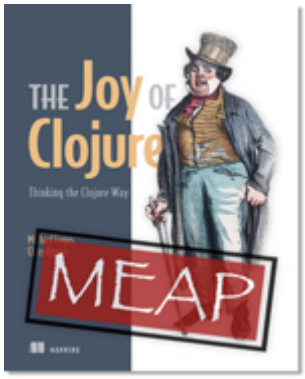
MANNING PUBLICATIONS

Macros combining forms

An article from



Joy of Clojure
EARLY ACCESS EDITION

Thinking the Clojure Way
Michael Fogus and Chris Houser
MEAP Release: January 2010
Softbound print: November 2010 (est.) | 400 pages
ISBN: 9781935182641

This article is taken from the book Joy of Clojure*. The authors discuss using macros to combine forms.*

Macros are often used for combining a number of forms and actions into one consistent view. In this article, we will use `do-until to` show how macros can be used to combine a number of tasks in order to simplify an API. For instance, Clojure's `defn` macro is this type of macro because it aggregates the processes of creating a function, including:

- Creating the corresponding function object using `fn`.
- Checking for and attaching a documentation string.
- Building the `:arglists` metadata.
- Binding the function name to a Var.
- Attaching the collected metadata.

You could indeed perform all of these steps over and over again every time you wanted to create a new function but, thanks to macros, you can instead use the more convenient `defn` form. Regardless of your application domain and its implementation, programming language boilerplate code inevitably occurs.

Identifying these repetitive tasks and writing macros to simplify and reduce or eliminate the tedious copy-paste-tweak cycle can work to reduce the incidental complexities inherent in a project. Macros differ from techniques familiar to proponents of Java's particular flavor of object-oriented style, including hierarchies, frameworks, inversion of control, and the like in that they are not treated differently by the language itself. That is, Clojure macros work to mold the language into the problem space rather than forcing you to mold the problem

space into the constructs of the language. There is a specific term for this—*domain-specific language*—but, in Lisp, the distinction between DSL and API is thin to the point of transparency.

Envision a scenario where you would like to be able to define Vars that notify when their root bindings change. You could do this using the `add-watch` function that allows for the attachment of a "watcher" to a reference type that is called whenever a change occurs within. The `add-watch` function itself takes three arguments: a reference, a watch function key, and a watch function called whenever a change occurs. You could certainly enforce that every time you wish to define a new Var, and the following steps must be taken:

- Define the Var
- Define a function (maybe inline to save a step) that will be the watcher
- Call `add-watch` with the proper values

A meager three steps is not too cumbersome a task to remember in a handful of uses but, over the course of a large project, it's easy to forget and/or morph one of these steps when the need to perform them many times occurs. Therefore, perhaps a better approach is to define a macro to perform all of these steps for us, as the following definition does:

```
(defmacro def-watched [name & value]
  `(do
     (def ~name ~@value)
     (add-watch (var ~name)
                :re-bind
                (fn [~'key ~'r old# new#]
                  (println old# " -> " new#)))))
```

Minus symbol resolution and auto-gensym, the macro called as `(def-watched x 2)` expands into roughly the following:

```
(do (def x 2)
    (add-watch (var x)
               :re-bind
               (fn [key r old new]
                 (println old " -> " new))))
```

The results of `def-watched` are thus:

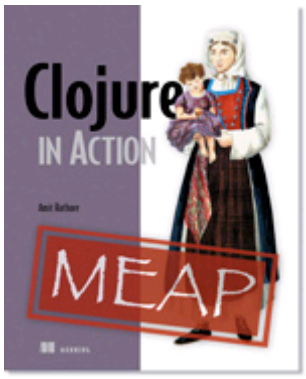
```
(def-watched x (* 12 12))
x
;=> 144

(def x 0)
; 144 -> 0
```

Lisp programs, in general, and Clojure programs, specifically, use macros of this sort to vastly reduce the boilerplate needed to perform common tasks.

Summary

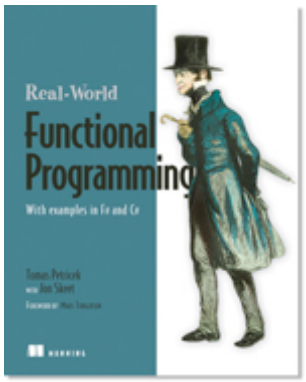
One potential use case for macros is taking one form of an expression and transforming it into another form. We looked at using macros to combine forms.

Here are some other Manning titles you might be interested in:



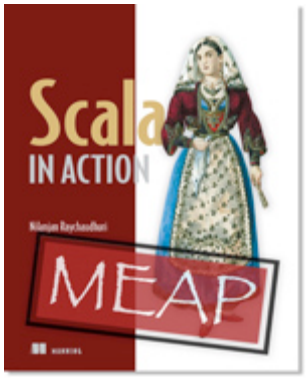
[Clojure in Action](#)
EARLY ACCESS EDITION

Amit Rathore
MEAP Release: November 2009
Softbound print: Winter 2010 | 475 pages
ISBN: 9781935182597



[Real-World Functional Programming](#)
IN PRINT

Tomas Petricek with Jon Skeet
December 2009 | 560 pages
ISBN: 9781933988924



[Scala in Action](#)
EARLY ACCESS EDITION

Nilanjan Raychaudhuri
MEAP Began: March 2010
Softbound print: Early 2011 | 525 pages
ISBN: 9781935182757