LAMBDA LOUNGE

Macros in Common Lisp

Christopher Mark Gore

`http://www.cgore.com`

`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[a] macros.
3. Microsoft Word/Excel macros.
4. C pre-processor macros.
5. Scheme “hygenic” macros.

[a] 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 shouls 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

Backquote

Backquotes let you quote out a list, but selectively evaluate some of the list elements. This shorthand notation makes it really easy to build up complicated s-expressions on the fly.

```
(let ((a 12)
      (b 'x)
      (c '(q r s)))
  `( a  b   c)          ; Becomes '(a  b c)
  `(,a  b   c)          ; Becomes '(12 b c)
  `( a ,b   c)          ; Becomes '(a  x c)
  `( a  b  ,c)          ; Becomes '(a  b (q r s))
  `( a  b ,@c)          ; Becomes '(a  b  q r s)
  `( a  b ,(+ 1 2 3))   ; Becomes '(a  b 6)
```

Nested Backquotes

Backquotes are a bit confusing when they are nested inside each other. It is generally best to avoid it as much as possible, but sometimes it is necessary. The general rule is they apply inside-to-outside.

```
(let ((a 'x))
  (eval `(let ((a 'y)
               (b ',a))
           `(,a ,b)))) ; Returns '(y x)
```

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.
```

Local Expansion

TO DO

Gensym

How do we avoid capturing a variable? By *generating brand new symbols.*[a] Each call to `gensym` makes a brand new uninterned symbol.

```
(defmacro swap (x y)
  (let ((temp (gensym)))
    `(progn (setf ,temp ,x)
            (setf ,x ,y)
            (setf ,y ,temp))))
```

[a]In real life, just use `psetf`.

Macro Expansion

TO DO

Macros are Programs Too

TO DO

WITH-* Macros

TO DO

Destructuring 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

Symbol Macros

TO DO

DSLs via Macros

TO DO

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?