Common Lisp: the programmable programming language

Sebastián González
25 June 2009
with many thanks to Pascal Costanza
• If you give a person a fish, he can eat for a day.

• If you teach a person to fish, he can eat his whole life long.

• If you give a person tools, he can make a fishing pole, even build a machine to crank out fishing poles. In this way he can help other persons to catch fish.

• How do we achieve this in a language?
Growing a Language

- How to design a language?
  - Build The Right Thing from the start.
  - Build a small language.
  ✓ Start small, and **plan for growth**.

- Design a language that can be grown by its **users**.
  - Expose the tools used to build the language to users.
  - Have user-defined constructs **look** as just one more part of the language.

- CL: started practical, and was planned for growth.
let the user participate in all stages of computation
... including the **read** phase!
Macros

• This is code: (+ 1 2 3)
• This is data: '(+ 1 2 3)
• A macro is a function that generates code: it takes code as argument and returns new code.
• The step of building the new expression is called _macroexpansion_.
• Macros are used at read time, rather than evaluation time.
• **READ**: parse code, and macroexpand.
A Bit of Background

- \[ \text{(let ((a 1) (b 2) (c 3) (d 4))} \]
  \[ \text{(list a b c d))} \]
  \[ \rightarrow (1 2 3 4) \; ; \text{everything is evaluated} \]

- \[ \text{(let ((a 1) (b 2) (c 3) (d 4))} \]
  \[ \text{(list 'a b c d))} \]
  \[ \rightarrow (A 2 3 4) \; ; \text{not everything is evaluated} \]
A Bit of Background

- (let ((a 1) (b 2) (c 3) (d 4))
  (list 'a 'b 'c d))

  → (A B C 4) ; very little is evaluated

- Here is a more concise way to write this:

  (let ((a 1) (b 2) (c 3) (d 4))
   `(a b c ,d))

  → (A B C 4) ; very little is evaluated
Backquote

- '(a b c d) uses quote
- `(a b c ,d) uses backquote
- backquote allows evaluating parts of an expression explicitly marked with a comma
- you can’t do this with quote
Backquote

- `(a b c) ↔ '(a b c) ↔ (list 'a 'b 'c)
- `(a ,b c ,d) ↔ (list 'a b 'c d)
- `(let ((b 2)) `(a (,b c)))
  → (A (2 C))
- `(let ((a 1) (b 2) (c 3))
  `(a b ,c (',(+ a b c)) (+ a b) 'c '(((,a ,b))))
  → (A B 3 ('6) (+ A B) 'C '(((1 2))))
Backquote

- (let ((list '(1 2 3)))
  `(a b ,@list c d))

  → (a b 1 2 3 c d)

- ,@ splices into the surrounding list
  (so there must be a surrounding list!)
Macro Example

• (defun while-fun (predicate thunk)
   (when (funcall predicate)
     (funcall thunk)
     (while-fun predicate thunk)))

• (defmacro while (expression &rest body)
   (list 'while-fun (list 'lambda '() expression)
     (list* 'lambda '() body)))

• Or more aesthetical:
  (defmacro while (expression &body body)
    `(while-fun (lambda () ,expression)
      (lambda () ,@body)))
Note

- Backquote is independent from macros.
- `(defun greet (name)
   `(hello ,name))

...is a function!
Macro Function in Action

- (funcall (macro-function 'while)
  '(while (< i 10)
    (print (incf i)))
  lex-env)

→ (while-fun (lambda () (< i 10))
    (lambda ()
      (print (incf i))))
Macro Expansion

(let ((i 0))
  ...
  (while (< i 10)
    (print (incf i))
  ...
)
→
(let ((i 0))
  ...
  (while-fun (lambda () (< i 10))
    (lambda ()
      (lambda ()
        (print (incf i))))))
  ...
)
Why Macros?

- Question: why not just say this?

\[
\text{(while (lambda () (< i 10))}
\text{ (lambda () (print (incf i))))}
\]
Syntactic Abstractions

(while (lambda () (< i 10)))
  (lambda () (print (incf i))))

• The while function **leaks**: you need to know details about its implementation.

• That is, the fact that it uses closures.

• The Law of Leaky Abstractions (Joel Spolsky)

• Leaky abstraction: an abstraction that exposes ("leaks") details it is supposed to be abstracting away.
Alternative Implementations

- `(defmacro while (expression &body body)
  `(do () ((not ,expression)) ,@body))`

- `(defmacro while (expression &body body)
  `(tagbody
    start
    (unless ,expression (go end))
    ,@body
    (go start)
    end))`
Abstractions

- Syntactic abstractions hide implementation details, just like functional abstractions.

- Hiding implementation details allows you to change your mind later on.

- It also allows the users of your library to think purely in terms of what they care about.
Abstractions

- (while-fun (lambda () (< i 10))
  (lambda ()
    (print (incf i))))

vs.

(while (< i 10)
  (print (incf i)))

- Macros allow user-defined syntactic abstractions which look as any other abstraction does.
How to Write Macros

• You need some functionality?
• Decide if the macro is really necessary.
• Write down the syntax of the macro.
• Figure out what the macro should expand into.
• Use defmacro to implement the syntax/expansion correspondence.
Idea

• Have a looping construct similar to dotimes...

(dotimes (i 10)
  (format t "~d " i)) ➞ “0 1 2 3 4 5 6 7 8 9”

... but for prime numbers

• (do-primes (p 0 19)
  (format t "~d " p)) ➞ “2 3 5 7 11 13 17 19”

• Could be needed in writing cryptographic software.
Is a Macro Necessary?

• (defun square (x) (* x x))

vs.

(defmacro square (x) `( (* ,x ,x)))

• Most of the time there is a clear distinction between the cases which call for macros and those which don’t.

• A proper ‘while’ can be defined only with a macro, and so does do-primes.
Syntax and Expansion

• Interface (syntax):

(do-primes (var start end)
  body)

• Behaviour (semantics):

(do ((var (next-prime start) (next-prime (1+ var)))))
  ((> var end))
  body)
Implement the Macro

- \( (\text{do-primes} \ (\text{var} \ \text{start} \ \text{end}) \ \text{body}) \)

- \( (\text{defmacro} \ \text{do-primes} \ (\text{var-and-range} \ \&\text{body} \ \text{body}) \)
  \( \) (let ((\(\text{var} \ (\text{first} \ \text{var-and-range})\))
    (\(\text{start} \ (\text{second} \ \text{var-and-range})\))
    (\(\text{end} \ (\text{third} \ \text{var-and-range})\)))
  \(\) `(\(\text{do} \ ((\text{var} \ (\text{next-prime} \ ,\text{start}) \ (\text{next-prime} \ (1+ \ ,\text{var})))))
    (\(> \ ,\text{var} \ ,\text{end})\))
  ,@\text{body}))\)

- Actually, you don't need to take apart var-and-range by hand.
Destructuring Lambda Lists

- `(do-primes (var start end) body)`

- `(defmacro do-primes ((var start end) &body body)
  `(do ((,var (next-prime ,start) (next-prime (1+ ,var))))
       ((> ,var ,end))
       ,@body))`

- Automatic syntax error checking for free.
- Integrates with IDEs such as SLIME.
- Destructuring parameter lists can contain &optional, &key, &rest and also nested destructuring lists.
Test the Macro

- **Expansion:**
  
  (macroexpand
   '(do-primes (p 0 19)
     (format t "~d " p))))

  ➞

  (DO ((P (NEXT-PRIME 0) (NEXT-PRIME (1+ P))))
      ((> P 19))
      (FORMAT T "~d " P))

- **Behaviour:**

  (do-primes (p 0 19)
    (format t "~d " p)) ➞ "2 3 5 7 11 13 17 19"
Plugging the Leaks

• Principle of Least Astonishment:
  ➡ Number of evaluations
  ➡ Parameter order
  ➡ Variable capture
Number of Evaluations

- (do-primes (p 0 (random 100))
  (format t "~d " p))

- Expansion:
  (DO ((P (NEXT-PRIME 0) (NEXT-PRIME (1+ P))))
    (> P (RANDOM 100)))
  (FORMAT T "~d " P))

- Why is it a leak in the abstraction?

- How to fix it?
Parameter Order

- Fixed version:

```lisp
(defmacro do-primes ((var start end) &body body)
  `(do ((ending-value ,end)
       (,var (next-prime ,start) (next-prime (1+ ,var))))
       ((> ,var ending-value))
       ,@body))
```

- One new leak. What’s wrong?
Variable Capture

- Fixed version:

```
(defmacro do-primes ((var start end) &body body)
  `(do ((,var (next-prime ,start) (next-prime (1+ ,var))))
       (ending-value ,end))
    ((> ,var ending-value))
   ,@body))
```

- What’s wrong? Consider:

```
(do-primes (ending-value 0 10)
  (print ending-value))

(let ((ending-value 0))
  (do-primes (p 0 10)
    (incf ending-value p))
  ending-value)
```
Variable Capture (1)

- (do-primes (ending-value 0 10)
  (print ending-value))

→

(DO ((ENDING-VALUE (NEXT-PRIME 0) ...)
     (ENDING-VALUE 19))
   ((> ENDING-VALUE ENDING-VALUE))
   (FORMAT T "~d " ENDING-VALUE))
Variable Capture (2)

- (let ((ending-value 0))
  (do-primes (p 0 10)
    (incf ending-value p))
  ending-value)

→

(LET ((ENDING-VALUE 0))
  (DO ((P (NEXT-PRIME 0) (NEXT-PRIME (1+ P)))
    (ENDING-VALUE 10))
  (> P ENDING-VALUE))
  (INCF ENDING-VALUE P))
  ENDING-VALUE)
Kinds of Capture

- **Macro argument capture**
  
  ```lisp
  (defmacro print10 (x)
    `(dotimes (i 10)
      (princ ,x)))
  ```

- **Free symbol capture**
  
  ```lisp
  (defconstant pi 3.1416)
  (defmacro sum-pi (x)
    `(+(,x pi)))
  ```

- **When does capture occur?**
Free Symbols

- A symbol $s$ occurs **free** in an expression when it is used as a variable in that expression, but the expression does not create a binding for it.

- e.g., `(let ((x y) (z 10))
  (list w x z))`

- e.g., `(let ((x x)) x)`
The **skeleton** of a macro expansion is the whole expansion, minus anything which was part of an argument in the macro call.

• (defmacro foo (x y)
   `(\(/ (\,+\ ,x\ 1\ ,y))\))

• (foo (- 5 2) 6) → (/ (+ (- 5 2) 1) 6)

• skeleton:  (/ (+ 1) )
A symbol is **capturable** in some macro expansion if

(a) it occurs free in the skeleton, or

(b) it is bound by a part of the skeleton in which macro arguments are either bound or evaluated.
Examples

- (defmacro cap1 () '(+ x 1))
- (defmacro cap2 (var)
  `(let ((x ...)
        (,var ...))
    ...))
- (defmacro cap3 (var)
  `(let ((x ...))
    (let ((,var ...)) ...)))
- (defmacro cap4 (var)
  `(let ((,var ...))
    (let ((x ...)) ...)))
Examples

• (defmacro safe1 (var)
  `(progn
    (let ((x 1)) (print x))
    (let ((,var 1)) (print ,var))))

• (defmacro cap5 (&body body)
  `(let ((x ...)) ,@body))

• (defmacro safe2 (expr)
  `(let ((x ,expr)) (cons x 1)))

• (defmacro safe3 (var &body body)
  `(let ((,var ...)) ,@body))
How To Fix Captures?

• (defmacro do-primes ((var start end) &body body)
  `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
       (ending-value ,end))
       ((> ,var ending-value))
       ,@body))

• Use symbols that will never be used outside the code generated by the macro.

  ➡ Use really unlikely names. (?)
  ➡ Define your macro in a separate package. (?)
  ➡ Use GENSYM!
How To Fix Captures?

- (defmacro do-primes ((var start end) &body body)
  (let ((ending-value-name (gensym)))
    `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
      (,ending-value-name ,end))
      ((> ,var ,ending-value-name))
      (@body))))

- GENSYM will generate a new uninterned symbol every time the macro is expanded.

- This fresh symbol cannot possibly occur in the expressions passed as arguments to the macro.
GENSYM in Action

- (do-primes (ending-value 0 10)
  (print ending-value))
  ➞
  (DO ((ENDING-VALUE (NEXT-PRIME 0) ...)
      (#:G1165 10)) ; cannot be captured
     ((> ENDING-VALUE #:G1165))
     (PRINT ENDING-VALUE))

- Remember syntax for uninterned symbols?
Recap: Rules of Thumb

Unless there's a particular reason to do otherwise:

- **Parameter order**: make sure macro arguments will be evaluated according to their position in the macro call.

- **Single evaluation**: make sure subforms are evaluated only once by storing their result in variables and using those variables instead of the original subforms.

- **No captures**: use GENSYM at macro expansion time to create variable names used in the expansion.
Uses of Macros

- Implicit quoting.
- Cosmetics.
- Evaluation control.
- Syntactic abstraction.
- Side effects.
- Macro-writing utilities.
Implicit Quoting

- (defun f (x) (+ x x))
- (setf (fdefinition 'f)
  (lambda (x) (+ x x)))
• (let ((x 42)
     (y 4711))
     (+ x y))

• ((lambda (x y) (+ x y)) 42 4711)
Evaluation Control

- Conditional evaluation: if, cond, when, unless, etc.
- Delayed evaluation: delay, force, run-in-thread, etc.
Syntactic Abstraction

- Hiding implementation details.
Side Effects

- Functions don’t take reference parameters.
- So only macros can modify variables that are passed as arguments.
Macro-Writing Utilities

• Certain patterns come up again and again in writing of macros, which can be abstracted away.

• Example: in macro definitions, it is very common to have a LET that introduces a few variables holding gensymmed symbols.

• Why not make a tool to automate this repetitive task?

• (defmacro do-primes ((var start end) &body body)
  (with-gensyms (ending-value-name)
    `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
         ,(ending-value-name ,end))
     ((> ,var ,(1+ ,var))
      (@body)))))
• Interface:
  (with-gensyms (var1 var2 ...) 
    body)

• Expansion:
  (let ((var1 (gensym)) 
        (var2 (gensym)) ...) 
    body)

• Definition:
  (defmacro with-gensyms ((&rest names) &body body) 
    `(let ,(loop for n in names collect `(,n (gensym))) 
        ,(\@body)))
• The classic ‘once-only’ macro generates code that evaluates the given macro arguments once only, in a particular order, and avoiding captures.

• `(defmacro do-primes ((var start end) &body body) (once-only (start end) ; evaluation order is given here `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))) ((> ,var ,end)) ,@body)))`

• Almost as simple as the original leaky version!
(defmacro once-only ((&rest names) &body body)
  (let ((gensyms (loop for n in names collect (gensym)))))
    `(let (_,(@(loop for g in gensyms collect `(`,g (gensym)))))
       `(let (_,(@(loop for g in gensyms for n in names collect `(`,g ,,n)))
           ,(let (_,(@(loop for n in names for g in gensyms collect `(`,n ,g)))
               ,@body)))))))

Better understood by examining its expansion.
How It Works

• (once-only (start end)
  `(do ((,var (next-prime ,start) ...))
      ((> ,var ,end))
      ,@body))

→

(LET (((#:G1191 (GENSYM)) ; avoid variable capture
       (#:G1192 (GENSYM)))
      `(LET (((#:G1191 ,START) ; evaluate only once, in order
              (#:G1192 ,END))
           ,(LET ((START #:G1191) ; use original names
                  (END #:G1192))
              `(DO ((,VAR (NEXT-PRIME ,START) ...))
                   ((> ,VAR ,END))
                   ,@BODY))))
Macros for Efficiency... Not

• (defmacro my-add (arg1 arg2)
  (if (and (numberp arg1) (numberp arg2))
    (+ arg1 arg2)
    `(+(arg1 arg2))))

• Better do this with compiler macros!
A Final Word

• The classic Common Lisp defmacro is like a cook’s knife: an elegant idea which seems dangerous, but which experts use with confidence.

• Not explained: symbol macros.
Important Literature

• Paul Graham, On Lisp - *the* book about macros (out of print, but see [www.paulgraham.com](http://www.paulgraham.com))


• Guy Steele, Growing a Language - keynote OOPSLA’98. Available at Google Video.