Bits and pieces

19 Apr 2015 • on Clojure macro development

A "dead simple" introduction to Clojure macros.

Macros are one of the topics which scares many new Clojure developers. Although I've seen many tutorials about the topic, I think that the approach used is often too complicated for someone who is new to Clojure. So I will try to explore the topic especially for developers who are new to Clojure and haven't yet grasped the macros.

You can find the code of this post at:
https://github.com/BrunoBonacci/clojure-simple-macro

Expar Hide Wrap Setting

```
DEMO: youtube.com/
Roam-highlighter S

[ALT + X]
        - Activate

[Ctrl + X] (WIN) o
        - Highligh
        - To remov

[Alt + Click]
        - Removes

[ALT + Q]
        - Remove a

[ALT + A]
        - Makes se
```

What is a macro?

The simple answer is: *A macro is a piece of code that is executed at compile time (rather than runtime) and it produces code which in turn is compiled.*

How this is possible? Many other languages have some sort of "macros", such as C/C++, Groovy and many others. However there is a fundamental difference. Languages such C and C++ have *"Text substitution macros"* while languages such as Groovy have Abstract Syntax Tree (AST) transformations. In

code, in its text form, is turned into a tree structure which describes the code in terms of its logical parts down to every single statement. This tree structure is then made available to the user (developer) who can *make some modifications* before the compiler turns the `AST` into compiled bytecode.

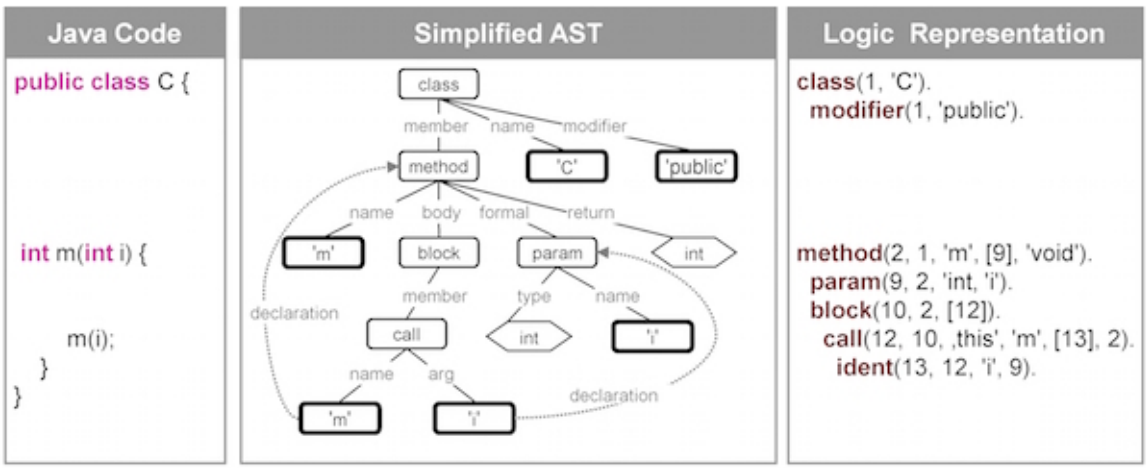
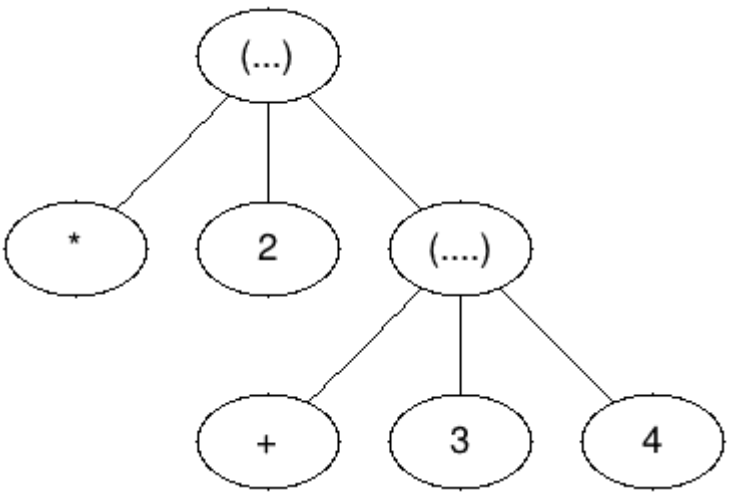


Image courtesy of: http://sewiki.iai.uni-bonn.de/_media/research/jtransformer/lmp.jpg

LISP languages (such as Clojure) don't need to transform the text representation into an Abstract Syntax Tree because the text itself is in AST form already. Clojure source files are written in terms of S-expressions (or `sexpr`) which are nothing else than *lists of lists* (= trees). Now the *list* is one of the fundamental data-structures in Clojure. Such languages which use their own data-structures to represent the source code are called *homoiconic*.

Tree structure representing the s-expression for `(* 2 (+ 3 4))`



The reason why LISP languages have this *funny looking syntax* is due to the fact that the code is written directly in their AST form. So compared to Groovy

humans and a different representation of the code for the compiler (machine). This apparently disadvantage makes macros in Clojure extremely easy to write in comparison to Groovy. Because macros are so easy to write, they can become a powerful way to process your source code and remove duplication, make your code more readable, or automatically inject additional code in your source code.

How can I write a macro?

In Clojure there are two ways to write the macros. The first one is to read and process the AST which is just a *list of lists*. This is very much the Groovy way, with the difference that you already know how to manipulate a Clojure's lists and you don't need a different API. The second way is very much like a *templating language*. Among these template languages there some like: Velocity or Mustache which are very popular.

For example a Velocity template looks like:

```
Hello ${user_name},
Today is the ${date} and we have ${num_users} users currently online.
```

Basically it is just plain text, very much like the output you want to produce, minus a few placeholders in the places where the content must be generated dynamically. Once rendered this text might just look like:

```
Hello John,
Today is the 19th April and we have 1,653 users currently online.
```

Like a templating engine, Clojure `syntax-quote` works pretty much the same way. *You start by writing the target code in the way you want to see it* and then put the placeholders.

For example let's write a macro which wraps a code execution with a `try`/`catch`, and in case of a exception is raised, a default value is used. For

Expar Hide Wrap Setting

```
DEMO: youtube.com/
Roam-highlighter S

[ALT + X]
        - Activate
[Ctrl + X] (WIN) o
        - Highligh
        - To remov

[Alt + Click]
        - Removes

[ALT + Q]
        - Remove a

[ALT + A]
        - Makes se
```

`java.lang.ArithmeticException` when `b` is `0`.

```
(try
  operation
  (catch Exception x
    dafault-value))
```

Where `operation` and `default-value` may vary from case to case. So if this was a Velocity template we would write the code in this way.

```
;; example using Velocity template.
(try
  ${operation}
  (catch Exception x
    ${dafault-value}))
```

In Clojure we can write a template using the syntax-quote (the ` backquote character) form.

```
;; if we write
(println "hello")
;;> hello        [in stdout]

;; it is executed and produces the output "hello"
;; however if we write:
`(println "hello"))
;;=> (clojure.core/println "hello")

;; note the namespace is added automatically
;; this actually produces a template
;; with the enclosed forms.
```

So if we want to re-write our template using `syntax-quote` we will write as follow:

```
`(try
   ~operation
   (catch Exception x#
     ~dafault-value)))
```

The *tilde* (`~`) works pretty much in the same way of `${placeholders}` of velocity. To avoid name conflicts with the local variables you have to add a `#` at the end of the name, which will generate a unique symbol name (see

our code.

```
(defmacro default-to [default-value operation]
  `(try
     ~operation
     (catch Exception x#
       ~default-value)))
```

This is a fully working template, not much different from the Velocity's style template. So let's see the template in its rendered form:

```
(macroexpand-1
 '(default-to 10 (/ 1 4)))

;;=>
 (try
   (/ 1 4)
   (catch java.lang.Exception x__8493__auto__
     10))
```

As you can see this was the template we designed earlier. Notice that the placeholders have been replaced.

macroexpand and macroexpand-1 both apply the given template and return the code after the placeholders have been replaced. The difference between `macroexpand` and `macroexpand-1` is that the latter does only one level of expansion, which means that if the code, after the template has been expanded, it still contains more macros, those won't be expanded; conversely `macroexpand` will recursively expand all the templates until there is no more macro code to expand.

Here there are a couple of interesting things to notice. Firstly the `x#` which it has been transformed into a unique local variable `__8493__auto__`. Another interesting thing is that the symbol `Exception` has been expanded to the fully qualified form (with namespace/package). Using fully qualified symbols will avoid problems of name clashing when the macros is used. A common example can be `log/debug`. If while you writing the macro you refer to a `log` namespace which contains a function named `debug`, and when the macro user tries to use the macro in a different namespace without having a local `log/debug` defined, or even worse, having different implementation many strange errors could occur. By automatically transforming all symbols in

Interesting note to observe is that `operation` was replaced with the actual code `(/ 1 4)` and not its result. Keep this in mind for later as we'll see how this can sometime be a problem.

Let's try to see how our macro works.

```
(default-to 10 (/ 1 4))
;;=> 1/4

(default-to 10 (/ 1 0))   ;; Exception
;;=> 10
```

Now let's improve our macro and ask it to log a message to a logging system in case of exceptions. We can use timbre logging library.

So let's starting by load the namespace.

```
;; add the dependency in your project.clj
;; [com.taoensso/timbre "3.4.0"]
;; and restart your REPL

(require '[taoensso.timbre :as log])
```

and now let's update our little macro.

```
(defmacro default-to [default-value operation]
  `(try
     ~operation
     (catch Exception x#
       (log/debug "The following error occurred:" x#
                  ", defaulting to:" ~default-value)
       ~default-value)))

(defn load-default-value []
  (println "loading default value from database")
  (comment loading from db)
  3)
```

Let's assume this time that the default value is retrieved using the `load-default-value` function and use `macroexpand-1` to see the generated code.

```
(macroexpand-1
 '(default-to (load-default-value)
    (/ 1 0)))
```

```
 (/ 1 0)
 (catch  java.lang.Exception x__9554__auto__
  (taoensso.timbre/debug "The following error occurred:" x__9554__auto__
                         ", defaulting to:" (load-default-value))
  (load-default-value)))


;; let's execute the macro
(default-to (load-default-value)
   (/ 1 0))

;;[stdout]
;; loading default value from database
;; DEBUG - The following error occurred: #<ArithmeticException
java.lang.ArithmeticException: Divide by zero> , defaulting to: 3
;; loading default value from database

;;=> 3
```

Again, you can see that the `log/debug` has been expanded to the fully qualified name. The interesting part is that the `default-value` has been expanded with the *sexpr* `(load-default-value)` and not with its result. This is important as, in case of exception, the `(load-default-value)` **will be called twice** because it appear twice in the macro. To verify this behaviour you may check the *stdout* and see the message: *"loading default value from database"* appearing twice. Since the `(load-default-value)` produces side effect, it might be an undesirable behaviour. So let see how we can fix it.

```
(defmacro default-to [default-value operation]
  `(try
     ~operation
     (catch Exception x#
       (let [default# ~default-value]
         (log/debug "The following error occurred:" x#
                    ", defaulting to:" default#)
         default#))))


(macroexpand-1
 '(default-to (load-default-value)
     (/ 1 0)))

;;=>
(try
 (/ 1 0)
 (catch java.lang.Exception x__6188__auto__
  (clojure.core/let [default__6189__auto__ (load-default-value)]
```

```
    default__6189__auto__)))
```

We used a `let` form to create a local var for `default-value` called `default#` (remember that the `#` sign a the end of the symbol generates unique symbol name) and assign the value of the `~default-value` expansion, then we can use the local var `default#` which will contains only the result of the operation, in every place we needed the `default-value`. Because we perform only one expansion of `~default-value`, the `(load-default-value)` code will be executed only once.

Last improvement we can make to this simple function is to account for a code block as operation. If instead of specifying only one operation we want to be able to specify multiple forms, we need to change the macro signature and accommodate the new params.

```
(defmacro default-to [default-value & operations]
  `(try
     ~@operations
     (catch Exception x#
       (let [default# ~default-value]
         (log/debug "The following error occurred:" x#
                    ", defaulting to:" default#)
         default#))))

(macroexpand-1
 '(default-to (load-default-value)
    (println "This is a multi sexpr operation")
    (println "Infact it will be captured by &operation as a list")
    (/ 1 0)))

;;=>
(try
 (println "This is a multi sexpr operation")
 (println "Infact it will be captured by &operation as a list")
 (/ 1 0)
 (catch java.lang.Exception x__6494__auto__
  (clojure.core/let [default__6495__auto__ (load-default-value)]
   (taoensso.timbre/debug "The following error occurred:" x__6494__auto__
                          ", defaulting to:" default__6495__auto__)
   default__6495__auto__)))
```

By changing the macro signature from `[default-value operation]` into its variadic form `[default-value & operations]` we give the possibility to accept a variable number of parameters (variadic functions/macros) which

use `~@operations`. The form `~@` called unquote-splicing expands a list into its individual elements.

Let's see a bit more about `unquote-splicing`:

```
;; range return a sequence of numbers
(range 10)
;;=> (0 1 2 3 4 5 6 7 8 9)

;; if we use the normal unquote
;; number will appear wrapped in a sequence
`(max ~(range 10))    ;; wrong, need (apply max ...)
;;=> (clojure.core/max (0 1 2 3 4 5 6 7 8 9))

;; notice here that the number are NOT wrapped
;; into the sequence but they appear directly
`(max ~@(range 10))
;;=> (clojure.core/max 0 1 2 3 4 5 6 7 8 9)
```

This concludes this basic introduction to Clojure's macros. By now you should have all the necessary tools to write basic macros.

Conclusion

Creating Clojure macros is a powerful way to write concise and beautiful code, or turn your declarative code into functional code. There are few takeaways from this blog post:

- When you can write function, not macros. *Macros are not composable and are not usable as high-order functions*, so if you can achieve the same result with a function, write a function instead.
- When writing macros as templates use always backquote (or backtick character) to create code templates
- Denote your placeholders with the tilde (`~`)
- If a placeholder appear more than once in your template wrap it with `let` binding and create a local var instead.
- For every var inside the template create a generated symbol by appending the hash sign (`#`) at the end of the symbol's name
- Expand lists with the `unquote-splicing` (`~@`) when necessary
- Use `macroexpand-1` and `macroexpand` to see the generated code.
- Unless your macro is for internal use (same namespace), *test your macros in a different namespace* to catch visibility issues.

If you want to get a deeper understanding of the Clojure's macros check the following links:

- Quoting without confusion
- http://www.braveclojure.com/writing-macros/
- Kyle Kingsbury's "Clojure from the ground up: macros"
- John Aspden's introduction in three parts: part-1 part-2 part-3

You can find the code of this post at:
https://github.com/BrunoBonacci/clojure-simple-macro

Many thanks to Sathya for his feedbacks

Emacs Incanter Hack

4 years ago • 6 comments

Between bits and bytes and all other pieces.
A tech blog about Clojure, …

Microservices with API Gateway, AWS …

4 years ago • 2 comments

Between bits and bytes and all other pieces.
A tech blog about Clojure, …

Learn Clojure - Cloj Basics

4 years ago • 2 comments

Between bits and bytes all other pieces.
A t blog about Clojure, …

Michael Carolin • 2 years ago

I found this really clear, and the Velocity/Mustache template analogies were particularly helpful. Thank you!

Reply • Share ›

brunobonacci Mod Michael Carolin • 2 years ago

Thanks for the comment!

1 Reply • Share ›

Expar Hide Wrap Setting

```
DEMO: youtube.com/
Roam-highlighter S

[ALT + X]
        - Activate

[Ctrl + X] (WIN) o
        - Highligh
        - To remov

[Alt + Click]
        - Removes

[ALT + Q]
        - Remove a

[ALT + A]
        - Makes se
```