Macros: Why, When, and How

Gary Fredericks

(@gfredericks_)

Why talk about macros?

- Key to my understanding of how Clojure works
- The major selling point of Lisp
  - Safe metaprogramming!
- Can be intimidating
  - Must mentally separate compile-time from runtime
    - And read-time! haha! oh dear.
  - Syntax-quote looks like a steaming pile of perl
- Appropriate use is a subtle issue

What we will talk about

- Preliminary concepts (code is data!)
- How Clojure macros work (functions on code!)
- When to write macros (sometimes!)
- What syntax-quote (`) does (three things!)

Code as Data

Data

Has just one meaning

```
1:  {:name "Jack Kemp"
2:   :birthdate [1935 7 13]
3:   :favorite-things #{:marmelade :marmite
4:                      :marmots   :marmosets}}
```

Code has two meanings

```
1:  (defn secure-password?
2:    "Checks if the password
3:     is totes uncrackable."
4:    [pw]
5:    (and (> (count pw) 6)
6:         (.contains pw "$")
7:         (.contains pw "1")))
```

Obtaining forms: `quote`

```
1:  (* 2 3 7)          ;; => 42
2:
3:  (quote (* 2 3 7)) ;; => (* 2 3 7)
4:
5:  '(* 2 3 7)         ;; => (* 2 3 7)
6:
7:  '(This is a list with (+ 5 2) elements)
8:  ;; => '(This is a list with (+ 5 2) elements)
```

Obtaining forms: `read-string`

```
 1:  (read-string "(* 2 3 7)")
 2:  ;; => (* 2 3 7)
 3:
 4:  (read-string "foo")
 5:  ;; => foo
 6:
 7:  (type (read-string "foo"))
 8:  ;; => clojure.lang.Symbol
 9:
10:  (read-string "'foo")
11:  ;; => (quote foo)
```

Building and Manipulating forms 1

```
1:  (reverse '(* 2 3 7))
2:  ;; => (7 3 2 *)
3:
4:  (take 2 '(* 2 3 7))
5:  ;; => (* 2)
6:
7:  (let [num (+ 3 2)]
8:    '(This list has num elements))
9:  ;; => (This list has num elements)
```

Building and Manipulating forms 2

```
1:  (let [num (+ 3 2)]
2:    (list 'This 'list 'has num 'elements))
3:  ;; => (This list has 5 elements)
```

`eval` ing forms 1

```
1:  (eval '(* 2 3 7)) ;; => 42
2:
3:  (eval (reverse '(5 37 +)))
4:  ;; => 42
5:
6:  (def does-not-compile '(* 2 3 x))
7:
8:  (eval does-not-compile)
9:  ;; CompilerException java.lang.RuntimeException:
10: ;; Unable to resolve symbol: x in this context
```

`eval`ing forms 2

```
1:  (list 'let '[x 7] does-not-compile)
2:  ;; => (let [x 7] (* 2 3 x))
3:
4:  (eval (list 'let '[x 7] does-not-compile))
5:  ;;=> 42
```

(eval (eval '''foo))

```
1:  ''foo ;; => (quote foo)
2:
3:  ''''foo
4:  ;; => (quote (quote (quote foo)))
5:
6:  (eval ''foo) ;; => foo
7:
8:  (eval (eval '''foo)) ;; => foo
9:
10: (eval (list 'quote 'foo)) ;; => foo
```

Wat a macro is?

A macro is a function the compiler calls with forms as arguments, and expects a form to be returned.

Macro calls are replaced at compile time with whatever the macro returns.

Implication: macros are virtually never necessary to make your code do something.

Macro call	Expanded code
`(if-not b v1 v2)`	`(if (not b) v1 v2)`
`(when b s1 s2)`	`(if b (do s1 s2) nil)`

Project Euler #4

A palindromic number reads the same both ways.

The largest palindrome made from the product of two 2-digit numbers is `9009 = 91 * 99`

Find the largest palindrome made from the product of two 3-digit numbers.

Partial solution

```
(for [x (range 100 1000),
      y (range 100 x),
      :let [z (* x y)]
      :when (palindrome? z)]
  z)
```

clojure.core/for

```
(let [iter__4609__auto__
      (fn iter__1163
        [s__1164]
        (lazy-seq
         (loop [s__1164 s__1164]
           (when-first [x s__1164]
             (let [iterys__4605__auto__
                   (fn iter__1165
                     [s__1166]
                     (lazy-seq
                      (loop [s__1166 s__1166]
                        (when-let [s__1166 (seq s__1166)]
                          (if (chunked-seq? s__1166)
                            (let [c__4607__auto__
                                  (chunk-first s__1166)
                                  size__4608__auto__
                                  (int (count c__4607__auto__))
                                  b__1168
                                  (chunk-buffer size__4608__auto__)]
                              (if (loop [i__1167 (int 0)]
                                    (if (< i__1167 size__4608__auto__)
                                      (let [y (.nth c__4607__auto__ i__1167)]
                                        (let [z (* x y)]
                                          (if (palindrome? z)
                                            (do
                                              (chunk-append b__1168 z)
                                              (recur (unchecked-inc i__1167)))
                                            (recur (unchecked-inc i__1167)))))
                                      true))
                                (chunk-cons
                                 (chunk b__1168)
                                 (iter__1165 (chunk-rest s__1166)))
                                (chunk-cons
                                 (chunk b__1168)
                                 nil)))
                            (let [y (first s__1166)]
                              (let [z (* x y)]
                                (if (palindrome? z)
                                  (cons z (iter__1165 (rest s__1166)))
                                  (recur (rest s__1166)))))))))))
                   fs__4606__auto__
                   (seq (iterys__4605__auto__ (range 100 x)))]
               (if fs__4606__auto__
                 (concat
                  fs__4606__auto__
                  (iter__1163 (rest s__1164)))
                 (recur (rest s__1164))))))))]
  (iter__4609__auto__ (range 100 1000)))
```

Macro Mechanics

Defining a pseudo-macro with `defn`

```
1:  (defn unless
2:    "Takes three expressions and
3:     returns a new expression."
4:    [condition false-case true-case]
5:    (list 'if
6:          condition
7:          true-case
8:          false-case))
```

Using `unless` (1)

```
1:  (unless (= 1 2)
2:    (println "Not equal")
3:    (println "Equal"))
4:
5:  ;; Prints:
6:  ;;   Not Equal
7:  ;;   Equal
8:  ;;
9:  ;; Returns: (if false nil nil)
```

Using `unless` (2)

```
1:  (unless '(= 1 2)
2:    '(println "Not equal")
3:    '(println "Equal"))
4:
5:  ;; Returns: (if (= 1 2) (println "Equal") (println "Not equal"))
```

Using `unless` (3)

```
1:  (eval (unless '(= 1 2)
2:          '(println "Not equal")
3:          '(println "Equal")))
4:
5:  ;; Prints:
6:  ;;   Not Equal
```

`unless` as a proper macro

```
1:  (defn unless
2:    [condition false-case true-case]
3:    (list 'if
4:          condition
5:          true-case
6:          false-case))
```

```
1:  (defmacro unless
2:    [condition false-case true-case]
3:    (list 'if
4:          condition
5:          true-case
6:          false-case))
```

Using `unless` (4)

```
1:  (unless (= 1 2)
2:    (println "Not equal")
3:    (println "Equal"))
4:
5:  ;; Prints:
6:  ;;   Not Equal
```

Debugging `unless`

```
1:  (macroexpand-1 '(unless (= 1 2)
2:                    (println "Not equal")
3:                    (println "Equal")))
4:
5:  ;; => (if (= 1 2) (println "Not equal") (println "Equal"))
```

Defining spy

```
1:  ;; Goal:
2:
3:  (spy (* 2 3 7))
4:
5:  ;; should print:
6:  ;;   (* 2 3 7) is 42
7:  ;;
8:  ;; and return
9:  ;;   42
```

spy as a function (1)

```
 1:  (defn spy
 2:    [expr]
 3:    (println expr "is" expr)
 4:    expr)
 5:
 6:  (spy (* 2 3 7))
 7:  ;; Prints:
 8:  ;;   42 is 42
 9:  ;; And returns:
10:  ;;   42
```

spy as a function (2)

```
 1:  (defn spy
 2:    [expr]
 3:    (println 'expr "is" expr)
 4:    expr)
 5:
 6:  (spy (* 2 3 7))
 7:  ;; Prints:
 8:  ;;   expr is 42
 9:  ;; And returns:
10:  ;;   42
```

spy as a function (3)

```
 1:  (defn spy
 2:    [expr value]
 3:    (println expr "is" value)
 4:    value)
 5:
 6:  (spy '(* 2 3 7) (* 2 3 7))
 7:  ;; Prints:
 8:  ;;   (* 2 3 7) is 42
 9:  ;; And returns:
10:  ;;   42
```

spy as a macro – desired expansion

```
1:  (spy (* 2 3 7))
2:
3:  ;; should expand to
4:
5:  (let [val (* 2 3 7)]
6:    (println '(* 2 3 7) "is" val)
7:    val)
```

spy as a macro – first try

```
1:  (defmacro spy
2:    [expr]
3:    (list 'let
4:          ['val expr]
5:          (list 'println 'expr "is" 'val)
6:          'val))
```

```
1:  (macroexpand-1 '(spy (* 2 3 7)))
2:
3:  ;; Returns:
4:
5:  (let [val (* 2 3 7)]
6:    (println expr "is" val)
7:    val)
```

spy as a macro – second try

```
1:  (defmacro spy
2:    [expr]
3:    (list 'let
4:          ['val expr]
5:          (list 'println expr "is" 'val)
6:          'val))
```

```
1:  (macroexpand-1 '(spy (* 2 3 7)))
2:
3:  ;; Returns:
4:
5:  (let [val (* 2 3 7)]
6:    (println (* 2 3 7) "is" val)
7:    val)
```

spy as a macro – third try

```
1:  (defmacro spy
2:    [expr]
3:    (list 'let
4:          ['val expr]
5:          (list 'println ''expr "is" 'val)
6:          'val))
```

```
1:  (macroexpand-1 '(spy (* 2 3 7)))
2:
3:  ;; Returns:
4:
5:  (let [val (* 2 3 7)]
6:    (println (quote expr) "is" val)
7:    val)
```

spy – when ' is confusing

```
1:  (let [val (* 2 3 7)]
2:    (println '(* 2 3 7) "is" val)
3:    val)
```

```
1:  (let [val (* 2 3 7)]
2:    (println (quote (* 2 3 7)) "is" val)
3:    val)
```

spy as a macro – fourth try

```
1:  (defmacro spy
2:    [expr]
3:    (list 'let
4:          ['val expr]
5:          (list 'println
6:                (list 'quote expr)
7:                "is"
8:                'val)
9:          'val))
```

```
1:  (macroexpand-1 '(spy (* 2 3 7)))
2:
3:  ;; Returns:
4:
5:  (let [val (* 2 3 7)]
6:    (println (quote (* 2 3 7)) "is" val)
7:    val)
```

Syntax-quote Preview

```
1:  (defmacro spy
2:    [expr]
3:    (list 'let
4:          ['val expr]
5:          (list 'println
6:                (list 'quote expr)
7:                "is"
8:                'val)
9:          'val))
```

```
1:  (defmacro spy
2:    [expr]
3:    `(let [val# ~expr]
4:       (println '~expr "is" val#)
5:       val#))
```

What can't you do with macros?

- Customize or extend reader syntax
  - New data structure syntax
  - I want `@` to mean something else in this expression
- Change the behavior of code you don't control
  - I want all the clojure.core functions to log their execution times
- Magically change things outside the scope of a macro-call
- Change macro-precedence

Macros??

- Why not to write macros
- Commonly tolerated macro usages
- Tips for avoiding macros
- Tips for writing tolerable macros

Don't Write Macros

"The first rule of Macro Club is
Don’t Write Macros."

-- Stuart Halloway

Macros are not Functions

Macros cannot be composed at runtime.

```
1:  (reduce or [false true false])
2:  ;; => CompilerException java.lang.RuntimeException:
3:  ;;    Can't take value of a macro: #'clojure.core/or
```

but you can…

```
1:  (reduce #(or %1 %2) [false true false])
2:  ;; => true
```

Macros beget more macros

```
1:  (defmacro macro-reduce
2:    [macro-name coll]
3:    `(reduce #(~macro-name %1 %2) ~coll))
4:
5:  (macro-reduce or  [false true false])  ;; => true
6:  (macro-reduce or  [false false false]) ;; => false
7:  (macro-reduce and [false true false])  ;; => false
8:  (macro-reduce do  [false true false])  ;; => false
9:  (macro-reduce do  [false true false])  ;; => false
```

Macros beget more more macros

```
1:  ;; In clojure.test
2:
3:  (is (= 42 (* 2 3 7)))
4:
5:  ;; But I want:
6:
7:  (is= 42 (* 2 3 7))
```

Macros can make code hard to understand

The reader has to understand the behavior of each macro individually to know what a piece of code is doing at the syntactic level.

Don't Write Macros (until it hurts)

- Macros are not functions
- Macros tend to result in more macros
- Macros require special-case understanding

Commonly Tolerated Macro Usages 1

- Wrapping execution: `with-foo`
  - `with-redefs`, `with-open`, `with-out-str`, `time`, `dosync`
- Delaying execution
  - `delay`, `future`, `lazy-seq`
- Defing things
  - `defn`, `defmacro`, `defmulti`, `defprotocol`, `defrecord`, `deftype`
  - `deftest` (clojure.test), `defproject` (leiningen)

Commonly Tolerated Macro Usages 2

- Capturing Code
  - `assert`, `spy`, `is` (clojure.test)
- DSLs (Korma, Compojure, midge)
- Compile-time Optimizations
  - Hiccup
    - `(html [:ul [:li foo] [:li {:id "7"} "WAT"]])`
  - String interpolation (clojure.core.strint)
    - `(<< "You have $~(double (/ x 100)) left.")`
  - `comment`
  - `assert`

Commonly Tolerated Macro Usages 3

- Implementing entirely different paradigms
  - Logic Programming (core.logic)
  - Concatenative Programming (factjor)

Abstinence Tips

- Learn and prefer functional patterns
  - Function decorators instead of wrapper macros (e.g. ring, clojure.test fixtures)
- Learn about the macros clojure already has
  - 1.5 introduced `cond->`, `some->`, and `as->`
- Tolerate a bit of repetition for the sake of clarity

Tips for Writing Tolerable Macros

- Use helper functions!
  - Many macros can be written in one or two lines by deferring to a helper function for most of the work
- Use naming conventions
  - Adverbs for execution-wrapping
  - `def-foo` if you def something
    - Though consider `(def foo (macro-call))` instead
- Don't def more than one thing
- Only introduce locals named by the user
  - `(dotimes [n 10] (foo n))`, `(run* [q] ...)`
- No side effects

Syntax-quote

Syntax-quote

`` ` `` is an enhanced `'`

`` ` `` is independent of macros, but not really useful for anything else.

~~Complects~~ Combines three different functionalities:

- Unquote
- Symbol qualification
- Gensym

Unquote: Problem

This is difficult to read. The shape of the final code gets lost in the calls to `list`.

```
1:  (defmacro spy
2:    [expr]
3:    (list 'let
4:          ['val expr]
5:          (list 'println (list 'quote expr) "is" 'val)
6:          'val))
```

Unquote: Resolution

```
1:  (defmacro spy
2:    [expr]
3:    (list 'let
4:          ['val expr]
5:          (list 'println
6:                (list 'quote expr)
7:                "is" 'val)
8:          'val))
```

```
1:  (defmacro spy
2:    [expr]
3:    `(let [val ~expr]
4:       (println '~expr "is" val)
5:       val))
6:
7:  ;; line 4 is equivalent to
8:  ;;
9:  ;; (println (quote ~expr) "is" val)
```

Unquote-Splicing: Problem

Often we have a list of expressions that we want to insert somewhere

```
 1:  ;; We want
 2:  (returning (slurp "data.csv")
 3:    (reset! running false)
 4:    (println "Done reading file"))
 5:
 6:  ;; to expand to
 7:  (let [val (slurp "data.csv")]
 8:    (reset! running false)
 9:    (println "Done reading file")
10:    val)
```

Unquote-Splicing: First try

```
1:  (defmacro returning
2:    [expr & side-effects]
3:    `(let [val ~expr]
4:       ~side-effects
5:       val))
```

```
1:  (macroexpand-1
2:   '(returning x (foo) (bar)))
3:
4:  ;; returns:
5:
6:  (let [val x]
7:    ((foo)
8:     (bar))
9:    val)
```

Unquote-Splicing: Second try

```
1:  (defmacro returning
2:    [expr & side-effects]
3:    (concat ['let ['val expr]]
4:            side-effects
5:            ['val]))
```

```
1:  (macroexpand-1
2:   '(returning x (foo) (bar)))
3:
4:  ;; returns:
5:
6:  (let [val x]
7:    (foo)
8:    (bar)
9:    val)
```

Unquote-Splicing: Third try

```
1:  (defmacro returning
2:    [expr & side-effects]
3:    `(let [val ~expr]
4:       ~@side-effects
5:       val))
```

```
1:  (macroexpand-1
2:   '(returning x (foo) (bar)))
3:
4:  ;; returns:
5:
6:  (let [val x]
7:    (foo)
8:    (bar)
9:    val)
```

Unquote Debugging

Syntax-quote can be used outside the context of macros

```
1:  `(1 2 3 (+ 4 5) 6 ~(+ 7 8))
2:
3:  ;; => (1 2 3 (+ 4 5) 6 15)
4:
5:  (let [nums [5 6 7 8]]
6:    `(1 2 ~@nums ~nums))
7:
8:  ;; => (1 2 5 6 7 8 [5 6 7 8])
```

Symbol Qualification: Problem

Using a macro defined in another namespace:

```
1:  (ns my.macros)
2:
3:  (defmacro returning
4:    [expr & side-effects]
5:    `(let [val ~expr]
6:       ~@side-effects
7:       val))
```

```
1:  (ns my.code
2:    (:refer-clojure :exclude [let])
3:    (:require [my.macros :refer [returning]]
4:              [other.lib :refer [let]]))
5:
6:  (defn main
7:    []
8:    (returning (* 2 3 7)
9:      (println "Computed special number")))
```

Symbol Qualification: Resolution

Syntax-quote automatically fully-qualifies symbols based on the current environment.

```
1:  `first   ;; => clojure.core/first
2:  `foo     ;; => user/foo
3:  `if      ;; => if
4:
5:  `(+ 1 2) ;; => (clojure.core/+ 1 2)
```

Symbol Qualification: Resolution 2

Using a macro defined in another namespace:

```
1:  (ns my.macros)
2:
3:  (defmacro returning
4:    [expr & side-effects]
5:    `(let [val ~expr]
6:       ~@side-effects
7:       val))
```

```
1:  (macroexpand-1
2:   '(returning (* 2 3 7)
3:      (println "Computed special number")))
4:
5:  ;; returns (sort of):
6:
7:  (clojure.core/let [val (* 2 3 7)]
8:    (println "Computed special number")
9:    val)
```

Gensym: Problem

Macros that create locals might accidentally shadow things

```
1:  (defn do-math
2:    [val]
3:    (returning (* 7 val)
4:      (println "Just multiplied 7 with" val)))
5:
6:  (do-math 2)
7:
8:  ;; prints:
9:  ;; Just multiplied 7 with 14
```

Gensym: Problem 2

```
 1:  (defn do-math
 2:    [val]
 3:    (returning (* 7 val)
 4:      (println "Just multiplied 7 with" val)))
 5:
 6:  ;; Effectively expands to:
 7:
 8:  (defn do-math
 9:    [val]
10:    (let [val (* 7 val)]
11:      (println "Just multiplied 7 with" val)
12:      val))
```

Gensym: Solution

Any symbols that end in `#` are expanded to gensyms

```
1:  `foo# ;; => foo__1179__auto__
2:
3:  `[foo# bar# foo#] ;; => [foo__1184__auto__
4:                    ;;     bar__1185__auto__
5:                    ;;     foo__1184__auto__]
6:
7:  [`foo# `foo#] ;; => [foo__1188__auto__
8:                ;;     foo__1189__auto__]
```

Gensym: Solution 2

```
 1:  (defmacro returning
 2:    [expr & side-effects]
 3:    `(let [val# ~expr]
 4:       ~@side-effects
 5:       val#))
 6:
 7:  ;; Effectively expands to:
 8:  (defn do-math
 9:    [val]
10:    (let [val__1168__auto__ (* 7 val)]
11:      (println "Just multiplied 7 with" val)
12:      val__1168__auto__))
```

Gensym: When to use?

Whenever you create a local in your macro definition.

- `let`, `loop`
- Arguments to a function
- Any other macro that expands to one of the above

It's difficult to miss, because if you forget to use it you will end up with a fully-qualified symbol that will likely not compile.

Syntax-quote: Reference

Syntax	What it does
`~foo`	insert `foo` unquoted
`~@foo`	insert `foo` unquoted and splice its elements in
`foo`	fully-qualified symbol based on what `foo` refers to in the local context
`foo#`	gensym, same as other uses of `foo#` in the same syntax-quote expression

Syntax-quote: All Together Now

```
(defmacro spy
  "Prints a debug statement with the
   given form and its value, and
   returns the value."
  [expr]
  `(let [val# ~expr]
     (println '~expr "is" val#)
     val#))
```

```
(macroexpand-1 '(spy (* 2 3 7)))

;; actually actually returns:

(clojure.core/let
    [val__1133__auto__ (* 2 3 7)]
  (clojure.core/println
   (quote (* 2 3 7))
   "is"
   val__1133__auto__)
  val__1133__auto__)
```

Macro Fun

Nested Syntax-quotes

```
``foo ;; => (quote user/foo)

```
```foo
;; => (clojure.core/seq
;;     (clojure.core/concat (clojure.core/list (quote quote))
;;                          (clojure.core/list (quote user/foo))))
```

```
`````foo
```

```
(clojure.core/seq (clojure.core/concat (clojure.core/list (quote clojure.core/seq)) (clojure.core/list (clojure.core/seq (clojure.core/concat (clojure.core/list (quote
clojure.core/concat)) (clojure.core/list (clojure.core/seq (clojure.core/concat (clojure.core/list (quote clojure.core/list)) (clojure.core/list (clojure.core/seq
(clojure.core/concat (clojure.core/list (quote quote)) (clojure.core/list (quote clojure.core/seq))))))))) (clojure.core/list (clojure.core/seq (clojure.core/concat
(clojure.core/list (quote clojure.core/list)) (clojure.core/list (clojure.core/seq (clojure.core/concat (clojure.core/list (quote clojure.core/seq)) (clojure.core/list
(clojure.core/seq (clojure.core/concat (clojure.core/list (quote clojure.core/concat)) (clojure.core/list (clojure.core/seq (clojure.core/concat (clojure.core/list (quote
clojure.core/list)) (clojure.core/list (clojure.core/seq (clojure.core/concat (clojure.core/list (quote quote)) (clojure.core/list (quote clojure.core/concat))))))))
(clojure.core/list (clojure.core/seq (clojure.core/concat (clojure.core/list (quote clojure.core/list)) (clojure.core/list (clojure.core/seq (clojure.core/concat
(clojure.core/list (quote clojure.core/seq)) (clojure.core/list (clojure.core/seq (clojure.core/concat (clojure.core/list (quote clojure.core/concat)) (clojure.core/list
(clojure.core/seq (clojure.core/concat (clojure.core/list (quote clojure.core/list)) (clojure.core/list (clojure.core/seq (clojure.core/concat (clojure.core/list (quote quote))
(clojure.core/list (quote clojure.core/list)))))))) (clojure.core/list (clojure.core/seq (clojure.core/concat (clojure.core/list (quote clojure.core/list)) (clojure.core/list
(clojure.core/seq (clojure.core/concat (clojure.core/list (quote clojure.core/seq)) (clojure.core/list (clojure.core/seq (clojure.core/concat (clojure.core/list (quote
clojure.core/concat)) (clojure.core/list (clojure.core/seq (clojure.core/concat (clojure.core/list (quote clojure.core/list)) (clojure.core/list (clojure.core/seq
(clojure.core/concat (clojure.core/list (quote quote)) (clojure.core/list (quote quote)))))))) (clojure.core/list (clojure.core/seq (clojure.core/concat (clojure.core/list (quote
clojure.core/list)) (clojure.core/list (clojure.core/seq (clojure.core/concat (clojure.core/list (quote quote)) (clojure.core/list (quote quote)))))))))))))))))))))))))
(clojure.core/list (clojure.core/seq (clojure.core/concat (clojure.core/list (quote clojure.core/list)) (clojure.core/list (clojure.core/seq (clojure.core/concat
(clojure.core/list (quote clojure.core/seq)) (clojure.core/list (clojure.core/seq (clojure.core/concat (clojure.core/list (quote clojure.core/concat)) (clojure.core/list
(clojure.core/seq (clojure.core/concat (clojure.core/list (quote clojure.core/list)) (clojure.core/list (clojure.core/seq (clojure.core/concat (clojure.core/list (quote quote))
(clojure.core/list (quote clojure.core/list)))))))) (clojure.core/list (clojure.core/seq (clojure.core/concat (clojure.core/list (quote clojure.core/list)) (clojure.core/list
(clojure.core/seq (clojure.core/concat (clojure.core/list (quote clojure.core/seq)) (clojure.core/list (clojure.core/seq (clojure.core/concat (clojure.core/list (quote
clojure.core/concat)) (clojure.core/list (clojure.core/seq (clojure.core/concat (clojure.core/list (quote clojure.core/list)) (clojure.core/list (clojure.core/seq
(clojure.core/concat (clojure.core/list (quote quote)) (clojure.core/list (quote quote)))))))) (clojure.core/list (clojure.core/seq (clojure.core/concat (clojure.core/list (quote
clojure.core/list)) (clojure.core/list (clojure.core/seq (clojure.core/concat (clojure.core/list (quote quote)) (clojure.core/list (quote
user/foo))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
```
```

Recursive ->>

```
(macroexpand-1 '(->> a b (->> c d)))

;; => (->> (->> a b) (->> c d))

(macroexpand-1
 (macroexpand-1 '(->> a b (->> c d))))

;; => (->> c d (->> a b))
```

(def defmacro ...)

```
(def

 ^{:doc "Like defn, but the resulting function name is declared as a
  macro and will be used as a macro by the compiler when it is
  called."
   :arglists '([name doc-string? attr-map? [params*] body]
               [name doc-string? attr-map? ([params*] body)+ attr-map?])
   :added "1.0"}
 defmacro (fn [&form &env
               name & args]
            (let [prefix (loop [p (list name) args args]
                           (let [f (first args)]
                             (if (string? f)
                               (recur (cons f p) (next args))
                               (if (map? f)
                                 (recur (cons f p) (next args))
                                 p))))
                  fdecl (loop [fd args]
                          (if (string? (first fd))
                            (recur (next fd))
                            (if (map? (first fd))
                              (recur (next fd))
                              fd)))
                  fdecl (if (vector? (first fdecl))
                          (list fdecl)
                          fdecl)
                  add-implicit-args (fn [fd]
                            (let [args (first fd)]
                              (cons (vec (cons '&form (cons '&env args))) (next fd))))
                  add-args (fn [acc ds]
                             (if (nil? ds)
                               acc
                               (let [d (first ds)]
                                 (if (map? d)
                                   (conj acc d)
                                   (recur (conj acc (add-implicit-args d)) (next ds))))))
                  fdecl (seq (add-args [] fdecl))
                  decl (loop [p prefix d fdecl]
                         (if p
                           (recur (next p) (cons (first p) d))
                           d))]
              (list 'do
                    (cons `defn decl)
                    (list '. (list 'var name) '(setMacro))
                    (list 'var name)))))
```

`defmacro` **expansion**

```
(defmacro dyslexially
  [expr]
  (reverse expr))
```

```
(do
  (clojure.core/defn dyslexially
    ([&form &env expr]
       (reverse expr)))
 (. #'dyslexially (setMacro))
 #'dyslexially)
```

Macros that write macros

```
1:  ;; from core.logic
2:
3:  (defmacro RelHelper [arity]
4:    (let [r (range 1 (+ arity 2))
5:          fs (map f-sym r)
6:          mfs (map #(with-meta % {:volatile-mutable true :tag clojure.lang.IFn})
7:                   fs)
8:          create-sig (fn [n]
9:                       (let [args (map a-sym (range 1 (clojure.core/inc n)))]
10:                        `(invoke [~'_ ~@args]
11:                                 (~(f-sym n) ~@args))))
12:         set-case (fn [[f arity]]
13:                    `(~arity (set! ~f ~'f)))]
14:     `(do
15:        (deftype ~'Rel [~'name ~'indexes ~'meta
16:                        ~@mfs]
17:          clojure.lang.IObj
18:          (~'withMeta [~'_ ~'meta]
19:            (~'Rel. ~'name ~'indexes ~'meta ~@fs))
20:          (~'meta [~'_]
21:            ~'meta)
22:          clojure.lang.IFn
23:          ~@(map create-sig r)
24:          (~'applyTo [~'this ~'arglist]
25:             (~'apply-to-helper ~'this ~'arglist))
26:          ~'IRel
27:          (~'setfn [~'_ ~'arity ~'f]
28:            (case ~'arity
29:                  ~@(mapcat set-case (map vector fs r))))
30:          (~'indexes-for [~'_ ~'arity]
31:            ((deref ~'indexes) ~'arity))
32:          (~'add-indexes [~'_ ~'arity ~'index]
33:            (swap! ~'indexes assoc ~'arity ~'index)))
34:        (defmacro ~'defrel
35:          "Define a relation for adding facts. Takes a name and some fields.
36:          Use fact/facts to add facts and invoke the relation to query it."
37:          [~'name ~'& ~'rest]
38:          (defrel-helper ~'name ~arity ~'rest)))))
```

))))))))))

Blatant Omissions

- Magical arguments: `&form` and `&env`
- clojure.tools.macro
  - local macros (without deffing anything)
  - symbol macros

What was that you said

- By taking advantage of homoiconicity, macros give you a relatively easy way to reduce syntactic repetition and ceremony, effectively adding new features to the language
- They require a decent understanding of how compilation works
- They make your code awkward and hard to reason about if used unnecessarily
- Syntax-quote makes it easier to write safe macros

So Thanks

Also thanks to

- Andrew Brehaut (`@brehaut`)
- Daniel Glauser (`@danielglauser`)
- Lucas Willett (`@ltw_`).

Groupon is hiring for Clojure