



LAMBDA LOUNGE

## Macros in Common Lisp

**Christopher Mark Gore**

<http://www.cgore.com>

[cgore@cgore.com](mailto:cgore@cgore.com)

 @cgore

**Thursday, May 1, AD 2014**

## Getting Started with Common Lisp

### 1. Install Linux.

`http://aptosid.com`

### 2. Install SBCL and some libraries.

```
apt-get install sbcl{,-doc,-source} \  
cl-{asdf,cffi}
```

### 3. Install Emacs and SLIME *(Not strictly required.)*

```
apt-get install emacs{,-goodies-el} cl-swank \  
cl-swank slime common-lisp-controller
```

---

Since my new employer bought me a shiny new MacBook Pro:

```
brew install sbcl
```

## **Lisp Macros Are Vastly Superior To:**

1. Not having macros at all.
2. Vim/Emacs<sup>a</sup> macros.
3. Microsoft Word/Excel macros.
4. C pre-processor macros.
5. Scheme “hygienic” macros.

---

<sup>a</sup>Excluding `defmacro` in Elisp, which is equivalent.

## Defun versus Defmacro

We can define functions with `defun`, and we can define macros with `defmacro`. These two definitions achieve the same goal in quite different ways.

```
(defun add-one (x)  
  (+ 1 x))
```

```
(defmacro add-one (x)  
  '(+ 1 ,x))
```

## Why Not Just Functions?

You should prefer functions instead of macros if they can do the job. There are lots of operations that can only be done with macros: functions can't make it happen.

- Linguistic extensions
- Preventing the execution of the arguments
- Controlling the execution of the arguments
- Rewriting the arguments before they are executed
- Natural DSLs

## **How Macros Work**

Macros have two main features:

1. Controlling, preventing, or manipulating the evaluation of their arguments.
2. Local expansion within the calling context.

If you really want a macro, it is because you need one of those abilities.

## Controlling Argument Evaluation

```
(defmacro upside-down (first last)
  `(progn ,last
          ,first))

(upside-down
 (format t "The~first~shall~be~last~%")
 (format t "The~last~shall~be~first~%"))
```

This prints:

```
The last shall be first
The first shall be last
```

```
(let ((x 42))
  (upside-down (setf x (/ x 2))
               (setf x (+ x 2)))) ; 22, not 23.
```

**TO DO**

**Local Expansion**



**TO DO**

**Backquote**

## Gensym

```
(defmacro broken-swap (x y)
  '(let ((temp ,x))
      (setf ,y ,x
            ,x temp)))
```

How do we avoid capturing a variable? By *generating brand new symbols*.<sup>a</sup>

```
(defmacro swap (x y)
  (let (
```

---

<sup>a</sup>In real life, just use `psetf`.

## **Macro Expansion**

**TO DO**

**Macros are Programs Too**

**TO DO**

**TO DO**

**WITH-\* Macros**

## **Deconstructing Parameter Lists**

**TO DO**

## **Anaphoric Macros**

**TO DO**

## Compile Time and Run Time

In Lisp, this distinction still exists, but it doesn't work at all like you are used to.

**TO DO**



## **Redefining Macros**

**TO DO**

**TO DO**

**Symbol Macros**

**TO DO** **DSLs via Macros**

**Growing the Language Itself via Macros**  
**TO DO**

**Ancient History: F-Expressions**

**TO DO**

## **Reader Macros**

These are really cool and complicated. You can completely redefine the language with these, making your own language, even one with syntax (Ruby, C, C++, etc.)

Perhaps another day.

***Questions?***