Leveraging macros to reimplement language features

Veit Heller

March 13, 2019

Clojure Meetup Berlin

Agenda

- Comments
- Conditionals
- Lists
- Local bindings

Agenda

Shameless plug: agenda of my series of blog post on Scheme macros!

- Modules
- Generic functions
- Classes/Inheritance
- Keyword Arguments in functions
- Design by Contract
- Green Threads

Comments

Comments

```clojure
(defmacro mcomment [& args] nil)
```

Listing 1: Implementing comments

Comments

```
(mcomment
  (+ 40 3)
  (undefined-function)
  (println "nope"))
```

Listing 2: Using comments

Conditionals

```clojure
(defn mtrue [x y] (x))
(defn mfalse [x y] (y))
```

Listing 3: Booleans as functions

```
(defn mtrue [x y] (x))
(defn mfalse [x y] (y))

(defmacro mif [cnd then else] `(~cnd #(do ~then) #(do ~else)))
```

Listing 4: `if` as a macro

```
(mif mfalse
  (println "nope")
  (println "yup"))
```

Listing 5: Using `if`, the macro

cond

```
(defmacro mcond [& args]
  (when args
    `(if ~(first args)
       ~(second args)
       ~(cons 'mcond (rest (rest args))))))
```

Listing 6: cond as a macro

cond

```
(mcond
  (= 1 2) (println "no")
  (= 2 3) (println "still no")
  true (println "yup"))
```

Listing 7: Using cond, the macro

Lists

cons, car, cdr

```
(defn mcons [h t] #(if % h t))
(defn mcar [l] (l true))
(defn mcdr [l] (l false))
```

Listing 8: `cons`, `car`, and `cdr` as functions

cons, car, cdr

```
(println (mcar (mcdr (mcons 1 (mcons 2 3)))))
```

Listing 9: Using `cons`, `car`, and `cdr`, the macros

```
(defn mlist [& args]
  (loop [res nil, args args]
    (if (empty? args)
      res
      (recur (mcons (first args) res) (rest args)))))
```

Listing 10: Reimplementing `list`

```
(defmacro mquoted [& args] `(apply mlist '~args))
```

Listing 11: `mquoted`

Local bindings

let

```
(defmacro mlet [args body]
  (if (= (count args) 2)
    `((fn [~(first args)] ~body) ~(second args))
    `((fn [~(first args)]
        (mlet ~(rest (rest args)) ~body))
      ~(second args))))
```

Listing 12: Reimplementing `let`

```
(println (mlet [x 1 y 2] (+ x y)))
```

Listing 13: Using `let`

References

- “Reversing the technical interview” uses the lists for great effect: `https://aphyr.com/posts/340-reversing-the-technical-interview`
- These slides: `https://github.com/hellerve/talks`
- A series of blog posts on Scheme macros: `https://blog.veitheller.de/scheme-macros`

Thank you!

Questions?

Slides at `https://github.com/hellerve/talks`