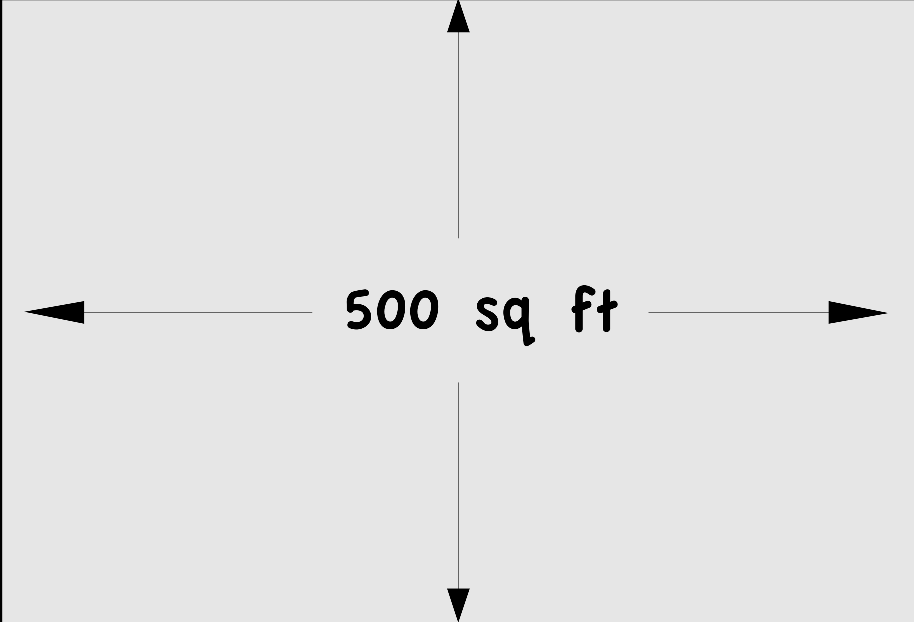
Macro Club

@stuartsierra #clojureconj


500 sq ft

I

( )

THE EXPERT'S VOICE® OPEN SOURCE

Practical Clojure

Full Introduction to Clojure, a full Lisp variant for the JVM

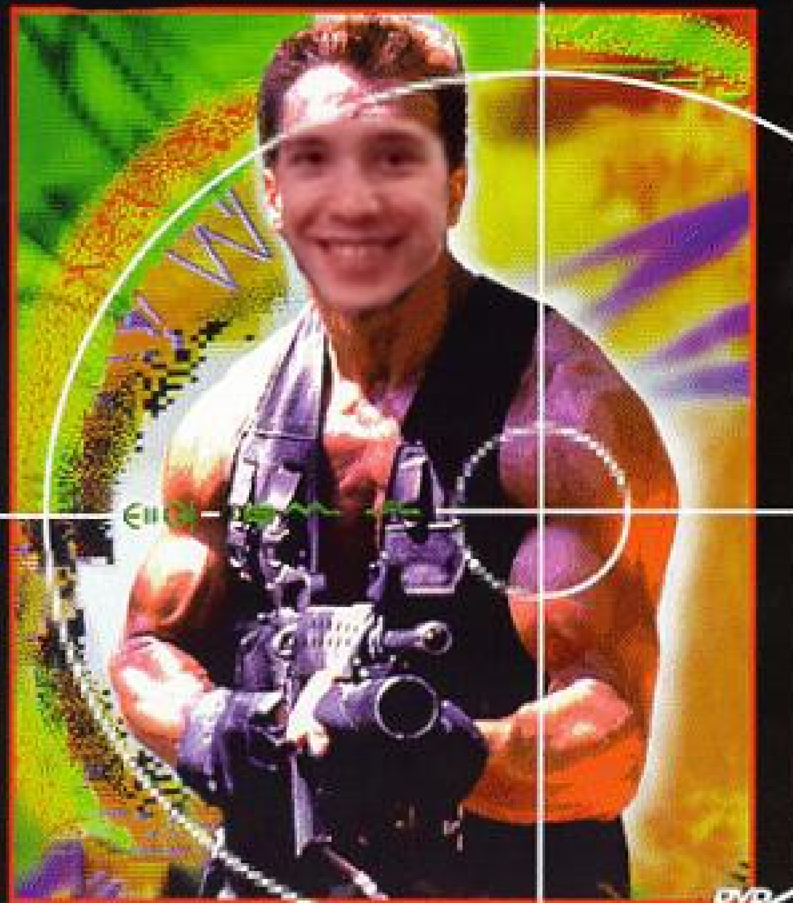
Luke VanderHart and Stuart Sierra

Apress®

“If it seems simple, you're probably doing it wrong.”

“You can pry EMACS from my cold, dead fingers.”

SIERRA
PAREDITOR



The First Rule of Macro Club

You do not write macros.

“Macros are harder to write than ordinary Lisp functions, and it's considered to be bad style to use them when they're not necessary.”

-Paul Graham, “Beating the Averages”

I

` ~@

clojure.test

```
(is (= 4 (+ 2 2)))

true
```

clojure.test

```clojure
(macroexpand '(is (= 4 (+ 2 2))))
```

```clojure
(try
  (let [values (list 4 (+ 2 2))
        result (apply = values)]
    (if result
      (do-report {:type :pass, ...})
      (do-report {:type :fail, :actual values, ...}))
    result)
  (catch java.lang.Throwable t
    (do-report {:type :error, :actual t, ...})))
```

clojure.test

```
(macroexpand
 '(is (thrown? ArithmeticException (/ 1 0))))

(try
  (try
    (/ 1 0)
    (do-report {:type :fail, :actual nil, ...})
    (catch ArithmeticException e
      (do-report {:type :pass, :actual e, ...})
      e))
  (catch java.lang.Throwable t
    (do-report {:type :error, :actual t, ...})))
```

clojure.test

```
(thrown? ArithmeticException (/ 1 0))))

#<CompilerException java.lang.Exception:
Unable to resolve symbol: thrown? in this context>
```

clojure.test

```
(defmacro is [message form]
  `(try ~(assert-expr message form)
        (catch Throwable t#
          (do-report {:type :error, ...}))))

(defmulti assert-expr (fn [msg form] (first form)))

(defmethod assert-expr '= [msg form] ...)

(defmethod assert-expr 'thrown? [msg form] ...)
```

The Second Rule of Macro Club

You do not write macros…
that violate expectations
of normal code behavior.

Lazytest

github.com/stuartsierra/lazytest

The Third Rule of Macro Club

If this is your first time,
you must write a macro.

lazytest.expect

```
(defmacro expect
  ([expr] `(expect nil ~expr))
  ([docstring expr]
     {:pre [(or (string? docstring) (nil? docstring))]}
     (if (function-call? expr)
       ;; Normal function call
       `(let [f# ~(first expr)
              args# (list ~@(rest expr))
              result# (apply f# args#)]
          (or result#
              (throw (ExpectationFailed.
                      (merge '~(meta &form)
                             '~(meta expr)
                             {:form '~expr
                              :locals ~(local-bindings &env)
                              :evaluated (list* f# args#)
                              :result result#
                              :file ~*file*
                              :ns '~(ns-name *ns*)}
                             ~(when docstring {:doc docstring}))))))
       ;; Unknown type of expression
       `(let [result# ~expr]
          (or result#
              (throw (ExpectationFailed.
                      (merge '~(meta &form)
                             '~(meta expr)
                             {:form '~expr
                              :locals ~(local-bindings &env)
                              :result result#
                              :file ~*file*
                              :ns '~(ns-name *ns*)}
                             ~(when docstring {:doc docstring})))))))))
```

lazytest.expect

```
(expect (= 4 (+ 2 2)))


;; If that's not true...
(throw
 (ExpectationFailed.
   {:form       ; what you passed in
    :evaluated ; each component, evaluated
    :result     ; return value
    :locals     ; local bindings
    :doc        ; optional docstring
    :file       ; file name
    :line       ; line number
    :ns         ; namespace name
   }))
```

lazytest.expect

```java
package lazytest;

public class ExpectationFailed
               extends AssertionError {

    public final Object reason;

    public ExpectationFailed(Object reason) {
        this.reason = reason;
    }
}
```

Test Case

Something that can be run.

Result: “pass” or “fail”

Suite

Collection of Test Cases

and/or other Suites.

clojure.test Grouping

```
(deftest test-one ...)

(deftest test-two ...)


(deftest my-test-group
  (test-one)
  (test-two))
```

I ♥

Protocols

```clojure
(defprotocol Testable
  (run-tests [this]
    "Run tests in this suite."))

(deftype Suite [children]
  Testable
   (run-tests [this]
     (map run-tests children)))

(deftype TestCase [f]
  Testable
   (run-tests [this]
     (try (f) ...)))
```

The Fourth Rule of Macro Club

Don't think
object-oriented.

“Objects are a poor man's closures.”

-Lisp dude

“Closures are a poor man's objects.”

-OOP dude

```clojure
(defprotocol Testable
  (run-tests [this]
    "Run tests in this suite."))

(deftype Suite [children]
  Testable
   (run-tests [this]
     (map run-tests children)))

(deftype TestCase [assertion]
  Testable
   (run-tests [this]
     (try (assertion) ...)))
```

Test Case

Something that can be run.

Result: “pass” or “fail”

Test Case

```
(fn [] (expect ...))
```

Suite

Collection of Test Cases

and/or other Suites.

Suite

```
(fn []
  (seq ... test cases ...
       ... and/or suites ...))
```

Suite

```
(defmacro describe [doc & children]
  `(fn []
     (with-meta
       (list ~@children)
       {:doc ~doc
        :file ...
        :line ...})))
```

Test Case

```
(defmacro it [docstring expr]
  `(with-meta
     (fn [] (expect ~expr))
     {:doc ~docstring
      :file ...
      :line ...}))
```

“DSL”

```
(describe "A list"
  (it "is a sequence"
    (seq? (list)))
  (it "is countable"
    (= 3 (count (list 1 2 3))))
  (it "grows at the front"
    (= (list 2 1) (cons 2 (list 1)))))
```

Context

... set up some state ...

... make assertions about it ...

... clean up ...

clojure.test “Fixtures”

```
(defn my-fixture [test-fn]
  ... stuff that happens before ...
  (test-fn)
  ... stuff that happens after ...)

(use-fixtures :each my-fixture)
```

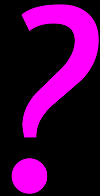
clojure.test “Fixtures”

```
(declare *state*)

(defn my-fixture [test-fn]
  (binding [*state* ...setup...]
    (test-fn)
    ... cleanup state ...))
```

The Fifth Rule of Macro Club

No shirt,
No shoes,
No dynamic scope.



Context

... set up some state ...

... make assertions about it ...

... clean up ...

Context

```
(defprotocol Context
  (setup [this]
    "Setup and return some state")
  (teardown [this]
    "Clean up state"))
```

“System Property” Context

```
(deftype PropertyContext [name value]
  Context
    (setup [this]
      ... set Java system property ...)
    (teardown [this]
      ... restore original value ...))
```

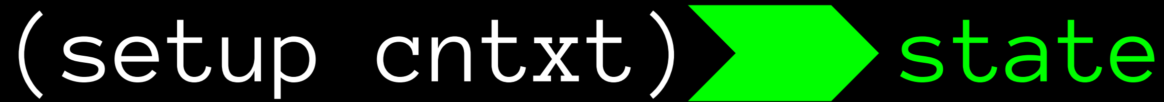
“Temp File” Context

```
(deftype FileContext []
  Context
    (setup [this]
      ... create temporary file ...)
    (teardown [this]
      ... delete file ...))
```

“Alter Var Root” Context

```
(deftype GlobalStubContext [var value]
  Context
    (setup [this]
       ... set root binding of var ...)
    (teardown [this]
       ... restore original value ...))
```

Context:



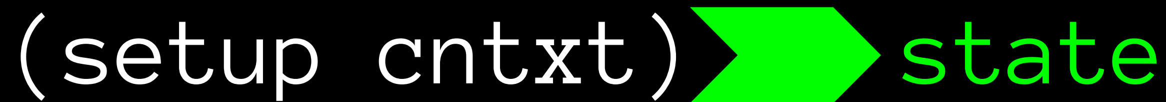
Test Case:

```
(fn [] (expect (= state ...)))
```

Context:

```
(teardown cntxt)
```

Context:



Test Case:

```
(fn [state] (expect (= state ...)))
```

Context:

```
(teardown cntxt)
```

"DSL"

```
(describe "A list"
  (it "is a sequence"
    (seq? (list)))
  (it "is countable"
    (= 3 (count (list 1 2 3))))
  (it "grows at the front"
    (= (list 2 1) (cons 2 (list 1)))))
```

“Stateful” Context

```
(deftype StatefulContext [...]
  Context
      (setup [this]
         ... create state ...)
      (teardown [this]
         ... destroy state ...)
  clojure.lang.IDeref
      (deref [this]
         ... return state ...))
```

“Stateful” Context

```
(stateful-fn-context
  (fn [] ... create & return state...)
  (fn [state] ... destroy state ...))
```

“Stateful” Context

```
(def cntxt (stateful-fn-context ...))

(setup cntxt)

;; get the state:
@cntxt

(teardown cntxt)
```

“DSL”

```
(def cntxt (stateful-fn-context ...))

(describe "..."
  (with [cntxt]
    (it "..."
      ... @cntxt ...)
    (it "..."
      ... @cntxt ...)))
```

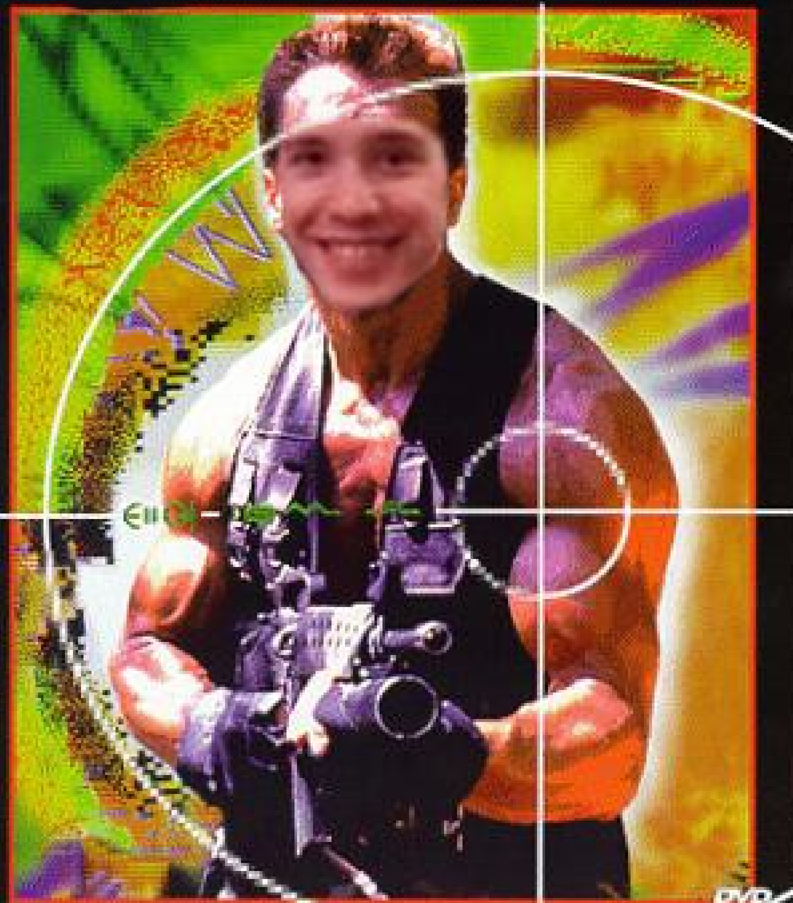
The Sixth Rule of Macro Club

You don't create
new scoping rules.

Bullet Points

- Build abstractions on Clojure's existing types
- Functions are the most powerful abstraction
- Use macros for syntactic sugar
- Don't break expectations of normal code
- Don't think OOP

SIERRA
PAREDITOR



#clojureconj

@stuartsierra

stuartsierra.com clojure.com

github.com/stuartsierra

Practical Clojure