Clojure Macros

((((((((((((((()))))))))))))))))))))))))

This is nice and simple!

String -> Reader -> Evaluator

Example

```
(+ 1 2)

gives me 3
```

- (+ 1 2) is a persistant list - an output from Reader.
- When evaluated, it takes the first symbol, identifies that it is a function.
- Evalutes the arguments and pass it to function +

AST

The output from reader will be an AST,
which is basically clojure's data structure typically List - one of the easiest to represent trees.

```
1 <--- (+) ---> 2
```

Is that a clojure structure?

`Yes! it is (list + 1 2)`

list of symbols/forms/values - unevaluated!

Tweak the Reader output

Hey Reader! You have to do a few more things before you pass the weird tree on to the evaluator!

That means?

Typically something like this:

```
(def readoutput (read-string “(1 + 1)”)

(def input-for-eval
(list (second readoutput)
(first readoutput)
(last readoutput))

;; Passing on to the evaluator
eval(input-for-eval)

;; Result is 2
```

Macros?

```clojure
(defmacro thatisit! [argument]
  (list (second argument)
  (first argument)
  (last argument)))
```

Macros give you a convenient way to manipulate lists before Clojure evaluates them.

```clojure
;; gives you the expaned form that
;; is passed on to the evaluator
(macroexpand 'thatisit(1 + 2))
```

And where to use?

That means you can use Clojure to extend itself so you can write programs however you please. In other words, macros enable syntactic abstraction.

Syntactic abstraction?

Allowed	syntactic abstraction
`(+ 1 2)`	`(1 + 2)`
`(filter ..((map ..[1 2]))`	`(-> [1 2] (map...) (filter...)`

Those abstractions were made possible through macros