes

- You can put shapes on top of each other

```
(overlay window roof)
```



```
(overlay-align :bottom :center  
  door  
  red-rect)
```



```
(scale-shape  
  (above (overlay top bottom))  
  1.4 1.4)
```



Other complex functions

- You can orient your besides and aboves as well

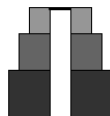
```
(beside-align :top  
  tower  
  tight-rope  
  tower)
```



```
(above-align :right  
  block1  
  block1.3  
  block1.6)
```



```
(beside-align :top  
  tower-aligned-R  
  tight-rope  
  tower-aligned-L)
```



The draw-shape function

- Draw-shape (or ds, either work) is the function that handles taking in these shape objects and drawing the elements of them appropriately.

```
(draw-shape  
  (create-rect 200 200 :red)  
  400 400)
```

```
(draw-shape  
  (beside box1  
          box2  
          circle  
          triangle))
```

The draw-shape function

```
{:w w
 :h h
 :tw w
 :th h
 :dx 0
 :dy 0
 :angle 0
 :ds (fn [x y pict wid hei cs angle]
       (*large cond to check fill/stroke here*)
       (with-translation [x y]
         (with-rotation
          [(/ (* PI angle) 180)]
          (f-rect 0 0 wid hei)))
       (no-fill)))}
```

Our direction

- Less paintbrush, more collage
- Create shapes, not just draw them
- Easier student code
- Give students an idea of how good software should be built

A few examples

Please Enjoy a Few Live Examples

Future work

- Fill out more functionality
 - Rotate more complex shapes
 - Pixel-detail Overlay and Overlay-Align
 - More seamless integration with Quil fun-mode
- Open Source the project
- Integrate a Clojure sound library

Acknowledgments

Our research was sponsored by:

- HHMI
- LSAMP

Thank you!
Any questions?