Clojure Macros

Drew Colthorp

July 9, 2013

Contents

Chapter 1

Homoiconicity

Clojure is a lisp, one of a family of languages known for their prefix syntax and copious parentheses. Those parentheses are there for good reason. Lisps are *homoiconic*, meaning code written in the language is encoded as data structures that the language has tools to manipulate.

Consider the following expression:

```
(let [x 1]
    (inc x))
```

This code, entered directly into the REPL, returns `2` because the repl compiles and executes any code entered into it. But `[x 1]` is also a literal vector data structure when it appears in a different context.

All Clojure code can be interpreted as data. In fact, Clojure is a superset of EDN - Extensible Data Notation, a data transfer format similar to JSON. EDN supports numbers, strings, lists `(1 2 3)`, vectors `[1 2 3]`, maps {`"key" "value"`}, and more. If this sounds and looks a lot like Clojure syntax, it's because it is. The relationship between Clojure and EDN is similar to that of Javascript and JSON, but much more powerful.

In Clojure, unlike JavaScript, all code is written in this data format. We can look at our `let` statement not as Clojure code, but an EDN data structure. Let's take a closer look:

```
(let [x 1]
    (inc x))
```

In this data structure, there are four different types of data.

- `1` is a literal integer.

- `let`, `x`, and `inc` are *symbols*. A symbol is an object representing a name - think a string, but as an atomic object and not a sequence of characters.

- `[x 1]` is a *vector* containing two elements: symbol, `x`, and an integer, `1`. Square brackets always signify vectors when talking about EDN data structures.

- `(inc x)` is a *list* (a linked list data structure) containing two symbols, `inc` and `x`.

When thinking about a piece of Clojure code as a data structure, we say we are talking about the **form**. Clojure programmers don't normally talk about EDN, there are just two ways to think about any bit of Clojure: 1) as code that will execute or 2) as a *form*, a data structure composed of numbers, symbols, keywords, strings, vectors, lists, maps, etc.

Symbols are particularly important. They are *first class names*. In Clojure, we distinguish between a variable and the name of that variable. When our code is executing, `x` refers to the variable established by our `let` binding. But when we deal with that code as a form, `x` is just a piece of data, it's a name, which in Clojure is called a **symbol**

This is why Clojure is homoiconic. Code forms are data structures and data structures can be thought of as forms and executed as code. This transformation is quite literal, and two core operations, `quote` and `eval` are key ingredients to this potion.

1.1 Quote

Exercise: Find the value of each of the following in the repl:

```
+
(quote +)
(+ 1 2)
(quote (+ 1 2))
(= (quote +)
   (first (quote (+ 1 2))))
(= (+ 1 2)
   (quote (+ 1 2)))
(= (quote (+ 1 2))
   (list (quote +) 1 2))
```

```
(= ["string" :keyword #{:set} {:map true}]
     (quote ["string" :keyword #{:set} {:map true}]))
(quote (quote +))
```

`quote` is a special form, a magic keyword built in to the compiler, which suspends computation. `quote` means "give me the form of this thing, not its value". Function calls are instead treated as list constructors, and any names are returned as symbols instead of resolved to runtime values.

As you've seen, `quote` leaves some things alone and changes other things. Things that don't *do* stuff in clojure code are left alone (strings, keywords, etc.). But the stuff that does, like symbols (which refer to variables) and parentheses (which invoke functions), are left as data.

`quote` normally isn't written out in Clojure. An apostrophe is used instead. An apostrophe is short-hand for quote. Thus, the following are all true:

```
(= (quote +) '+)
(= (quote (+ 1 2)) '(+ 1 2))
(= ''+
   '(quote +)
   (quote '+)
   (quote (quote +)))
```

1.2 Eval

Exercise: Find the value of each of the following in the repl:

```
+
'+
(eval '+)

'(+ 1 2)
(eval '(+ 1 2))

''(+ 1 2)
(eval ''(+ 1 2))
(eval (eval ''(+ 1 2)))
```

`eval` is the opposite of `quote`[1]. `quote` is a special form that stops the execution of code and instead treats it like a data structure. `eval` takes a data structure and executes it as code.

```
> (quote (+ 1 1))
(+ 1 1)
```

With `eval`, we can undo our quotation:

```
> (eval
    (quote
      (+ 1 1)))
2
```

Using `eval` to immediately undo quotation is not very useful. Where it starts to get interesting is in the space between the `eval` and the `quote`. It's here that we can redefine the rules of the language. By treating our quoted code as data and manipulating it, we can translate it into a different data structure. Something that was previously meaningless to the Clojure compiler can become executable code. We can introduce new language constructs or play with scoping or even change the foundations of the language.

1.3 Exercises

Let's build a fun-house alternate arithmetic universe where up is down and left is right. Addition becomes subtraction and multiplication is swapped with division.

1.3.1 `funhouse`

Define the function `funhouse`, so that the following work:

```
(funhouse '(+ 3 1)) ;=> (- 3 1)
(funhouse '(- 3 1)) ;=> (+ 3 1)
(funhouse '(* 3 1)) ;=> (/ 3 1)
(funhouse '(/ 3 1)) ;=> (* 3 1)
```

[1] Not exactly. Eval doesn't take lexical bindings into account. If `x` is `def`ed, `(eval 'x)` returns its value, but if not `(let [x 1] (eval 'x))` will fail because `x` is not defined.

`funhouse` should take a form and return a form. It **should not** `eval` its results.

It should operate on the *symbols* `+,-,*,/`, not the *functions*. If your outputs contain `#<` in them, you have symbols that need to be quoted.

Once all that is sorted, you should be able to `eval` your function's return value and get the right result.

Useful functions: `first`, `rest`, `cons`, `quote`.

1.3.2 Recursive `funhouse`

Now, let's make `funhouse` work on more sophisticated examples. Make it work on trees of expressions.

Write a recursive version of `funhouse`. It should work on flat function calls, as in the previous exercise, but also arguments:

```
> (funhouse '(/ 3 (+ 2 1)))
(* 3 (- 2 1))
```

Start with this implementation skeleton:

```
(defn funhouse [expr]
  (if (list? expr)
    ...
    ...))
```

Useful functions: `map`

1.3.3 Extra Credit

Extra Credit: Expand `funhouse` to properly recurse into maps, vectors, and sets:

```
> (funhouse '(+ (/ 2 2)
                (first #{(- 1 1)})
                (first [(- 1 1)])
                (:2nd {:1st 1, :2nd (+ 5 1)})))
(- (* 2 2)
   (first #{(+ 1 1)})
   (first [(+ 1 1)])
   (:2nd {:1st 1, :2nd (- 5 1)}))
```

Chapter 2

Macros

We've seen with `funhouse` that transformations on code before it's evaluated can have a huge, arbitrary impact on what that code does. Bringing this power to developers is exactly what macros are designed to do.

Macros are functions with a twist: instead of taking the values of evaluated arguments, macros run at compile time and operate on the forms themselves. For a simple, initial intuition, it's not too wide of the mark to think of a macro as a function whose arguments are automatically quoted for you, and whose result is then `eval`ed.

This explanation doesn't really convey the power of macros. Macros are not just a feature that lets you do nifty tricks. Macros allow you to extend the compiler and redefine Clojure itself. And they do that job well. So well in fact, that core, seemingly irreducible Clojure features are macros themselves.

Consider `let`. It binds variables. You might think `let` is built into the language and irreducible, but it's not. Even `let`, the thing you use to bind variables in just about every piece of Clojure in existence is a macro. Let's take a look.

A trivial example of using `let` might be as follows:

```
(let [x (first my-vector)]
  x)
```

This use of `let` binds `x` to the first element of `my-list` and returns it. But `let` also supports *destructuring*. We can use special syntax to grab the first element for us:

```
(let [[x] my-vector]
  x)
```

This does the same thing. The brackets around `x` tells Clojure that the thing on the right is a sequence and that we want `x` to be the first entry.

Here's the thing: this destructuring feature of `let` is not built into the compiler, but a macro in `clojure.core` that adds the feature to a simpler binding form.

We can see what a macro is equivalent to by using the function `macroexpand-1`. `macroexpand-1` takes a quoted form and, if it is an invocation of a macro, shows you what that macro returns before the result is evaluated. Here's how it works with our destructuring `let`:

```
(macroexpand-1 '(let [[x] my-vector]
                  x))
```

The result, after being line-broken and indented, is

```
(let* [vec__1442 my-vector
       x (clojure.core/nth vec__1442 0 nil)]
  x)
```

`let*` is the simple, fundamental way to bind variables that `let` turns into. `let*` is built right into the compiler, but it doesn't support destructuring. `let` uses `let*` to first rename `my-vector` to a generated name (`vec__1442`), then uses the `nth` function to get the first element, labeling that value `x` as we requested.

Not only is `let` a macro, but so is `fn`, `defn`, `for`, `doseq`, `dotimes`, and `ns`. These are fundamental building-blocks of Clojure the language, and they're defined with the exact same feature we're going to be using ourself. With this power, Clojure is not so much the language in which you write your program, it's the starting point from which you grow the language into the best language for the program you're writing.

2.1 Clojure Compilation and Execution Phases

We introduced macros as functions whose arguments are quoted and whose result is evaled. This is roughly what happens, but not in the way that description implies. This is true because of the *time* at which macros are executed: during compilation, not runtime.

This becomes clear when you understand the stages Clojure code goes through.

2.1.1 Read

Clojure code starts out not as data structures, but as text. Clojure is a homoiconic language, but we encode Clojure data structures with string representations. The first phase our program passes through is the **read** phase, whose job it is to get it out of chars and into symbols and sequences.

The Clojure reader transforms strings of Clojure code or data in the language's notation (a superset of EDN) into Clojure data structures. The reader is available to Clojure programmers as the `read` and `read-string` functions.

```
> (read-string "(foo 1 :bar [#\"regexp\"])")
(foo 1 :bar [#"regexp"])
```

Think of the compiler as reading through your files, form-by-form, and executing them one-by-one.

2.1.2 Macroexpansion

After a form is read by the compiler, it goes through *macro-expansion*. Whenever the compiler sees a list whose first element is the name of a macro, it passes the subsequent forms to the macro, using the return value of the macro instead of the macro invocation. Macro-expansion continues recursively until no forms in the tree refer to macros. This happens at every level of the tree from the top down.

So macros are executed on forms not because they're literally quoted by Clojure for you, but because your macro runs while they're still simple data structures and before anything has been done to them.

Another implication of this phase is that the code returned by your macro is run *as is* in the middle of someone else's code, where they called it. Your macro will be called within another function, injecting code in an unknown future context. If you define a variable in a `let` block in your macro that is the same name as one the user defined, you will redefine it - possibly without their knowledge. (We'll see tools for avoiding this soon.)

2.1.3 Analysis/Emission

The macro-expanded form is analyzed by the compiler and "native" instructions (e.g. JVM bytecode) are emitted.

2.1.4 Evaluation

The form is evaluated by the compiler. Much like the `quote`-ing in our initial intuition, the `eval` is also not literal: it's `eval`ed because your macro runs before the entire program is run. It's output executes when the rest of the program does.

When a top-level form is evaluated, it's effect is available to subsequent macros as well as functions. This means you can define helper functions and macros that you then use in more sophisticated macros later in the file. Your macros run at compile time, but they can call functions. When you define a function and use it from a macro, the function is executed at compile time when the macro is expanded.

2.2 Writing macros

Let's write our first macro, `debug`. Occasionally, it's helpful to be able to see what expressions evaluate to in the middle of a more complicated expression. Let's build a simple `debug` macro we can wrap around an expression to see what it evaluates to at run-time.

When writing macros, we first start with what we want the macro to do. `debug` should have the following behavior:

```
> (debug (+ 1 2))
(+ 1 2) :> 3
3
```

That is, when debug is wrapped around an expression, we want to print the expression, a keyword that looks like an arrow (:>), and its value, then return the value.

The next step in writing a macro is to figure out what code we want the macro to return. If `(+ 1 2)` is our argument, we want the following code to be returned:

```
(do (println '(+ 1 2) :> (+ 1 2))
    (+ 1 2))
```

Every call to a macro form expands to another form. In order to have a side effect (`println`) and return the value we want, we use `do` to execute in sequence and return the final value.

Looking at the form of this code, have a list containing the symbol `do`, another list, and our expression. Notice that our argument appears three times in the expanded code.

Now we can write the code to build this structure as the body of our macro:

```
(defmacro debug [expr]
  (list 'do
    (list 'println (list 'quote expr) :> expr)
        expr))
```

`defmacro` tells Clojure this is a macro and not a regular function, but other than that the body of the macro is plain old Clojure operating on a data structure argument. We use the `list` function and `quote`d symbols to assemble the desired output. We quote our symbols because we want to return the `form` of the new code. We *do not* want to return the `println` function itself.

Let's verify that our macro works with `macroexpand-1`:

```
> (macroexpand-1 '(debug (+ 1 2)))
(do (println '(+ 1 2) :> (+ 1 2))
    (+ 1 2))
```

That is, in fact, what we wanted to return. `macroexpand-1` allowed us to *execute our macro as though it were a function*, getting the result of the macro function instead of executing the code. Clojure automatically displayed `(quote (+ 1 2))` into `'(+ 1 2)`, since they're equivalent and the apostrophe is easier to read.

Here's what happens when we use it:

```
> (debug (+ 1 2))
(+ 1 2) :> 3
3
```

Clojure sees the invocation of our `debug` macro, executes it against the quoted form of the argument instead of the runtime value, then evaluates the code that `debug` returns, giving us the behavior we wanted to see.

2.3 Exercises

2.3.1 `in-funhouse`

`funhouse` is too inconvenient to use. It would be nice to have a macro `in-funhouse` that can be used without quoting. Create an `in-funhouse` macro that performs the `funhouse` transformation.

When you use it, it should work like this:

```
> (in-funhouse (+ 6 1))
5
```

When you `macroexpand-1` it, it should work like this:

```
> (macroexpand-1 '(in-funhouse (+ 6 1)))
(- 6 1)
```

Start with this definition:

```
(defmacro in-funhouse [expr]
...)
```

Hints: Reuse `funhouse`. **Do not** use `eval` in the implementation of this or any other macro.

2.3.2 `do-funhouse`

Let's take this a step further. Write a `do-funhouse` macro that can take multiple expressions. Here's an example of how we'd like to be able to use it:

```
> (do-funhouse (println (+ 1 2))
               (* 2 2))
-1
1
```

Here's an example of how it should `macroexpand-1`:

```
> (macroexpand-1
    '(do-funhouse (println (+ 1 2))
                  (* 2 2))))
(do (println (- 1 2)) (/ 2 2))
```

To take arbitrarily many arguments, `do-funhouse` will need to be a variadic macro (taking arbitrarily many arguments). Here's a start of the definition:

```
(defmacro do-funhouse [& exprs]
  ...)
```

Functions you will need: `map`.

Chapter 3

Syntax-quote

With macros being such a common thing in Clojure programming, the language has built-in syntax to make constructing code in macros more convenient. This syntax, called syntax-quotation, is like the built-in syntax for `quote`-ing, but targeted at macro-building.

Syntax-quote is a way to build data structures up by interpolating values into sequences. The result is quoted, except where syntax is used to specify values to be interpolated.

Within a syntax-quotation, an expression prefixed with ~ embeds the value of the expression in the result. Prefixing an expression returning a sequence with ~`@` splices the entire sequence in place:

```
> (let [x 1 v [2 3] more [4 5]]
    `(~x ~v ~@more 6))
(1 [2 3] 4 5 6)
```

As you can see, ~ (unquote) just interpolates values as is. ~`@` (unquote-splice) works on sequences and interpolates every value within in that spot in the containing form.

Also, we did not need to use the `list` function to construct our lists - since syntax-quote quotes anything that is not unquoted with ~ or ~`@`, we can just use parentheses to denote lists within a syntax-quote.

3.1 Qualified symbols

Syntax-quote is not a general purpose interpolation utility, however. One way to see this is the effect it has on symbols:

```
> `(loop [a 1]
     (recur (inc a)))
(clojure.core/loop [user/a 1]
  (recur (clojure.core/inc user/a)))
```

Any symbol within the quoted form is fully qualified to the namespace that defined the symbol at the point the syntax-quote was created. Since `clojure.core` was imported at the point this expression was executed, all references to values from that namespace are qualified.

When writing macros, we're writing code that will be compiled in code that exists in another namespace. Users of our macros will want them to work consistently, regardless of which functions they've defined locally. Fully-qualified symbols fix the intention of the code generated with syntax-quote so that the meaning of the macro doesn't change for its users.

Consider our `debug` macro from earlier.

```
(defmacro debug [expr]
  (list 'do
    (list 'println (list 'quote expr) :> expr)
        expr))
```

It is not using syntax-quote, so the symbols are *not* fully qualified. As a result, `debug` has a potential bug. It uses the meaning of `println` in the place it is used, not the `println` we intended to be invoked. If we were to use `debug` in an environment where `println` meant something else, our macro wouldn't do what users expect.

For example, if we try to debug an expression where we've defined a local `println` variable,

```
> (let [println 32] (debug (+ 1 2 3)))
ClassCastException java.lang.Long cannot be
  cast to clojure.lang.IFn...
```

We get an error. `debug` is expanded in place within the `let` statement, and it's using an unqualified `println` symbol, so it gets the local value of `println` instead of the one we intended. (This is unlikely to be an issue in the case of `println`, but there are many other cases where this would be a serious issue.) If we had used syntax-quote to define `debug` instead, this never would have happened.

3.2 Variable capture

There's another important reason symbols are fully qualified in syntax-quoted forms. They can't be bound as variables.

Since macros allow us to change the code that's being compiled, macros can and often do introduce local variables to do their job. In most cases however, we don't want to introduce variables that will clash with the code that's using the macro (the mirror of the problem `debug` had). While there are cases where we wish to do this explicitly, such cases are the exception and not the rule.

Consider the questionably useful macro

```
(defmacro twice [& body]
    (concat '(dotimes [i 2]) body))
```

`twice` is a new looping construct that executes its body exactly twice. Note that we did not use syntax-quoting here, but just quoted our symbols like we had with `debug`.

When we `macroexpand-1` twice, we get:

```
> (macroexpand-1 '(twice (println "hello")))
(dotimes [i 2] (println "hello"))
```

As currently defined, twice expands to code which *captures* the variable `i`. Any reference to `i` inside the body of the `twice` form will refer to the `i` defined by the macro instead of the existing variable. Thus

```
(doseq [i (range 3)]
    (twice (pr i)))
```

is equivalent to

```
(doseq [i (range 3)]
  (dotimes [i 2]
    (pr i)))
```

which prints "010101", the values of `i` from `dotimes`, instead of the expected "001122", the `i` variable from `doseq` the code was evidently referring to.

If we had used syntax-quote in the first place, this problem *never would have happened*. Let's take a look:

```
> (defmacro twice [& body]
    `(dotimes [i 2] ~@body))
> (twice (println "hello"))
CompilerException java.lang.RuntimeException: Can't let
    qualified name: user/i...
```

Instead of capturing the variable `i`, we get a compilation error when we go to use the macro. The compiler complains that we "can't let the qualified name: user/i". If we examine the expanded code, we see that

```
> (macroexpand-1 '(twice (prn i)))
(clojure.core/dotimes [user/i 2] (prn i))
```

Yes, we are indeed trying to let a qualified name. Fully qualified symbols can't be used as variable names or `fn` parameters, so you need to go out of your way to capture variables when using backquote, so this problem doesn't occur.

3.3 Auto-gensyms

So how do we fix the problem? We simply suffix the variables we want to introduce with a `#`. Any symbols with that suffix get transformed into unique symbols that will not clash with any other variable.

```
> (defmacro twice [& body]
    `(dotimes [i# 2] ~@body))
> (twice (println "hello"))
hello
hello
nil
```

By simply appending a `#` to the variable name, the problem goes away. What did that `#` do?

```
> `i#
i__1647__auto__
```

Within a backquote, any symbol ending in a `#` is an **auto-gensym**. A "gensym" is a *generated symbol*. They're used in lisps for exactly this reason. There are two important things to note about auto-gensyms:

1. They're not namespace-qualified, so they can be used in `let`.
2. They're generated so as not to clash with any other symbol, and thus guarantee variable capture will not occur.

Furthermore, all references to the same `#`-suffixed symbol within the same syntax-quote resolve to the same unique symbol

```
> `(x# x#)
(x__1440__auto__ x__1440__auto__)
```

This ensures that any variable we introduce in a syntax-quoted form will be used anywhere we reference it:

```
> (eval `(let [x# 1] (inc x#)))
2
```

It's important to keep in mind that auto-gensyms vary from syntax-quote to syntax-quote.

```
> [`x# `x#]
[x__1650__auto__ x__1651__auto__]
```

Thus, if you need to use a gensym in multiple backquotes within the same macro (which is fairly common), you need to use the `gensym` function to create one you can share.

```
> (let [sym (gensym 'my-gensym)]
   `[~sym ~sym])
[my-gensym1656 my-gensym1656]
```

We can simply create our own `gensym` and interpolate it anywhere we want. Since it's a value stored in a variable, it will not change from syntax-quote to syntax-quote.

3.4 How to use syntax quote

Using syntax quote is pretty straightforward. First, consider the code you want to generate, let's say in this case it is:

```
(let [x (+ 1 2)]
   (println x)
   ...)
```

First, put a backtick in front of the outer-most form

```
`(let [x (+ 1 2)]
   (println x)
   ...)
```

Next, add `#` at the end of any variable you're introducing, and all of it's uses.

```
`(let [x# (+ 1 2)]
   (println x#)
   ...)
```

Finally, replace any code you're creating or is passed into your macro with an unquoted reference

```
`(let [x# ~expr]
   (println x#)
   ~@body)
```

Note that we **let syntax-quote do the work** of creating the constant parts of our forms for us. With syntax-quote, you don't usually need to use the `list` function to create forms yourself.

3.5 Exercise

Define `debug` using syntax-quotation to fix the variable capture problem. You will need to use syntax-quote (backtick) and ~ (unquote).

Your new `debug` implementation should work as follows:

```
> (macroexpand '(debug (+ 1 2)))
(do (println '(+ 1 2) :> (+ 1 2))
  (+ 1 2))
```

Keep in mind that

```
(list 'foo bar)
```

is roughly equivalent to the syntax-quoted

```
`(foo ~bar)
```

You should not need to work "list" anywhere in your solution.

Chapter 4

Macro trade-offs

The way macros allow you to hook in and extend the Clojure compiler provides a lot of power. Macros give you the ability to define new control constructs, new languages, or redefine Clojure itself. But, like all powerful things, there are trade-offs involved.

- Macros are not values. They can't be passed to `comp` or `map` or any other higher order function. At best, you need to call a macro in a lambda and pass that to a higher-order function.

- Macros aren't present in stack traces, making error diagnosis harder.

- Macros are harder to debug. Code that writes code isn't nearly as straightforward as the code itself.

These trade-offs lead to the golden rule of macro programming: don't use them if a function will do. Macros are the ultimate tool for shooting yourself in the foot if you're trying to be clever. Think about what's built-in to Clojure or well-known Clojure libraries. Does your macro seem in the spirit of something you've seen before? Was that a macro or a function? Think about whether what your building will fit in with the rest of the language if it's a macro. Even if your intended behavior couldn't possibly be a function, it may very well be the case that it would be better if it worked in a way that could be a function.

So once you've convinced yourself that a macro really is worth it, what are macros good for?

4.1 Macros can transform code

Consider the threading macro `->`, which takes a value and a series of forms, and threads the value through the forms. For example:

```
> (macroexpand-all '(-> 1 inc (* 2)))
(* (inc 1) 2)
```

`1` gets passed to `inc`, and the result of that call is multiplied by 2. (`macroexpand-all` is in the `clojure.walk` namespace.You'll need to import it in order to use it.)

This transformation couldn't possibly be done with a function. Sure, you could write a higher-order function which takes a value and a bunch of functions of one argument and threads the value through those, but you'd need to define a lambda for the `(* 2)` in the example.

This is a transformation on the *text* of the program, the realm of macros.

4.2 Macros can bind variables

With macros, you can introduce new variables into the lexical environment. You don't generally want to do this without your users' permission, but can be important in certain cases.

4.3 Macros can control execution

Arguments can be executed as many times as you wish. This, combined with the ability to bind variables, allows you to define arbitrary control flow constructs.

4.4 High performance abstraction

Since macros are executed before your program actually executes, you can define very high level abstractions that compile down to extremely efficient code. `core.match`, for example, introduces pattern matching to Clojure (think destructuring, but extensible and used as a test in control flow). `core.match` allows you to write higher level code but with zero overhead - your code is compiled down to if statements that are likely *more efficient* than what you would have written yourself.

4.5 Building up the language

As a result of all of the above, macros ultimately provide a way to build Clojure into the language you want to use to write your program. Macros provide the link missing in most languages, between a way of creating abstractions, and a way of defining language.

In *The Joy of Clojure*, Michael Fogus and Chris Houser state "Clojure programmers dont write their apps in Clojure. They write the language that they use to write their apps in Clojure." Macros, together with lambdas, are the defining feature that make this happen.

Chapter 5

Multiple execution

One challenge when writing macros is controlling execution. Consider the syntax-quoted version of the `debug` macro from earlier:

```
(defmacro debug [expr]
  `(do
    (println (quote ~expr) :> ~expr)
        ~expr))
```

`debug` does what we wanted it to, print the form and the result while returning the result, but if you ever use it with a side effect, you'll notice an issue.

Try this:

```
> (debug (do (println "executing!") 1))
executing!
(do (println executing!) 1) :> 1
executing!
1
```

Our argument is executed not once, but twice.

This is a common challenge when writing macros. Some arguments need to be executed zero times, others multiple, but often you want something to be executed exactly once.

The solution is to `let` the argument to a new variable in the generated code, then only use that variable.

5.1 Exercises

5.1.1 Fix `debug`

Our debug macro from earlier has another bug. Though we've fixed the variable capture problem with syntax-quote, we have another issue: our expressions are evaluated twice.

Consider this example:

```
> (debug (do (println "expression evaluated!") 1))
expression evaluated!
(do (println expression evaluated!) 1) :> 1
expression evaluated!
1
```

Fix it. Change the implementation of `debug` so that its argument is evaluated only once.

5.1.2 `and-1`

One core use of macros is control flow constructs. `and`, `or`, `when`, `cond`, `condp` are all macros. They have to be, because their arguments are not always executed. Deciding if and when execution of arguments happens is the domain of macros.

Let's build up a 2-argument equivalent to `and` using macros. Recall that `and` short-circuits execution. For example

```
> (and nil (println "I will not print"))
nil
```

If the first argument is logically false, it is returned and the second argument is not executed. However, if the first argument is true, the second is executed and its value is returned.

```
> (and true :foo)
:foo
```

Write the macro `and-1` which takes exactly two arguments and is equivalent to `and`.

Also note that and should evaluate arguments at most once, as shown in the following example:

```
> (and-1 (do (println "I will print")
             true)
         1)
I will print
1
```

Chapter 6

Using functions in macros

Sophisticated macros are usually not written as complicated `defmacro`s, but instead call into helper functions defined only for the purpose of implementing the macro. In this section, we'll explore this technique while exploring another use of macros, optimization through compile-time computation.

Consider the following use of the hiccup HTML templating library:

```
> (let [myvar 1]
    (html [:ul
      [:li "li1"]
      [:li {:class "second"} myvar]]))

"<ul><li>li1</li><li class=\"second\">1</li></ul>"
```

The `html` macro takes a tree of vectors and constructs HTML corresponding to their structure. This work could be entirely performed at runtime, but `html` is smarter than that. There are a number of static elements in this example:the `:ul`, the first `:li`, and the tag-name and class of the second `:li`, though its content is a run-time value.

Hiccup takes advantage of known elements and precompiles static elements for performance.

```
> (macroexpand-1
   '(html [:ul
      [:li "li1"]
      [:li {:class "second"} myvar]]))
(clojure.core/str
   "<ul"
```

```
  ""
  ">"
  "<li>li1</li>"
  "<li"
  " class=\"second\""
  ">"
  (#'hiccup.compiler/render-html myvar)
  "</li>"
  "</ul>")
```

All of the static elements are precompiled into strings, and only the dynamic reference to `myvar` remains, which is converted at runtime using the internal `render-html` function.

`hiccup`'s macro does not itself walk trees of vectors to generate html. The macro itself is almost trivial, but it uses a helper function that walks the data structure and creates the appropriate strings.

This approach is critical for tackling non-trivial macros.

The key thing to remember is that macros are just functions that are called at macroexpansion time instead of run time. They can include arbitrary Clojure code, such as function calls, and not just syntax quotations. And these functions can be your own.

Whenever a macro starts to get challenging, pull some or even all of the logic out into a function you can test and perfect independently, then use that function from your macro.

6.1 Exercise

Practice compile-time optimization with a simplified example of the type of work hiccup is doing. Let's create a macro `pre-str` which looks for static elements and converts them to a string at compile time. (This macro is unnecessary, but good practice.)

`pre-str` should work just like `str` in terms of runtime behavior:

```
> (pre-str 1 " " :foo " " (+ 1 2) " " *file*)
"1 :foo 3 NO_SOURCE_PATH"
```

However, `pre-str` converts static values into strings:

```
> (macroexpand-1 '(pre-str 1 " " :foo " " (+ 1 2) " " *file*))
(clojure.core/str "1" " " ":foo" " " (+ 1 2) " " *file*)
```

`pre-str` should detect of static values and convert them into strings. Run-time values should still be executed.

6.1.1 `stringify-static`

Define the function `stringify-static` which takes a sequence of values and returns a sequence with all static values converted to strings.

```
> (stringify-static '[:foo 1 (+ 1 2) " " bar])
("foo" "1" (+ 1 2) " " bar)
```

Helpful Clojure: `keyword?`, `string?`, `number?`, `map`, `or`, `some-fn`.

6.1.2 `pre-str`

Create the macro `pre-str` which works as described above. Call `stringify-static` from it in order to accomplish part of the work.

6.1.3 Extra Credit

Modify `stringify-static` to convert runs of static values into a single string. The updated `pre-str` implementation should expand as follows:

```
> (macroexpand-1 '(pre-str 1 " " :foo " " (+ 1 2) " " *file*))
(clojure.core/str "1 :foo " (+ 1 2) " " *file*)
```

Helpful Clojure: `apply`, `partition-by`, and possibly `mapcat`.

Chapter 7

Recursive macros

Macros, like functions, can be recursive. Recursion in macros is just as useful as with regular functions. They're written similarly as well - you need a non-recursive base case so that your macro doesn't expand forever.

Where recursive macros differ from recursive functions is that recursive macros don't recurse directly, but expand to code that contains a call to the macro. Recall that macro-expansion continues until no macros remain in the output-recursive macros exploit this by returning calls to themselves that the macro-expander will then expand for them.

Recall that -> threads the results of one form through the first argument of subsequent forms, so

```
(-> 1 (+ 2) inc (* 3))
```

is transformed into

```
(* (inc (+ 1 2)) 3)
```

-> is basically defined as follows in clojure.core:

```
(defmacro ->
  ([x] x)
  ([x form] (if (seq? form)
              `(~(first form) ~x ~@(next form))
              (list form x)))
  ([x form & more] `(-> (-> ~x ~form) ~@more)))
```

A few things to notice:

1. This macro uses the built-in syntax to define different implementations for differing numbers of arguments. This works just as in `defn`. This can be useful for simple recursive macros.

2. Note that recursive calls to `->` don't happen directly in the code of `->`. They're in syntax-quoted forms. Thus, `->` returns forms which call itself. The macroexpander deals with expanding the recursive calls.

`macroexpand-1` doesn't usually do what you want when testing recursive macros - it only expands once at the top level of the form. To expand recursive macros, use `macroexpand-all` from the `clojure.walk` namespace.

```
> (use 'clojure.walk)
> (macroexpand-all '(-> 1 (+ 2) inc (* 3)))
(clojure.core/* (clojure.core/inc (clojure.core/+ 1 2)) 3)
```

7.1 Exercise

Define `and-*`, which works just like `and`. This should be a recursive macro which can take arbitrarily many arguments instead of only two. Use `->` above as a guide for how to structure this macro.

Chapter 8

Code walking

Sophisticated macros usually do more than rearrange their arguments. They often need to impact the behavior of subexpressions nested arbitrarily deep within their arguments.

`funhouse` is an example of this. Taken to its logical conclusion, `funhouse` would ideally transform `+`, `-`, `/`, and `+` at any point in any data structure, besides strings or part of a symbol or keyword.

This is common enough that there are tools built in to Clojure to help with this. This section explores some of the options.

8.1 clojure.walk

The `clojure.walk` namespace includes functions designed to correctly walk Clojure forms. It handles maps, vectors, lists, and sets perfectly.

Two functions, `postwalk` and `prewalk`, allow you to apply a function to every node in a tree of code. `prewalk` visits parent nodes before children, and `postwalk` visits children first.

To see an example of how these work, you can use `prewalk-demo` and `postwalk-demo` to see how a structure will be walked by these functions. For example:

```
> (walk/postwalk-demo '(foo {:bar [:baz :qux]}))
Walked: foo
Walked: :bar
Walked: :baz
Walked: :qux
Walked: [:baz :qux]
```

```
Walked: [:bar [:baz :qux]]
Walked: {:bar [:baz :qux]}
Walked: (foo {:bar [:baz :qux]})
```

This allows you to transform all leaves in a form, for example:

```
> (postwalk #(if (coll? %) % (name %))
           '(foo {:bar [:baz :qux]}))
("foo" {"bar" ["baz" "qux"]})
```

8.1.1 Exercise

Use `postwalk` or `postwalk-replace` to implement `in-funhouse` from scratch.

Recall that `in-funhouse` swaps the meaning of `+` with `-` and `*` with `/` within it.

```
> (in-funhouse (+ 10 5))
5
> (in-funhouse (- 10 5))
15
> (in-funhouse (/ 3 10))
30
> (in-funhouse (* 10 2))
5
```

8.2 Composability

Code-walking macros such as `in-funhouse` can cause issues with composability. What if you have another macro, `adventure-town` that makes different choices about how to remap arithmetic?

```
(defmacro adventure-town [expr]
  (postwalk-replace {'+ `* '* `+ '- `/ '/ `-}
                    expr))
```

(Note the syntax-quote before the new values for the symbols - this avoids a variable capture issue present in our earlier explorations in `funhouse` and are critical for this example.)

If `in-funhouse` maps `+` to `-`, and `adventure-town` maps `+` to `*`, what happens if you compose `in-funhouse` and `adventure-town`?

```
> (in-funhouse (adventure-town (+ 10 10))
1
```

Since `in-funhouse` runs before `adventure-town`, and indiscriminately rewrites symbols, it doesn't respect the alternate universe `adventure-town` is trying to achieve and stomps all over it. In this case, `(+ 10 10)` would be rewritten as `(- 10 10)` *before* `adventure-town` has had a chance to run. In this case, the result would be the result of `(- 10 10)` in adventure town, 1, and not `(+ 10 10)`, which we expect to be 100.

The solution is to use `let` or `let`-like constructs (described in following sections) to establish new meanings. `let` creates a new lexical context, and re-`let`ing a variable overrides the previous meaning.

8.2.1 Exercise

Create a macro `composable-funhouse` which will respect redefinition of arithmetic operations. Instead of rewriting symbols, use `let` to establish new local bindings for the symbols.

In order to do this, you need to explicitly capture the local variable - auto-gensyms can't be used here, since the code passed to `composable-funhouse` will expect to use the usual arithmetic symbols. You can do this by splicing in a quoted symbol. Here's a start:

```
(defmacro composable-funhouse [expr]
  `(let [~'+ ...]
     ...))
```

To prove that `composable-funhouse` solves the problem, test it in combination with `adventure-town` as defined above:

```
> (composable-funhouse (adventure-town (+ 10 10)))
100
```

Chapter 9

Macro-writing macros

Macros aren't limited to generating run-time code. Macros which generate macros can be very useful, but they can be challenging to get right. It's hard enough to write code which writes the right code but writing the right code to write the right code is even trickier.

What gets hard is managing the various levels of quotation. When a macro runs, you can think of it as `eval`ing it's quoted form. When you generate a macro, there are two levels of evaling happening, so you need two levels of quotation to balance it out. One level for your outer macro, and another for the body of the macro you're generating.

Consider the following examples:

```
> (def x 'x-value)
> `x                      ; => macro-playground.core/x
> ``x                     ; => (quote macro-playground.core/x)
> ```~x                   ; => macro-playground.core/x
> ````~~x                 ; => x-value
> ```~'~x                 ; => (quote x-value)
> ``'~~x                  ; => (clojure.core/seq
                          ;   (clojure.core/concat
                          ;   (clojure.core/list (quote quote))
                          ;   (clojure.core/list x-value)))
> `'```~~x   ; => (quote x-value)
> `'``~'~'x   ; => (quote (quote x))
> (eval `'``~'~'x)   ; => (quote x)
> (eval (eval `'``~'~'x)) ; => x
```

As you can see, depending on what you want to happen with a value,

and which level of quotation that value originates, you can end up needing to carefully quote, syntax quote, and unquote values.

Here's a somewhat pathological example. Let's create a macro `deffoo` that works as follows:

```
> (deffoo bar)
> (defbar baz)
> (baz 2)
(foo bar baz 2)
```

`deffoo` generates `defbar` based on its argument. `defbar` generates a function based on the `defbar` argument name, which returns a list of the various arguments that occurred along the way.

This is one way to define that macro:

```
(defmacro deffoo [fooname]
  (let [macroname (symbol (str "def" fooname))]
    `(defmacro ~macroname [name#]
       `(defn ~name# [~'x#]
          (list '~'~'foo '~'~fooname '~name# ~'x#)))))
```

Pretty straightforward until you get to the second syntax-quote. The ∼`name#` is just interpolating a value from the previous quote level. This is just like a regular macro, but the variable name happens to be an auto-gensym from the inner `defmacro`.

The ∼`'x#` is weirdness #1. If we just use `x#`, as you might expect (and I tried on my first pass at writing this), you end up with a fully qualified symbol. This is because the `x#` gets turned into a regular symbol by the inner syntax-quote, but then the outer syntax-quote fully qualifies it, leading to a name error trying to use a fully-qualified name as a function argument. Using ∼`'x#` expands the auto-gensym at the first level of syntax-quotation, then splices it into the second, so it remains unqualified.

Now let's start at the end of our `list` on the last line. To refer to the argument to the function, we use the same ∼`'x#`.

To include the name of the function in the output, we need to interpolate it, the value of `name#`, into our list. We need to quote it, however, so that it isn't executed. Hence `'`∼`name#`

`'`∼`'`∼`fooname` starts to get a little ridiculous. `fooname` is interpolated, then quoted, then interpolated, then quoted. `fooname` must be quoted in the function, because we want it to appear in the output. It must be quoted in

the generated macro, since `defbar` does not have a `bar` variable. Each level of evaluation strips a quote, so we must have one quote for the function, and one quote for the execution of `defbar`. We *want* evaluation of `fooname`, so it need not be quoted at the level of `deffoo`.

`'~'~'foo` just takes this one step further. We want the symbol `foo` to be unevaluated through three levels of execution, so we need three quotes. It's going through two levels of syntax-quotation, so we need two unquotes.

The good news is that this example is quite extreme with the quoting-gymnastics required. Also, it's not necessary to use nested backquotes. Constructing sequences in your generated macros manually can sometimes be easier.

9.1 Exercise

When dealing with maps or records with the same keys again and again, it can be inconvenient to continually extract values. It would be nice to be able to bundle up a set of commonly used names with a short-hand syntax.

Create a macro called `deflet` which which takes a bundle name and a list of attributes and creates a customer `let` function to grab them all at once.

With this macro, we can first define our `let` function:

```
> (deflet circle [cx cy radius])
```

With that, we can now use a newly-generated `let-circle` macro to extract the values from a "circle" record.

```
> (let-circle {:cx 10 :cy 20 :radius 100}
    [cx cy radius])
[10 20 100]
```

Chapter 10

Macrolet

Sometimes we want to build macros that introduce new language within them, macros that work within other macros to enable otherwise impossible features.

One way to do this is to define multiple macros that cooperate, but this won't work in cases where the outer macro must influence the behavior of those used within. Another approach is to use code walking, but this can lead to the same composability issues we explored earlier.

The `clojure.macro.tools` namespace, which is not included with Clojure by default but available via maven or leiningen, provides a tool for accomplishing this: `macrolet`.

`macrolet` is almost exactly like `letfn`, but it creates macros and not functions. What's great is that these macros are lexically scoped. They respect let bindings internally and are therefore composable. Further, the macros aren't available outside of the block.

Here's an example:

```
> (macrolet [(foo [] `(inc ~'x))]
            (let [x 1]
              (foo)))
2
> (foo)
CompilerException java.lang.RuntimeException: Unable to resolve
    symbol: foo in this context...
```

10.1 Exercise

Create a looping construct called `reloop` that works like `loop`, but lets you refer back to the previous versions of variables. Instead of `recur`, use `rec`. (`recur` can't be overwritten by `macrolet`.) Finally, `(prev x)` should return the previous version of the variable `x`.

Here's an example computing a Fibonnaci number:

```
(reloop [x 0]
        (cond
          (zero? x) (rec (inc x))
          (< x 15) (rec (+ (prev x) x))
          :else x))
```

`reloop` should compile into a regular `loop` statement. `prev` also need only apply to variables bound by the `reloop`.

Chapter 11

Symbol macros

One final tool to introduce for macro-writing. `symbol-macrolet` works exactly like `let`, but values bound to variables are textually replaced at compile-time instead of being values computed at runtime. Think of it as a lexically scoped `#define`.

Here's an example:

```
> (symbol-macrolet [x (println "foo")]
    [x x x])
foo
foo
foo
[nil nil nil]
```

Symbol macros provide a composable way to achieve the textual substitutions you would otherwise do by code walking.

11.1 Exercise

Create a version of `deflet` (the exercise from the macro-writing-macros section) whose let blocks do not extract values from the passed-in value unless used.

Thus, if we

```
> (deflet square [x y w h])
> (let-square my-square
    x)
```

Only `:x` will be extracted from `my-square`.

Chapter 12

Acknowledgements

Large parts of this were inspired by `On Lisp` by Paul Graham and `Let Over Lambda` by Doug Hoyte.